

Kapitel 2

Grundlagen der Serviceorientierung

*I have travelled the length
and breadth of this country
and talked with the best people
and I can assure you
that data processing is a fad
that won't last out the year.*

Editor Business-Books
Prentice-Hall
1957

Warum überhaupt Services und die Serviceorientierung? Heute bestehende, aber vor allen Dingen sämtliche zukünftige Systeme dürften von diversen Lieferanten zusammengestellt werden, aus Teilen, welche alle miteinander interagieren und so ein neues großes System aus Software erzeugen.¹ In einem solchen Umfeld ist es leicht, sich im Detail des einzelnen Geräts, Übertragungsprotokolls oder der Schnittstellendefinition zu verlieren.² Die Abstraktion von Geräten und Softwarebestandteilen in diesen Geräten in Form von Services ermöglicht es uns, solche Systeme auf Elementebene zu beschreiben, beziehungsweise die Beschreibung erst möglich zu machen. Daher eignen sich Services in einem solchen Gebiet als Beschreibungs- und Modellierungssprache. Die aus der Abstraktion auf Services entstehenden Vergrößerungen, als auch die Tatsache, dass die Elemente des beschreibenden Systems keine „echten“ Services sind, erzeugen zwar ein Maß an Unschärfe, können aber im Sinne des Verständnisses für das Gesamtsystem hingenommen werden.

2.1 Serviceorientierungsparadigma

Den Begriff des Service zu definieren ist nicht einfach, obwohl er zunächst intuitiv klar erscheint. Etymologisch betrachtet geht das Wort Service auf das Lateinische *servitium* mit der Bedeutung von Sklaverei und Knechtschaft zurück. Ein Service ist nicht deckungsgleich mit einer Komponente, wie man sie aus der komponentenbasierten Softwareentwicklung her kennt, wenngleich viele Konzepte auch auf die serviceorientierte Softwareentwicklung übertragen werden können. Eine andere

¹ Die heutigen Softwarehersteller erwecken zwar gerne den Eindruck, dass die Software, die sie liefern, vollständig von ihnen selbst stammt, aber meist ist dies nicht der Fall. Aus Marketinggründen scheint das Alles-Aus-Einer-Hand-Argument jedoch erfolgreich zu sein.

² Unsichere Menschen flüchten sich gerne in Details, da ihnen das subjektive Sicherheit bietet. Außerdem können sie hier ihr „Fachwissen“ beweisen. . .

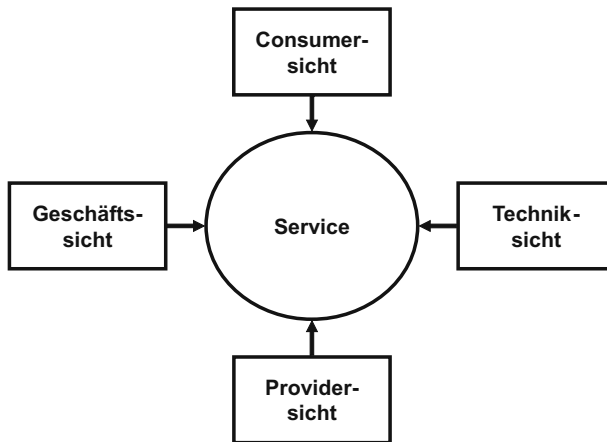


Abb. 2.1 Die unterschiedlichen Sichten auf einen Service

Vorstellung ist, dass ein Service von einer Komponente implementiert wird und der Service damit nur das Interface der Komponente darstellt. Diese Betrachtungsweise ist etwas zu eng. Ein Service beinhaltet zwar auch ein Interface, zusätzlich dazu existiert aber ein Vertrag³, der eine Beschreibung von Eigenschaften enthält, welche die Semantik betreffen und die nicht über ein Interface definiert werden können. Dazu gehören Qualitätseigenschaften wie Verfügbarkeit oder Performance, die auch im Sinne eines juristischen Vertrages vereinbart werden können. Dafür reicht dann die Beschreibung eines Interface nicht mehr aus, sondern es bedarf eines komplexen **Service Level Agreements (SLA)**, in dem Rechte und Pflichten von Provider und Consumer klar geregelt sind.

Die Nutzung unterschiedlicher Perspektiven ist wichtig für das Verständnis eines Service. Hier gibt es zwei wesentliche Sichtweisen:

- **Geschäftssicht** – Innerhalb der Geschäftssicht wird ein Service als Teil einer Geschäftstransaktion betrachtet, die in einem Vertrag beschrieben und die durch die Geschäftsinfrastruktur durchgeführt wird. Was ein Service leistet, ist eng verknüpft mit den Erfahrungen des Geschäftsbereichs. Einen Service auf dieser Seite bezeichnet man mit dem Begriff **Geschäftsservice** (genau wie eine Dienstleistung). Typische Eigenschaften eines solchen Geschäftsservice sind:
 - **Geschäftsvisibilität** – Ein Service muss etwas sein, wofür Consumer (Kunden)⁴ bereit sind, etwas zu bezahlen. Die Consumer ihrerseits erhalten etwas, das für sie von Wert oder von Nutzen ist. In diesem Zusammenhang muss der Begriff „Kunde“ weiter gefasst werden als im Bereich von Produkten. Als

³ Dieser wird meist bi- oder unilateral geschlossen.

⁴ In diesem Buch wird der Begriff Consumer für jeden beliebigen Abnehmer eines Service genutzt, dabei kann es sich um einen Menschen, ein anderes System, einen anderen Service, einen anderen Computer, denselben Service und so weiter handeln. Der Nutzer eines Interface wird stets als Consumer bezeichnet.

Consumer werden nicht nur externe, sondern auch interne Nutzer des Service bezeichnet.

- Technologie – Hierbei steht das, was implementiert werden soll im Fokus, nicht wie etwas implementiert werden soll.
- Kontext – Die Services werden neben anderen Eigenschaften auch durch ihren jeweiligen fachlichen Kontext definiert.
- Techniksicht – Ein Service stellt eine mehr oder minder geschlossene Funktionalität bereit, wobei die Semantik eines Service in Form eines Interface beschrieben wird. Eigenschaften technischer Services sind:
 - Technologie – Bei technischen Services steht die Technologie, also wie etwas implementiert werden soll, im Vordergrund.
 - Kontext – Ein Service ist eine vom Kontext gekapselte und abstrahierte Funktionalität.

Es existieren einige Unterschiede zwischen den Sichten und damit auch zwischen den daraus jeweils abgeleiteten unterschiedlichen Servicedefinitionen. Der wesentliche Unterschied beider Sichten liegt in der Betrachtung des Kontexts. Für Geschäftsservices ist der Kontext von großer Wichtigkeit; alle Organisationen arbeiten schließlich in einem dynamischen Marktumfeld. Werden Marktchancen ergriffen, so wird von dieser Seite erwartet, dass neue Services erstellt oder bestehende Services verändert werden. Die technische Seite ist bestrebt, den Kontext möglichst statisch zu halten, da nur so die Forderung nach Effizienz und Wiederverwendbarkeit erreicht werden kann. Diese Differenzen lassen sich nicht wirklich überbrücken. Die negativen Effekte lassen sich aber abmildern, indem kommuniziert wird, welche Bedeutung ein Service etwa auf organisatorischer Ebene hat oder wie wichtig Wiederverwendung für einen implementierten Service ist.

Zusätzlich zu den beiden Geschäfts- und Technikperspektiven existiert eine Consumer- und eine Providerperspektive. Die Implementierung ist Teil der Providersicht und muss für den Consumer von geringem Interesse sein. Die Providersicht überschneidet sich mit der Geschäftssicht und der technischen Sicht. Die Consumersicht ist mehr auf die Geschäftssicht fokussiert. Wenn ein Consumer einen Service in Anspruch nimmt, muss für ihn der Wert des angebotenen Service schon vorab erkennbar sein. Ist dies der Fall, stellt der Service eine Schnittstelle zwischen Geschäftswert und Implementierung dar. In einer sehr technisierten Umgebung verschwindet oft die Fähigkeit einen Service zu wählen, da solche Umgebungen auf die Anwesenheit oder Nichtanwesenheit von Services automatisch reagieren und dies für den Menschen nicht erkennbar ist.

Die Verwendung von Services in Geräten, Organisationen und in der Software folgen einem gemeinsamen Grundsatz, dem „Paradigma⁵ der Serviceorientierung“:⁶

⁵ Paradigma aus dem Griechischen *παράδειγμα* aus *παρα* (neben) und *δειγναι* (zeigen). Spötter behaupten ein Paradigma sei die Summe aller Vorurteile, die über etwas existiere.

⁶ Siehe auch Servicedefinition, S. 15.

Alle Funktionen in einem realen System, seien es Abläufe in Organisationen, Prozesse, Aktivitäten, Funktionen in Softwaresystemen, Applikationen, Teile von Applikationen oder Softwarefunktionen, lassen sich als Services darstellen und aus Services aufbauen!

Oder kürzer formuliert:

Alles, was aus- oder durchgeführt werden kann, ist ein Service!

Hierbei hat jeder Service mindestens einen Provider (den Lieferanten) und einen Consumer (den Kunden oder Benutzer). Außerdem ist jeder Service in seiner Funktionalität und seinen Randbedingungen vorab definiert und diese Bedingungen sind sowohl dem Provider als auch dem Consumer bekannt.

Die Idee des Service lässt sich von kleinsten Geräten – individuellen Chips – bis hin zu riesigen Organisationen einsetzen, eine Tatsache, welche von einem hohen Maß an Universalität des Serviceparadigmas zeugt. Der technische Fortschritt, speziell bei der Miniaturisierung führt dazu, dass unsere Umgebung immer stärker mit Computern durchsetzt ist. Diese Umgebung weist dadurch ständig mehr Funktionalität auf. Die heutige Standardsicht auf solche Umgebungen ist sehr technikzentriert – sie stellt das einzelne Gerät oder Protokoll in den Vordergrund – und wird oft auch auf das einzelne Gerät reduziert; dabei ist es viel erfolversprechender diese Funktionalitäten als Services zu modellieren und unsere technische Umgebung als ein „System of Systems“ oder ein System von Services zu verstehen.

Der Ursprung der Idee des Service liegt in den Dienstleistungen, dies hat zur Folge, dass es Unterschiede zwischen Produkten und Services gibt. Unabhängig von einer exakten Definition des Begriffs Service teilen sich in der Praxis alle Services eine Reihe von Eigenschaften:

- Services sind sofort nutzbar.
- Services sind modellierbar.
- Services haben ein wohldefiniertes Verhalten.
- Services haben definierte Ein- und Ausgaben.
- Services stellen Fähigkeiten oder Funktionen zur Verfügung.
- Services werden „gemanagt“, um nichtfunktionale Ziele zu erfüllen.
- Services sind zusammenbaubar⁷, um damit neue Services zu erzeugen.

Die Serviceidee unterscheidet sich von der klassischen Produktidee durch folgende Merkmale:

- Services sind nicht direkt greifbar. Ihr Wert ist oft nur schwer quantifizierbar, wobei die Nichtgreifbarkeit sich in zwei Dimensionen vollzieht. Zum einen sind Services physisch ungreifbar, das heißt sie können nicht berührt oder betastet werden, zum anderen sind sie mental ungreifbar, das heißt sie können nicht mental als ein Gegenstand⁸ verstanden werden.

⁷ Servicekomposition, s. Abschn. 2.7.4.

⁸ Ein Konto auf der Bank oder ein Lebensversicherungsvertrag ist zwar nicht physisch greifbar, beide stellen jedoch mentale Gegenstände dar und sind somit Produkte.

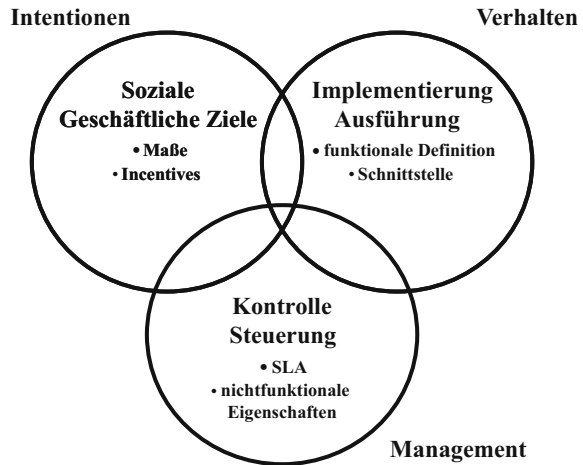


Abb. 2.2 Unterschiedliche Sichten auf Services

- Services sind heterogener und vielgestaltiger als Produkte, da sie oft von Menschen direkt erzeugt werden.
- Services werden fast gleichzeitig produziert und verbraucht, im Gegensatz zu Produkten, welche meistens eine Lagerhaltung haben. Zentralisierung und Massenproduktion sind für Services problematisch.
- Services sind „verderblich“, das heißt Services können nicht gespeichert werden, aber aus Sicht des Consumers kann das Ergebnis des Service durchaus unverderblich sein.
- Die Qualität eines Service hängt von vielen sehr schwer kontrollierbaren Faktoren ab. Da ein Schwerpunkt der Services im direkten Kundenkontakt besteht, hat dies zur Konsequenz, dass die Qualität des Service von der Fähigkeit des Kunden determiniert wird, sich zu artikulieren, beziehungsweise der Provider in der Lage ist, dem Kunden zuzuhören.
- On-demand-Delivery – Die Services werden meistens direkt durch die Nachfrage bezüglich einer zu erbringenden Leistung ausgelöst.

Damit diesen unterschiedlichen Charakteristika Rechnung getragen werden kann, muss eine umfassende Servicedefinition sehr abstrakt sein. Die hier gewählte Definition des Service ist daher:

Ein Service ist die Summe des beobachtbaren Verhaltens eines Systems (genannt Provider), gegeben durch die Menge aller möglichen Interaktionen und deren Relationen zwischen dem System und seiner Umgebung.

Diese Definition ermöglicht es, Services als systemtheoretische Gebilde (s. Anhang A) zu beschreiben. Dieses Servicekonzept ist das Resultat der Trennung zwischen dem internen (Implementation) und externen (Interface) Verhalten eines Systems. Für den Consumer (als Teil der Umgebung) ist nur der externe Teil interessant. Implizit lassen sich aus dieser Definition eine Reihe von Eigenschaften für die Services ableiten:

- Ein Service hat einen Grad an Autonomie, da ohne Autonomie ein System überhaupt nicht identifizierbar ist.
- Services besitzen ein Interface (eine Schnittstelle), dieses wird durch die Grenze zwischen System und Umgebung gebildet. Da der Consumer ein Teil der Umgebung ist⁹, bildet seine Umgebungsteilmenge das für ihn wahrnehmbare Interface. Der andere Teil der Grenze zur Umgebung bildet für die Serviceimplementierung den Kontext.
- Da Systeme zu größeren Systemen zusammengesetzt werden können, gibt es auch die Möglichkeit zur Servicekomposition.
- Die funktionalen Eigenschaften sind die erwarteten Verhaltensmuster des Systems und ergeben sich aus dem Interface.
- Die nichtfunktionalen Eigenschaften sind die durch den Kontext (ohne den Consumer) ausgelösten Verhaltensmuster des Systems.

Ein Service muss hinreichend gut beschrieben werden. Speziell dann, wenn der Consumer den Provider nicht kennt, ist eine semantisch reichhaltige und strukturell gute Beschreibung notwendig, damit ein Service auch entdeckt und nachfolgend eingesetzt werden kann.

Die funktionalen Eigenschaften eines Service beschreiben genau die fachlich gewünschten Funktionen, die der Service erfüllen soll; das, was er tatsächlich für seinen Kunden durchführt.¹⁰ Die funktionalen Eigenschaften beschreiben die Wirkung, nicht die Implementierung des Service, folglich wird hier die Grenze zur Umgebung und nicht die Substruktur des Systems beschrieben. Neben der Festlegung von dem, was durchgeführt werden soll, wird auch der Informationsfluss für den Service beschrieben. Services, die durch Menschen durchgeführt werden, wie zum Beispiel Reinigung, Outsourcing oder Reparaturen, werden im Allgemeinen auch als manuelle Services oder Dienstleistungen bezeichnet. Bei manuellen Services gestaltet sich der Informationsfluss in Form eines Dialogs zwischen dem Kunden (Consumer) und dem Dienstleister (Provider). Die starke menschliche Verflechtung und das iterative Verhalten aller Beteiligten ermöglichen es, dass sich diese Services auf diverse Kontexte einstellen können.¹¹ Im Fall von Software werden die funktionalen Eigenschaften von Services durch Interfaces, sowie die Vor- und Nachbedingungen spezifiziert. Services in der Software erzeugen ein eindeutiges syntaktisches Verhalten auf Grund der Tatsache, dass die genutzten Programmiersprachen Anforderungen an den Typ der Information in Form von Datentypen stellen – im Gegensatz zu manuellen Services.

Eine Servicebeschreibung besteht neben der Funktionalität des Service auch aus der Beschreibung der nichtfunktionalen Eigenschaften. Speziell das Fehlen von Beschreibungen nichtfunktionaler Eigenschaften verhindert eine „vernünftige“ Suche und Entdeckung von Services. Sinnvolle nichtfunktionale Eigenschaften für einen Service sind:

⁹ Aus Sicht der Implementierung.

¹⁰ Allerdings nur aus Sicht des Kunden.

¹¹ Viable System Services sind der Versuch, diese Flexibilität auch auf Software zu übertragen, s. Anhang B.1.

- Name des Providers – Provider müssen eine eindeutige und verifizierbare¹² Identität haben. Die logische Folge einer Identität ist die Existenz eines eindeutigen Namens und die Zugehörigkeit zu einer Organisation.
- Zeit – Angaben über die Servicezeiten oder auch Wochentage sind wichtige Größen für den Consumer.¹³
- Sprache –
- Qualität des Service –
- Sicherheit –
- Ort – Obwohl es in den meisten Fällen transparent sein sollte, wo der Service tatsächlich ausgeführt wird, kann der Ausführungsort bezüglich rechtlicher Belange oder Sicherheitsaspekte durchaus relevant sein. Bei nicht-IT-basierten Services kann es durchaus sein, dass der Service nur an einem bestimmten Ort durchgeführt werden kann oder darf. Neben den klassischen Formen der Ortsangabe können Telefonnummern oder URI-Adressen bei SLA-Verletzungen Kontakte und Möglichkeiten zum Ausweichen anbieten.
- Verfügbarkeit – Unter Verfügbarkeit versteht man die Kombination aus den zeitlichen und örtlichen Aspekten der Services.
- Obligation – Stellt die Verpflichtungen von Provider und Consumer dar.
- Preis – Neben dem Preis pro Serviceaufruf sind auch andere Formen von Preismodellen möglich, so zum Beispiel Flat, Bulk oder Prime Rate.
- Zahlungsmodalitäten – Analog zum Preis der Services kann auch die Art und Weise der Zahlung relevant sein.
- Strafen – Werden Zahlungsmodalitäten oder Obligationen verletzt, so treten die entsprechenden Strafen ein.

Für die Consumer sind die funktionalen und nichtfunktionalen Eigenschaften von Services zum Teil nicht voneinander unterscheidbar, insofern ist eine solche Unterteilung bis zu einem gewissen Grad willkürlich. Je bekannter und standardisierter die funktionalen Eigenschaften von Services sind, desto stärker rücken die nichtfunktionalen Eigenschaften bei der Entscheidung des Kunden für einen spezifischen Provider in den Vordergrund.

Der Ansatz, eine Organisation mit allen Prozessen und der gesamten Menge an Aktivitäten und Software durch ein Netz von miteinander wechselwirkenden Services zu modellieren, führt dazu, dass Organisationen als ein Graph von Services dargestellt werden können. Diese haben auf abstrakter Ebene ähnliche Charakteristika und bilden auf Grund ihrer Autonomie in sich geschlossene Systeme. Daher kann die Organisation als ein System von Systemen betrachtet werden. Auf ein solches Konstrukt lassen sich dann auch die systemtheoretischen Methodiken anwenden. Aber nicht nur Organisationen, auch unsere Umgebung lässt sich als ein solches „System of Systems“ modellieren und durch die entsprechenden systemtheoretischen Instrumente steuern.

Die gesamte technologische Entwicklung der letzten Jahre strebt in diese Richtung; durch die zunehmende Bandbreite und Geschwindigkeit des Informations-

¹² Die Verifikation kann auch über Dritte, so zum Beispiel Trustcenter, erfolgen.

¹³ Der Service Personentransport der Deutschen Bahn kennt Wochen- und Feiertagsfahrpläne.

austauschs, als auch der stark gestiegenen Automatisierung, sind heutige Systeme deutlich stärker miteinander vernetzt, als dies je zuvor der Fall war. Der Endpunkt dieser Entwicklung ist dann ein Ultra Large Scale System (s. Abschn. 6.9). Mittlerweile sind Phänomene wie Emergenz und nichtlineare Skalierbarkeit in großen Organisationen und in technisierten Umgebungen merklich; diese Phänomene lassen sich nicht durch die Betrachtung einzelner Geräte, Applikationen oder Teilsysteme erklären, hier bedarf es des Einsatzes der Systemtheorie. Mit zunehmender Abhängigkeit und Wachstum der IT-Systeme treten Fragestellungen in Richtung der Überlebensfähigkeit solcher Konstrukte in den Vordergrund, womit große serviceorientierte Systeme immer stärker natürlichen oder sozialen Systemen ähneln und Fragestellungen nach Selbsterhalt, Selbstorganisation und Selbstreproduktion des jeweiligen Systems immer interessanter werden.

2.2 Service Oriented Enterprise

Eine Organisation lässt sich am besten analysieren, wenn nicht die formale Struktur gesehen, sondern die Kommunikationswege und -stile betrachtet werden. Dies liegt darin begründet, dass aus systemtheoretischer Sicht alle Teile einer Organisation – seien es Menschen oder ganze Suborganisationen – folgende notwendigen Bedürfnisse haben:

- Die Elemente müssen ihre Tätigkeiten synchronisieren. Schließlich verkörpern Organisationen auch Prozesse, beziehungsweise dienen dazu bestimmte Prozesse zu leisten, dies geht nicht ohne Formen der Koordination mit den Basistypen: Orchestration und Choreographie (s. S. 46).
- Die Elemente führen Diskussionen über Ziele, beziehungsweise kommen überein bestimmte Ziele zu verfolgen.
- Es müssen Fehlfunktionen entdeckt und kommuniziert werden, damit Reaktionen auf diese Fehlfunktionen stattfinden können.

Folglich entsteht eine Organisation erst dann, wenn sie kommuniziert! Umgekehrt betrachtet kann man aber auch eine Organisation als Ergebnis von Kommunikation betrachten und folglich die Kommunikationsformen und -wege als ein Architekturmodell der Organisation nutzen.

Die unterschiedlichen Stadien einer Organisation in ihrer chronologischen Erscheinung sind in Abb. 2.3 dargestellt. Die Evolution von Organisationen ist in aller Regel problemgetrieben, das heißt die Organisation existiert so lange in einem Stadium, bis die nächste Krise eintritt. Die Krisenlösungen sind immer kontrollgetrieben: Zunächst werden Kontroll- und Steuerungsmechanismen ausgebaut, gefolgt von den operationalen Aspekten. Nicht alle Organisationen passen exakt in dieses Schema, oft beinhalten die Organisationen Mischformen oder haben etwas andere Ausprägungen der verschiedenen Stadien, aber es ist hilfreich Stereotypen zu betrachten, damit der organisatorische Kontext überhaupt klar wird. Neben den unterschiedlichen Stadien ist bei den meisten Organisationen – langfristig gesehen –

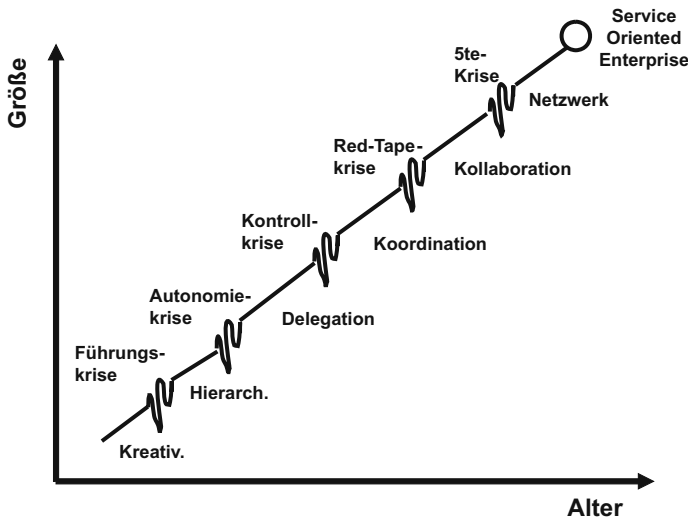


Abb. 2.3 Die Evolutionsstufen einer Organisation

ein Größenwachstum zu beobachten. Üblicherweise korrelieren die Größe mit dem Alter und der Führungsstil mit der Größe. Letzteres resultiert aus der Notwendigkeit einer effektiven Kommunikation und Steuerung von großen Organisationen. Im Laufe ihrer Entwicklung durchlaufen die meisten Organisationen verschiedene Stadien (wenn sie lange genug überleben):

- **Kreativitätsstadium** – Das erste Stadium einer Organisation ist das Kreativitätsstadium. Dieses ist dadurch charakterisiert, dass es eine kleine Zahl von Mitarbeitern gibt, welche in aller Regel sehr informell miteinander kommunizieren.
- **Hierarchiestadium** – Die nächste Stufe ist die der funktionalen Organisation. Es bilden sich immer mehr Suborganisationen heraus, welche jeweils spezifische Aufgaben erledigen. Im Laufe der Zeit werden explizite Hierarchien¹⁴ geformt. Die Suborganisationen beginnen oft als Abteilungen und werden im Laufe zunehmenden Wachstums erst als Profitcenter, dann als Einheiten ausgegründet. Charakteristisch ist ein hohes Maß an Machtasymmetrie innerhalb der Organisation. Es werden definierte formale Kommunikationsstrukturen herausgeformt, Buchhaltung und Verwaltungseinheiten zeigen sich als Substrukturen. Auf Grund der Hierarchie werden alle Entscheidungen zentral gefällt und der Fokus der Führung ist auf Effizienz der Operationen, sowie auf einen hohen Grad an Professionalität innerhalb der Organisation gerichtet.
- **Delegationsstadium** – Das Hierarchiestadium wird durch die Delegation von Macht an die einzelnen Organisationsteile aufgelöst. Diese verselbstständigen sich und werden zu sogenannten „Divisionen“. Jede einzelne Division ist alleine

¹⁴ Aus dem Griechischen *τεραρχία*, setzt sich zusammen aus *Hieros* = heilig und *Archos* = herrschen. Ursprünglich bedeutete es *Herrschaft durch den Priester*. Der Begriff wurde auf alle Organisationen verallgemeinert.

für alle Operationen und Geschäftsergebnisse im Markt verantwortlich (analog zum Kreativitätsstadium). Die Aufgabe des Topmanagements in einer solchen Organisation ist es primär, die langfristige Strategie sowie aktive Akquisitionen von neuen Divisionen zu betreiben. Intern sind die Divisionen meistens hierarchisch oder sogar in Subdivisionen organisiert. Jede einzelne Division ist innerhalb ihres Marktes auf Expansionskurs und versucht ein lokales Optimum bezüglich Profit und Marktanteilen zu erreichen, was in einigen Fällen zu einer innerorganisatorischen Konkurrenz führen kann.

- Koordinationsstadium – Dies ist eine Organisationsform, welche dezentral den Divisionen die Durchführung und die Verantwortung überträgt, aber zentral plant und standardisiert. Trotz organisationsweiter Standards und formalisierter Mechanismen bleibt die Divisionsstruktur faktisch vorhanden. Der Fokus des Topmanagements ist es, die einzelnen Divisionen zu Kooperationen zu bewegen und damit eine neue und wiederum „kohärente“ Organisation zu erschaffen.
- Kollaborationsstadium – In diesem Stadium der Entwicklung wird die Zentrale entmachtet und die Gesamtorganisation erhält eine Matrixstruktur. Der Fokus des Topmanagements liegt auf dem Lösen von Problemen und der Einführung von Innovationen in der Gesamtorganisation. Es kommt zu Teambildungen, welche funktionsübergreifend organisiert werden.
- Netzwerkstadium – Meist treten verschiedene Organisationen als Konkurrenten um einen gemeinsamen Markt auf. Konkurrenz ist aber nur eine mögliche Beziehung zwischen zwei Organisationen, eine andere Form ist die Kooperation. Wenn ein Netzwerk wächst, erreicht es irgendwann einmal eine kritische Masse. Dann zieht es neue Teilnehmer an, erzeugt einen hohen Mehrwert für alle Beteiligten und ist anschließend in der Lage, kleinere Netzwerke zu verdrängen. Innerhalb des Netzwerkstadiums besteht die Gesamtorganisation aus einem Netz von hochgradig autonomen Organisationen, welche eindeutige Services und Produkte mit einer klar umrissenen Funktionalität anbieten. Die Kooperation dieser einzelnen Teile geschieht auf einer Ad-hoc-Basis, um gemeinsam ein Ziel zu erreichen. Eine besondere Form eines solchen Netzwerks ist unter dem Begriff „virtuelles Enterprise“ bekannt. Eine der großen Stärken einer solchen Organisationsform liegt darin, dass verteilte Organisationen einen Teil der vertikalen Bürokratie innerhalb einer einzelnen Organisation ersetzen.

Die Beurteilung einer Organisation geschieht meist in Form eines Graphen, bei dem N_{Knoten} -Knoten über N_{Kanten} miteinander verbunden sind. Hierbei beträgt die minimale Zahl von Kanten: $N_{\text{Kanten}} = N_{\text{Knoten}} - 1$, während die längste Kette in solchen streng hierarchischen Bäumen durch:

$$L \sim \ln N_{\text{Knoten}}$$

gegeben ist. Eine solche Organisationsform ist jedoch nur sinnvoll, wenn sich die Prozesse und Funktionen der Organisation in Teilfunktionen und -prozesse zerlegen lassen und diese mehr oder minder unabhängig von den Suborganisationen ausgeführt werden können. Dies widerspricht in den meisten Fällen jeder Realität! Heutige Organisationen müssen Aufgaben erledigen, welche nicht klar definiert sind,

welche oft die gleiche Zeitskala haben wie die Produktion¹⁵, welche ihr Wissen erst durch den Produktionsprozess selbst aufbauen. Bei solchen Problemstellungen ist eine Zerlegung faktisch nicht mehr möglich, die Arbeit wird hier sehr stark kooperativ erledigt.

Moderne Organisationen haben daher neben der Effizienz auch das Ziel eine Form zu finden, welche der Gesamtorganisation ein hohes Maß an Robustheit gibt. Denn der einzelne Knoten im Netzwerk muss vor Überlastung durch Arbeit, nicht-antizipierten Kollaborationsmustern oder Reorganisation geschützt werden. Gleichzeitig muss aber auch die Gesamtorganisation vor einem Zerbrechen bewahrt werden. In komplexen Netzwerken werden die Informationen zur Koordinierung oft über Mittler weitergegeben, welche dadurch zusätzliche Informationslasten haben. In sehr großen Netzen, wie zum Beispiel dem Internet, ist die Last durch die Koordinierungsnachrichten immens hoch (s. Abschn. 6.7). Solche Netze haben das Risiko einer „Verstopfung“, auf das aktiv reagiert werden muss. Neben der Robustheit gegen Überlastung muss ein stabiles Netzwerk auch robust gegen den Wegfall einer Kante oder eines Knotens sein. Netzwerke, die in beiden Fällen robust sind, werden als ultrarobust bezeichnet.

In dem Modell von Dodds¹⁶ wird ein allgemeines Netzwerk nach zwei Parametern charakterisiert: λ und ζ (s. Abb. 2.4). In diesem Modell existieren zwei für den einzelnen Knoten interessante Größen: Zum einen der Abstand zwischen zwei Knoten i und k mit:

$$x_{ik} = \sqrt{d_i^2 + d_k^2 - 2}, \quad (2.1)$$

und die Tiefe D_{ik} des kleinsten gemeinsamen Vorgängers a_{ik} . Durch diese Betrachtungsweise entstehen zwei charakteristische Skalen: die Tiefe λ und die Breite ζ . Kanten zwischen zwei Knoten entstehen nach einer Wahrscheinlichkeitsverteilung durch:

$$p(i, k) \sim e^{-\frac{1}{\lambda} D_{ik}} e^{-\frac{1}{\zeta} x_{ik}}. \quad (2.2)$$

Die entstehenden Graphen der Grenzfälle lassen sich in mehrere Kategorien, je nach Parameterbereich (λ, ζ) , einteilen:

- $(\lambda, \zeta) \mapsto (\infty, \infty)$ – Der Zufallsgraph, hier sind alle Kanten rein zufällig und gleichmäßig verteilt.
- $(\lambda, \zeta) \mapsto (\infty, 0)$ – Das lokale Team, Kanten sind nur bei direkten Nachbarn und mit direkten Eltern vorhanden.
- $(\lambda, \zeta) \mapsto (0, \infty)$ – Die Divisionsorganisation, Kanten sind nur innerhalb einer Division erlaubt.
- $(\lambda, \zeta) \mapsto (0, 0)$ – Nur die Wurzel kann sich direkt mit einem weiteren Knoten verbinden, eine Art Zentralknoten in Form eines Sterns entsteht.

¹⁵ Typisch für Projekte, oft unterscheidet sich die Dauer des Aufbaus der Projektorganisation nur wenig von der Dauer des Projekts.

¹⁶ Dodds, Information exchange and the robustness of organizational networks, 2003.

Wirklich stabile Netzwerke entstehen in der Nähe von $(\lambda, \zeta) \approx (0.5, 0.5)$, welche auch ultrarobust sind. Dieses einfache Modell zeigt, dass sich nur bestimmte Netzwerkformen auf Dauer als sinnvoll herausstellen.

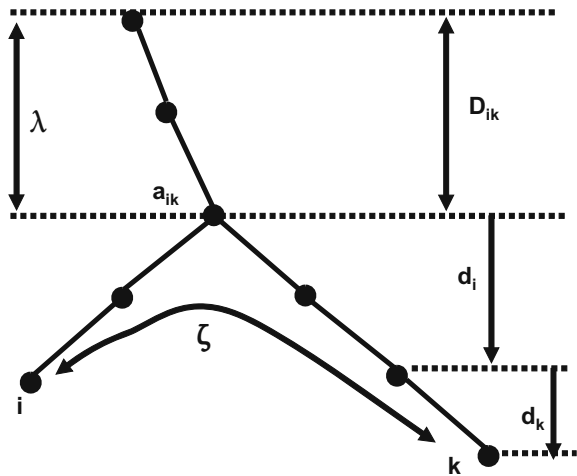


Abb. 2.4 Allgemeiner Graph eines hierarchischen Netzwerks nach Dodds

Ein virtuelles Enterprise ist strenggenommen eine Organisationsform, welche unabhängige Partner vereint, um einen einmaligen Auftrag zu erfüllen und danach wieder aufgelöst wird. Ein virtuelles Enterprise wird versuchen, für jede Teilaufgabe den bestmöglichen Provider einzusetzen, dessen Kernkompetenz identisch mit der Aufgabe sein sollte. Zu den daraus resultierenden Problemen zählen effiziente und durchaus verletzbare Verbindungen zwischen den einzelnen Organisationen. Diese Verbindungen müssen in Bezug auf Flexibilität und Versatilität ein ungleich höheres Maß an Anforderungen erfüllen als das sonst, im Rahmen einer normalen Organisation, notwendig ist. Als Organisationsform zerfallen die virtuellen Enterprises in first- und second-level Organisationen. Zu den typischen first-level virtuellen Enterprises zählen Projekte innerhalb eines Konzerns (s. Abb. 2.5), während eine echte virtuelle Organisation zum second-level Typ gehört. Die zeitliche Begrenzung des virtuellen Enterprises ist ein Schlüsselement zur Unterscheidung der beiden Typen.

Ein virtuelles Enterprise existiert mit zwei Basistopologien:

- Mit einem Provider, der den Beteiligten alle Infrastrukturfunktionen zur Verfügung stellt – so zum Beispiel für eine Open Source Entwicklung bei Sourceforge.
- Mit einem Provider, der eine erweiterte Plattform zur Koordination der individuellen Organisationen, die sich am virtuellen Enterprise beteiligen, zur Verfügung stellt.

Da das virtuelle Enterprise auf der freiwilligen Kooperation unabhängiger Partner basiert, fehlen im Vergleich zu klassischen Organisationsformen einige Elemente. Welche im Detail fehlen, kann von Fall zu Fall variieren, was dazu führt, dass

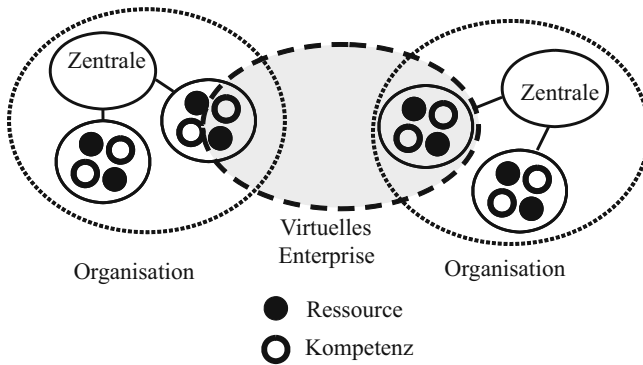


Abb. 2.5 Das virtuelle Enterprise

ein weites Spektrum unterschiedlicher Organisationsformen dem Konzept des virtuellen Enterprises zugerechnet werden kann. Es reicht von einmaligen, temporär befristeten Ad-hoc-Verbindungen bis hin zum eher längerfristig orientierten Netzwerkpool, dessen Mitglieder sich für die Leistungserstellung immer wieder neu zu virtuellen Enterprises zusammenschließen. Dabei bildet das virtuelle Enterprise nicht wirklich eine eigentliche Organisationsform und kann daher nur unzureichend über entsprechende Strukturmerkmale definiert werden. Die idealtypische Form des virtuellen Enterprises verzichtet auf die Institutionalisierung zentraler Funktionen zur Gestaltung, Lenkung und Entwicklung der Zusammenarbeit. Anstelle der dadurch fehlenden formalen Regelungen für die Koordination unter den Beteiligten treten informelle, fallweise abgestimmte Vereinbarungen. Dieser Mangel an zentralen Strukturen wird durch die Vernetzung der Kooperationspartner mittels Software ausgeglichen. Daher kann ein virtuelles Enterprise als eine interorganisationale, softwaregestützte Verbindung von Organisationen angesehen werden.

Virtuelle Enterprises sind Ad-hoc-Netzwerke¹⁷, die zum Zweck der Erfüllung einer spezifischen, genau festgelegten Leistungserstellung gegründet werden, um sich nach Erreichen der Zielsetzung wieder aufzulösen. Für die Erfüllung der Aufgaben stellen die einzelnen Kooperationspartner ihre Kernkompetenzen und ihre Ressourcen zur Verfügung. Bei virtuellen Enterprises dominiert die Ablauf- gegenüber der Aufbauorganisation. Die Konturen zwischen den virtuellen Strukturen, den jeweils beteiligten Organisationen sowie den Systemgrenzen sind einem ständigen Wandel unterworfen, wodurch sie zunehmend verschwimmen und oft nicht mehr klar wahrzunehmen sind. Außerdem zeichnet sich in dieser arbeitsteiligen Leistungserbringung das virtuelle Enterprise durch ein einheitliches Auftreten gegenüber Kunden und Konkurrenten aus und erscheint dadurch für den Außenstehenden wie eine real existierende Organisation. Die zeitliche Befristung der Zusammenarbeit im virtuellen Enterprise sowie die fehlende Absicherung der beteiligten Organisationen durch detaillierte formale Regelungen begünstigen jedoch auch eine opportunistische Haltung.

¹⁷ Siehe auch MANETs, S. 115.

Gerade kleineren und mittleren Organisationen bieten derartige virtuelle Enterprises die Möglichkeit auf die Globalisierung zu reagieren, indem sie sich zu strategischen Kooperationen zusammenschließen. Durch den entstehenden Kooperationsverband ist es den Organisationen möglich auf den Markt zu reagieren und multinationale, zweckorientierte Netzwerkzusammenschlüsse aufzubauen, ohne hohe Investitionen für einzelne Wertschöpfungsprozesse zu tätigen.

Tabelle 2.1 Virtuelle Enterprises, Chancen und Risiken

Dimension	Chancen	Risiken
Kompetenz	Kombination nur der besten Partner	Einseitiger Kompetenzaufbau
Effizienz	Redundanzarme und virtuelle Struktur	Erhöhte Koordinations- und Infrastrukturkosten
Flexibilität	Raumzeitliche Entkoppelung der Prozesse	Massive Personenabhängigkeit im Netzwerk
Motivation	Erfüllung individueller Bedürfnisse	Verlust der sozialen Identität
Koordination	Zentral oder selbstbestimmt	Verlust der Organisationsidentität

Virtuelle Enterprises lassen sich an Hand dreier stereotyper Ausrichtungen gut aufzeigen:

- **Dell** – Dieses amerikanische Unternehmen produziert nichts selbst. Alle Produkte sowie der Zusammenbau innerhalb der Produktion werden durch Dritte gefertigt. Dell ist jedoch immer bestrebt, eine stabile Partnerschaft mit seinen Lieferanten einzugehen und sichert diese auch vertraglich ab. Bei diesem Stereotyp spricht man auch von einem permanenten virtuellen Enterprise.
- **Filmproduktion** – Ein anderes Beispiel für ein virtuelles Enterprise ist die Filmproduktion. Hier kommen verschiedene Schauspieler, Kameralleute, Regisseure und viele andere mehr temporär für die Dauer der Filmproduktion als Team unter der Leitung des Produzenten zusammen. Am Ende zerfällt die Produktion wieder in die individuellen Freiberufler.¹⁸ Filmproduktionen sind das „klassische“ Beispiel für virtuelle Enterprises.
- **Open-Source-Projekte** – Die Open-Source-Projekte sind ein weiteres Beispiel für virtuelle Enterprises. Es handelt sich zwar nicht um kommerzielle Organisationen, trotzdem stellen sie virtuelle Enterprises dar. Hier arbeiten projektbezogen eine Reihe von Softwareentwicklern auf zeitlich befristeter Basis zwecks eines gemeinsamen Ziels zusammen.¹⁹

Eine der Schwierigkeiten hinter der Modellierung von virtuellen Enterprises besteht darin, dass es nicht „das“ virtuelle Enterprise gibt. Diese Organisationen verändern sich viel zu schnell, daher ist man nicht in der Lage, sie generisch zu beschreiben. Es gibt jedoch ein Charakteristikum, welches in allen Erscheinungsformen vorkommt: Alle virtuellen Enterprises nutzen das Internet als Trägermedium für ihre

¹⁸ Obwohl dies alles bewusst temporär angelegt ist, schafft es die Filmindustrie, Kinohits zu produzieren.

¹⁹ Das Betriebssystem *Linux* ist durch ein solches virtuelles Enterprise entstanden.

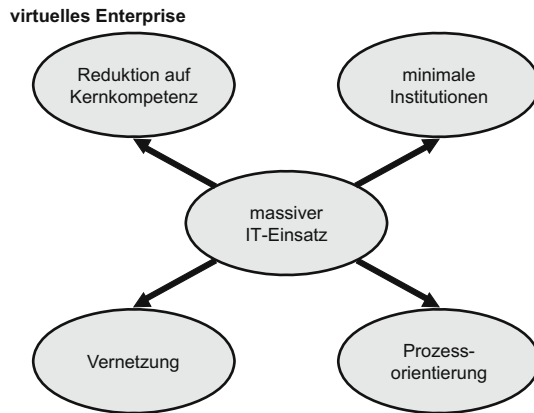


Abb. 2.6 Die typischen Merkmale eines virtuellen Enterprises

gemeinsamen Prozesse. Innerhalb des Netzwerkstadiums tritt ein weiteres Problem auf: Das virtuelle Enterprise ist ohne eine verlässliche und verteilte IT-Infrastruktur nicht denkbar, aber es existiert keine singuläre Institution, die Entscheidungen über die Infrastruktur sowie die Software oder die Systeme im Allgemeinen trifft. Folglich ist nicht nur eine verteilte Architektur, beziehungsweise eine generische Architektur, welche verschiedene Implementierungsformen verkraften kann, notwendig, sondern auch die Verteilung der Ressourcen und des Aufwands auf die unterschiedlichen Organisationen spielt eine Rolle. Die typische IT-Infrastruktur eines virtuellen Enterprise muss in der Lage sein, folgende Ereignisse reibungslos verkraften zu können:

- Eine Organisation wird neu aufgenommen.
- Eine Organisation verlässt das virtuelle Enterprise.
- Die Services des virtuellen Enterprise ändern sich spontan.
- Das virtuelle Enterprise verschmilzt mit einem anderen virtuellen Enterprise.

Außerdem darf die Infrastruktur weder von der Zahl der beteiligten Organisationen noch von dem Vorhandensein einer einzelnen Teilnehmerorganisation abhängig sein.

Es findet im Laufe der Evolution innerhalb der Organisationen eine Verschiebung zu immer stärker netzwerkartig funktionierenden Formen statt. Eine solche Netzwerkorganisation bedarf aber auch einer entsprechenden Strukturierung. Diese Strukturierung kann durch Services erreicht werden. Die Services werden typischerweise in einer sich rasch verändernden Umgebung zur Verfügung gestellt. Eine Organisation, die sich komplett in Services organisiert hat, wird als **Service Oriented Enterprise (SOE)** bezeichnet. Das SOE ist die logische Weiterentwicklung des virtuellen Enterprise, in dem es ein virtuelles Enterprise ist, welches ausschließlich aus Services besteht.

Der primäre Unterschied zwischen einem SOE und anderen mehr traditionellen Organisationsformen ist die Forderung, dass das SOE seine Kernprozesse definiert

Tabelle 2.2 SOE im Vergleich zur traditionelle Organisation

	Traditionelle Organisation	Service Oriented Enterprise
IT-Rolle	IT hat eine unterstützende Rolle. Fachbereiche müssen sicherstellen, dass die IT die Requirements erfüllt.	IT hat eine strategische Rolle. Die Services spiegeln die Geschäftsprozesse wider.
Wertschöpfung	Der Mehrwert wird in jedem Teil der Wertschöpfungskette primär in der Organisation erzeugt.	Ein Teil der Wertschöpfung findet in einem Netz aus Partnern statt. Der Mehrwert wird durch die Nutzung von Services erzeugt.
Requirements	Requirements werden von der gleichen Suborganisation gestellt, die sie auch implementiert.	Trennung der Requirements (Interface) von der Implementierung (Provider).
Prozessablauf	Sequenzieller Ablauf mit kumulativer Wertschöpfung.	Netzwerkartiger Ablauf oft mit paralleler Ausführung.
Prozessdesign	Meist reine Dekomposition. Statischer sequenzieller Ablauf.	Dynamisches Prozessdesign basierend auf den Ergebnissen der Subprozesse und Events.
Struktur	Hierarchie	Netzwerk
Broker	–	Notwendig für den Providerwechsel.
Semantik	Semantische Inseln in den Applikationen (Code) und Suborganisationen (Mitarbeitern)	Organisationsweite und -übergreifende Semantik durch Ontologien.

und für den Markt öffnet. Dabei erzeugt die Nutzung von Standardisierung ein hohes Potenzial an Interoperabilität. Diese Restrukturierung ermöglicht es, sehr schnell auf Veränderungen der Umgebung zu reagieren und sich dabei immer wieder neu zu „erfinden“.

Im Vergleich zwischen traditionellen Organisationen und SOEs gilt:

- **Bindung** – Traditionelle Organisationen haben eine statische Struktur und die Suborganisationen sind auf langfristige Beziehungen zu anderen Organisationen ausgelegt, so dass jede Veränderung hohen Aufwand produziert. Im Gegensatz dazu benutzen SOEs kurzfristige Beziehungen mit den Services von eventuellen Partnern, Veränderungen geschehen häufiger und werden durch Marktanforderungen ausgelöst.
- **Effizienz** – Traditionelle Organisationen werden optimiert, um interne Abläufe möglichst effizient zu gestalten. Allerdings sind die Wechselwirkungen mit anderen Organisationen ad-hoc und sehr arbeitsintensiv. Das Ziel hinter einem SOE ist es, Geschäftslogik mit der Zielsetzung von Benutzerzufriedenheit und -profitabilität zu erreichen. Eine solche Betrachtungsweise impliziert, dass interne wie externe Abläufe explizit betrachtet werden.
- **Koppelung** – Enge Koppelungen sind ein Charakteristikum für alle Formen der klassischen Organisation, folglich sind solche Organisationsformen für langfristige Operationen ausgelegt und widersetzen sich einem Wandel. Im Gegensatz

dazu muss ein SOE in der Lage sein, sich sehr kurzfristig zu verändern, was nur durch lose Koppelung zu erreichen ist, da so ein Wandel deutlich einfacher wird.

- **Zielsetzung** – Eines der Ziele einer traditionellen Organisation ist es, das Dokumentenmanagement durch effektiven Austausch der Dokumente zwischen den Beteiligten zu ermöglichen. Die Folge dieser Dokumentenorientierung ist eine Abkehr von der Kundenorientierung, ein SOE hingegen ist auf Nachfrage und Kundenzufriedenheit ausgerichtet.
- **Identität** – Integration in herkömmlichen Organisationen ist Bottom-Up. Partner sind bekannt, bevor sie integriert werden. Im Gegensatz dazu erlaubt ein SOE eine Top-Down-Integration, da das Ziel des SOE a priori bekannt ist und dies die Selektion der entsprechenden Services steuert.
- **Semantik** – SOEs sind im Gegensatz zu traditionellen Organisationen nicht atomar, sondern hierarchisch geschachtelt (s. Anhang A). Folglich sind ihre inneren und äußeren Beziehungen sehr viel semantikreicher als es bei herkömmlichen Organisationen der Fall ist.

Traditionelle Organisationen haben Schwierigkeiten, sich auf rasche Veränderungen einzustellen. In einem SOE sollte dies deutlich einfacher sein, zumindest auf den ersten Blick. SOEs haben jedoch auch die typischen Probleme von Netzwerkorganisationen (s. Abschn. 2.2) und einige mehr, da der permanente Wechsel die Identität der Organisation und damit auch das Zugehörigkeitsgefühl der Mitarbeiter stark beeinträchtigen kann. Ein möglicher Ausweg für die SOEs ist es, eine VSM-Struktur (s. Anhang B) zu wählen und so, trotz hoher Agilität, eine strukturelle und verhaltensbedingte Stabilität zu erreichen.

Ein SOE lässt sich wie folgt definieren:

Ein Service Oriented Enterprise (SOE) ist eine temporäre und dynamische Kollaboration zwischen autonomen Services, welche gemeinsam Services für Consumer in einer sich ändernden Umgebung bereitstellen. Ein SOE wird durch das Tupel (Ziele, Entitäten, Relationen, Umgebung, Lebenszyklus) eindeutig identifiziert.

- **Ziele** – Das Ziel des SOE ist das zu erreichende Geschäftsziel. Üblicherweise stellt es die Services dar, welche genutzt werden müssen, beziehungsweise Kriterien oder Schwellen für die jeweilige Serviceauswahl.
- **Entitäten**²⁰ – Die Menge an „Dingen“, welche die Organisation bilden. Hierbei kann es sich um Services oder auch echte Suborganisationen handeln.
- **Relationen** – Die Abbildungen der Funktionen und Kardinalitäten jeder Entität in Bezug auf andere Entitäten innerhalb des SOE.
- **Umgebung** – Die Umgebung setzt sich aus einer Menge von Entitäten zusammen, welche das SOE beeinflussen.
- **Lebenszyklus** – Der Lebenszyklus gibt die aktuelle Phase an, in der sich das SOE befindet.

²⁰ Eine Entität im Sinne des SOE kann diverse Formen annehmen: Management, Service, Consumer, Provider, oder Broker.

Für jede Form der Serviceerstellung ist es wichtig, die Zusicherungen (Commitments) sauber zu planen und zu dokumentieren. Am besten funktioniert dies in Zusammenarbeit mit dem Kunden des Service; die so entstehenden Zusicherungen bilden dann einen Teil der Spezifikation, welche ein Service Level Agreement, basierend auf den Anforderungen des Kunden, entstehen lassen. Dieser Prozess ist nicht einfach, da sich viele Teile der Zusicherungen nicht quantifizieren oder messen lassen. Neben dem Inhalt der Zusicherung bezüglich eines Service sollte auch stets die Messmethode festgelegt werden.

In der Praxis reicht dies oft nicht aus, da Kunden zum Teil widersprüchliche Forderungen aufstellen. Das Service Level Agreement dokumentiert als eine Art Vertrag die Übereinkunft zwischen der Maintenanceorganisation und dem jeweiligen Kunden und sollte mindestens enthalten:

- Ausschlüsse für die Bereitstellung,²¹
- Rahmenbedingung und Beistellpflichten des Kunden.²²
- Reportingverfahren,
- Eskalationsverfahren,
- Spezifikation des zu liefernden Service,
- messbare Servicelevel, das heißt wie schnell, wie fehlerfrei und so weiter,

Besonders wichtig ist, dass die Servicelevel Agreements aus Consumersicht nachvollziehbar formuliert werden; Providerorganisationen tendieren oft dazu, ihre inneren Abläufe einem Kunden „überzustülpen“.

Der Lebenszyklus eines Service aus dem Blickwinkel des Providers betrachtet, lässt sich in mehrere chronologische Phasen einteilen:

- Servicedefinition,
- Eigenschaftenfestlegung,
- Serviceleistungserbringung,
- Serviceentkoppelung.

Die Definition des Service wird üblicherweise externalisiert und über andere Providerorganisationen oder Brokern diversen Consumern zur Verfügung gestellt. Nachdem ein potentieller Consumer Kontakt zum Provider aufgenommen hat, versorgt dieser den Consumer mit Informationen über die Serviceeigenschaften (funktional und nichtfunktional). Falls der Consumer einverstanden ist, gibt er den Auftrag zur Serviceleistungserbringung. Anschließend muss eine Entkoppelung vom Consumer vorgenommen werden, um die gebundenen Ressourcen anderen Services oder Consumern zur Verfügung stellen zu können.

2.3 Intelligente Organisation

Neben der Tatsache, dass ein SOE vollständig aus Services aufgebaut ist, muss eine solche Organisation auch in der Lage sein, sich auf Veränderungen in der Zu-

²¹ ...so zum Beispiel bei höherer Gewalt, Krieg, Unruhen, Bundestagswahlen und so weiter.

²² ...genau diese werden sehr häufig massiv verletzt.

kunft einzustellen, ein solches Verhaltensmuster wird als *intelligente Organisation* bezeichnet. Aus Sicht der Steuerung stellt das VSM (s. Anhang B) ein mögliches kybernetisches Modell einer solchen intelligenten Organisation dar. Für Organisationen gibt es keinen Stillstand, sie müssen sich permanent verändern. Eine solche Restrukturierung ist jedoch stets ein kosten- und nervenaufwendiger Prozess. Das sichtbarste Merkmal einer intelligenten Organisation ist es, dass sie diesen Prozess effizient und schnell gestalten kann (s. Abb. 2.7). Klassische Organisationen tendieren bei plötzlichen Veränderungen dazu, entweder zu langsam zu reagieren oder zu überreagieren (Aktionismus²³), intelligente Organisationen hingegen haben gelernt den Wandel zu gestalten, so dass er in einer optimalen Zeit effizient erreicht werden kann. Innerhalb der diversen intelligenten Organisationen kann es auch noch Unterschiede in der Adaptionsgeschwindigkeit geben, welche auf einen unterschiedlichen Reifegrad hindeuten. Organisationen zeigen, über die Zeit oder untereinander betrachtet, unterschiedlich stark intelligentes Verhalten, jedoch kann das intelligentere eine größere Menge von Veränderungen in kürzerer Zeit ertragen als das weniger intelligente.

Ausschlaggebend für intelligentes Verhalten sind neben der hohen Motivation aller beteiligten Menschen der Einsatz von Technologie, eine entsprechend gute Infrastruktur und Steuerungsmechanismen, welche auf Veränderungen adaptiert sind, beziehungsweise diese aktiv beeinflussen. Von Seiten der Ressourcen tendiert eine intelligente Organisation dazu, nie seinen Break-Even-Point bei der maximalen Kapazität zu erreichen, sondern stets genug Reserven für Notfälle zu besitzen. Eine geeignete Kombination aus SOE und VSM-Steuerung sollte es ermöglichen eine idealtypische intelligente Organisation zu bilden. Allerdings muss berücksichtigt werden, dass es für intelligente Organisationen auch Limitierungen gibt:

- **Größe** – Eines der Hauptprobleme innerhalb jeder Organisation ist das Change-management der beteiligten Mitarbeiter: Sobald eine gewisse Anzahl von Mitarbeitern überschritten wird, stellt Größe ein Problem dar. Damit eine soziale Gruppe im täglichen Arbeiten gut funktionieren kann, ist sie normalerweise auf einen sozialen und physischen Kontakt aller Beteiligten ausgelegt. Dies impliziert, dass Veränderungen nur in solchen kleinen Gruppen besonders schnell und umfassend ablaufen können. Die Gruppengröße liegt hier bei der typischen Sippengröße²⁴, circa 50–100, was der Größe eines kleinen mittelständischen Unternehmens entspricht.²⁵ Skalenökonomien zeigen bei Veränderungen meist eine sehr schlechte Wirkung, da diese auf große Hierarchien ausgerichtet sind.
- **Widerstand** – Neben den Mitarbeitern kann eine weitere Quelle des Widerstands gegenüber Veränderungen in der Architektur der eingesetzten Systeme beobachtet werden. Besonders sogenannte Legacysysteme in der Software oder stark institutionalisierte Prozesse und Policies können massive Widerstände gegen strukturelle Veränderungen produzieren.

²³ Aktionismus ist eine typische Reaktion für ein überfordertes Management. Viele Führungskräfte können den Zustand des Kontrollverlusts und der Passivität meist nicht mit ihrem eigenen Selbstbild vereinbaren.

²⁴ Hier lässt sich das archaische Erbe des modernen Menschen erkennen.

²⁵ Ein Grund dafür, dass kleine Unternehmen ideal für digitale Businessökosysteme sind, s. Kap. 5.

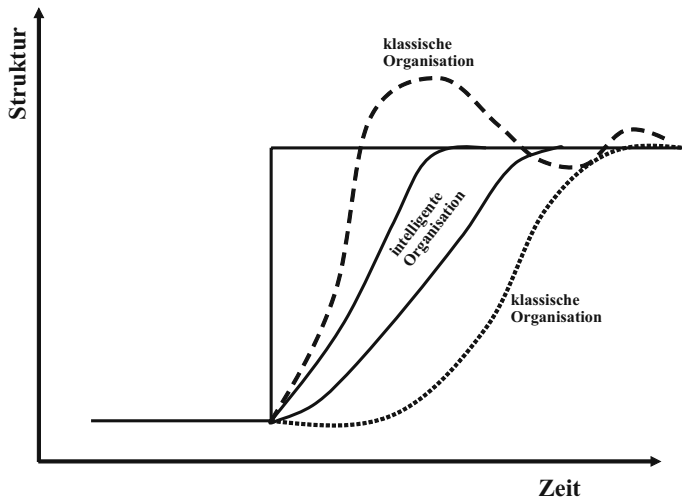


Abb. 2.7 Veränderungen in Organisationen

Damit eine Organisation überhaupt intelligent agieren kann, muss sie neben der Veränderung, welche durch die Marktbeobachtung ausgelöst wird, eine permanente Alignmentverbesserung in allen Dimensionen²⁶ betreiben. Es gibt noch ein zweites Charakteristikum für intelligente Organisationen: Wenn eine Organisation durch ihre Umgebung „gestört“ wird, dann reagiert sie mit Veränderungen auf diese Störung; ist jedoch diese Störung nicht mehr vorhanden, so fallen die meisten Organisationen wieder in den alten Zustand zurück.²⁷ Eine intelligente Organisation jedoch erinnert sich an die Störung und kehrt nicht mehr in den Ursprungszustand zurück, sondern hat „gelernt“, in Zukunft mit diesem Typus von Störung zu leben.

Eines der Schlüsselemente für jede Form von Kollaboration ist die gemeinsame Konstruktion von relevanter Bedeutung, welche von allen Betroffenen zur Erreichung eines gemeinsamen Ziels geteilt wird. Diese Form der Übereinstimmung wird auch als kognitives Alignment bezeichnet. Heutige IT-Technologie ermöglicht es, Kommunikation quasi instantan und mit völlig neuen Formen der gemeinsamen Bedeutungsfindung zu verwirklichen.²⁸ Im Rahmen der elektronischen Kollaboration gibt es drei Ebenen der Zusammenarbeit:

- **Kollektive Arbeit** – In diesem Fall arbeitet jeder für sich selbst und das Ergebnis ist die Summe der individuellen Anstrengungen. Typische Beispiele sind hier: Netzwerklaufwerke, Tabellenkalkulation und so weiter.
- **Koordinierte Arbeit** – Bei der koordinierten Arbeit wird die Tätigkeit der Individuen über Zuständigkeit und Übergaben gesteuert. Das Ergebnis ist eine Funk-

²⁶ Siehe Masak, *IT-Alignment*, Springer 2006.

²⁷ Dieses Verhaltensmuster macht das Changemanagement so besonders schwer.

²⁸ Einfache E-Mail-Systeme, aber auch Wikis und Yahoo- oder Google-Groups sind hierfür Beispiele.

tion der individuellen Anstrengung und der Qualität der Koordination. Typische Werkzeuge sind hier: E-Mail, Notes-Datenbanken aber auch Workflowsysteme.

- **Konzertierte Arbeit** – Hier müssen die Anstrengungen der Teilnehmer synchron geschehen und jede Veränderung eines Individuums beeinflusst die anderen Individuen. Es existieren nur Gemeinschaftsergebnisse. Typische Werkzeuge sind hier: elektronisches Brainstorming, Mindmapping und so weiter.

Ein Maß für den Grad an Intelligenz einer Organisation ist die Intelligenzbandbreite, welche sich als Produkt aus der Kollaboration und der Informationsassimilierung ergibt. Bei der Betrachtung der Bandbreite gehen dann die Gesamtmenge an Informationen in der Organisation und wie diese von den Individuen assimiliert werden kann ein. Neben dieser Assimilationsfähigkeit spielt die „Reife“ der Kollaboration eine Rolle; je reifer die Kollaboration, desto höher die Bandbreite. Eine solche Bandbreite ermöglicht es, den Wertschöpfungsprozess auf Grund von Koordination und Innovation mehr oder minder gut zu quantifizieren.

Eine der größten Herausforderungen für Organisationen ist es, ein hohes Maß an Agilität zu erreichen. Unter Agilität versteht man die Fähigkeit, Chancen zur Innovation oder Optimierung zu entdecken und diese durch schnelle Veränderung der Organisation oder Prozesse zu nutzen, um somit einen Wettbewerbsvorteil zu erreichen.²⁹ Zur gesamtheitlichen Agilität benötigt eine Organisation drei Fähigkeiten (s. Tabelle 2.3):

- **Kundenagilität** – Diese spiegelt die enge Zusammenarbeit mit aktuellen oder zukünftigen Kunden wider. Dabei spielen Kunden eine dreifache Rolle:

Tabelle 2.3 Agilitätstypen

Typ	Beschreibung	IT-Rolle	Beispiel
Kundenagilität	Fähigkeit, zusammen mit den Kunden neue Produkte oder Services zu entwickeln.	Aufbau virtueller Kundengruppen, Groupware, Produkte zum Testen und Produktfeedback	<i>Ebay</i> -Kunden sind Teil des de facto <i>Ebay</i> -Entwicklungsteams über die Vorschläge, Fehlermeldungen und den Work-Arounds, die diese erfinden.
Partneragilität	Fähigkeit, Wissen und Ressourcen von Partnern als Netzwerk einsetzen zu können.	Kollaborationstechnologien	<i>Yahoo!</i>
Organisationsagilität	Fähigkeit, die Prozesse der Organisation rasch abzuändern.	Modularisierung, Workflow, Prozesskoordination	<i>Ingram Micro</i> erlaubt es seinen Kunden und Lieferanten direkt auf das ERP-System zuzugreifen.

²⁹ Monopolisten sind nie agil.

- als eine Quelle innovativer Ideen³⁰,
 - als Co-Designer oder -erzeuger bei der Entwicklung und dem Design neuer Produkte oder Services,
 - als Consumer im Sinne eines Testers beim Einsatz neuer Produkte oder Services.
- Partneragilität – Bezeichnet die Fähigkeit, das Wissen, die Ressourcen und die Fertigkeiten anderer Organisationen durch Allianzen, Joint Ventures oder gemeinsame Projekte zu nutzen, um daraus neue Produkte oder Services zu erzeugen. Im Ressourcenteil zählt man dazu auch die Fähigkeit, geeignetes Outsourcing zu betreiben.

Tabelle 2.4 Die unterschiedlichen Umgebungsstimuli für den Geschäftsprozess

Stimulus	Beschreibung	Möglichkeiten	Verantwortung	benötigte Flexibilität
Konstant	Keine Variation des Stimulus, keine Kontingenz nötig, eine Standardreaktion kann definiert werden	Ein fester Ausführungspfad	Ausführende Personen für die definierte Reaktion verantwortlich.	Keine Flexibilität nötig
Unsicher, jedoch definiert	Endliche Menge an Stimuli ist vorab bekannt und Reaktion auf diese prädefiniert. Eintrittswahrscheinlichkeiten sind bekannt.	Multiple Ausführungspfade, aber alle sind vorab definiert. Pfade sind exklusiv und in der Gesamtheit überdeckend.	Ausführende Personen für die definierte Reaktion verantwortlich.	Flexibilität beruht auf dem Reaktionsrepertoire des Prozesses.
Mehrdeutig	Die Variationen der Stimuli bilden einen Satz von simultanen Stimuli.	Wenn die Mehrdeutigkeit der Stimuli beseitigt ist, reduziert sich der Fall auf die vorherigen Typen.	Prozessmanager ist verantwortlich, die Stimuli zu interpretieren und zu klassifizieren. Entscheidung beim Prozessmanager.	Die Flexibilität wird auf der Ebene des Managements benötigt, da die Optionen und Handlungsmuster identifiziert werden müssen.
Überraschung	Keine Reaktion ist vorab definierbar.	Vollständig neue Reaktion ist notwendig. Neuer Pfad mit neuen Personen und neuen Ressourcen ist nötig.	Die Reaktion und die Entscheidung liegt in den Händen der Akteure, da diese direkt die Auswirkungen beobachten können.	Die Reaktion muss lokal und dezentral stattfinden.

³⁰ Das SABRE-System wurde praktisch gesehen aus kontinuierlichen Kundenideen heraus geboren.

- Organisationsagilität – Hierunter wird die Fähigkeit verstanden, die eigenen Prozesse schnell anpassen zu können.

Der Einsatz von IT ist in diesen Fällen fast immer eine notwendige, aber keine hinreichende Voraussetzung, wobei für die eigentliche Agilität die IT sehr viel stärker die Rolle des Infrastrukturproviders spielt, die wirklichen Impulse müssen aus der Organisation und der Fachdomäne kommen.³¹

Eine wichtige Frage, die sich vor allen Dingen kleine und mittlere Organisationen stellen, ist: Was habe ich davon, wenn ich Teil eines Service Oriented Enterprise werde? Wird der daraus erzielte Gewinn die zusätzlichen Mehraufwände, den partiellen Verlust der Kontrolle und das veränderte Risiko übersteigen? Kollaboration benötigt immer eine Menge an Vorbereitung und Ressourcen. Damit sich ein SOE formen kann, müssen die Beteiligten schon im Vorfeld auf die Kollaboration eingestellt sein. Folglich ist einer der Erfolgsfaktoren für eine solche Kollaboration eine Reihe von Incentives, welche die einzelne Suborganisation auch langfristig befriedigen. Heute lässt sich noch wenig darüber sagen, ob eine kollaborative Organisation deutlich besser ist als ein „traditionelles“ Unternehmen. Einer der Gründe hierfür ist, dass es noch keine Maßzahlen für kollaborative Organisationen gibt. Unsere Metriken sind alle auf „traditionelle“ Organisationen ausgerichtet.

Aus der Perspektive des Changemanagements hat die intelligente Organisation nicht nur die Fähigkeit zur Adaption, sondern auch die Fähigkeit des „Entlernens“. Damit eine Organisation seine Strukturen verändern kann, muss sie in der Lage sein, bestehende Normen, Regeln, Werte und Grundsätze zu missachten, bevor diese durch neue ersetzt werden können. Je größer die Distanz zwischen den neuen Dingen und den alten, desto schwieriger ist der Entlernungsprozess für die Organisation.³²

2.4 Flexibilitätsdesign

Von Systemen und Organisationen wird in den letzten Jahren immer mehr Flexibilität verlangt. Doch wie kann man diese Flexibilität fundiert erreichen? Grundlage zur bewussten Steuerung von Flexibilität ist das „Law of Requisite Variety“ (s. S. 258), welches besagt, dass ein System eine hinreichende Vielfältigkeit³³ besitzt, wenn sein Repertoire an Reaktionen (Flexibilität) mindestens so mächtig ist, wie die Anzahl der Stimuli³⁴, der es in seiner Umgebung begegnen kann. Ein System ohne hinreichende Vielfältigkeit wird versagen, wenn es einem Stimulus begegnet, auf den es nicht ausgerichtet ist und ist damit nicht „lebensfähig“ (s. Anhang. B). Ein solches Versagen lässt sich in der Praxis bei Geschäftsprozessen be-

³¹ Die meisten Veränderungen, die durch IT-Abteilungen ausgelöst werden, haben langfristig betrachtet nur eine geringe Wirkung.

³² Autodidakten erleben dies als Individuen, wenn sie sich einen Trainer zulegen, dieser muss zunächst bestimmte Verhaltensweisen „abtrainieren“.

³³ Requisite Variety, s. Gl. C.3.

³⁴ Veränderungen.

obachten, welche nur eine beschränkte Menge an Antworten auf Variationen in den Anforderungen an den Prozess haben.

Tabelle 2.5 Benötigte Flexibilität in den Prozessveränderungen

Stimulus	Instanz	Typ
Konstant	–	–
Unsicher	✓	–
Mehrdeutig	✓	–
Überraschung	✓	✓

Man kann zwischen zwei Typen von Geschäftsprozessflexibilität unterscheiden:

- Prädefinierte Flexibilität – Hier wurde die Flexibilität schon zum Designzeitpunkt antizipiert, quasi als eine Variante in den Prozess eingebaut.
- Just-in-Time Flexibilität – Die spontane Reaktion auf Veränderungen verlangt einen intelligenten Prozessmanager, welcher die unerwarteten Variationen versteht und den Prozess darauf angemessen reagieren lässt.

Der Unterschied zwischen diesen beiden Typen von Flexibilität lässt sich auf die Variationen der Umgebung und die möglichen Geschwindigkeiten der Stimuli zurückführen. Folglich braucht man für den Entwurf eines flexiblen Geschäftsprozesses ein tiefes Verständnis der Variationen und Störungen der Geschäftswelt, welche die Stimuli bilden, die eine flexible Antwort des Geschäftsprozesses benötigen. In beiden Fällen wird sehr viel Wissen über den Geschäftsprozess auf der einen Seite und die möglichen (in aller Regel wahrscheinlichen) Veränderungen der Umgebung benötigt (s. Tabelle 2.4).

2.5 Service Oriented Architecture

Die ersten Softwaresysteme waren monolithische³⁵ siloartige Systeme. Sie entstanden aus dem Bedürfnis heraus, genau ein vorliegendes Problem zu lösen ohne Rücksicht auf die Architektur des Gesamtsystems. Kennzeichnend für die auf die Monolithen folgende Layerarchitektur ist der Aufbau der einzelnen Applikationen in sogenannte Layer (oder auch Schichten).³⁶ Die einfachste Form der Layerarchitektur ist die Client-Server-Architektur, welche eine primäre Einteilung in zwei Rollen vorsieht. Weitere Verfeinerungen dieser Form führen zu einer echten Layerarchitektur. Die Layerarchitektur wirkt wie eine „natürliche“ Architektur, da die Einteilung der Layer nach Funktionen und Abstraktionsgraden der intuitiven Zerlegung der Problemdomäne zu folgen scheint.

³⁵ Aus dem Griechischen *μονολιθος* wörtlich übersetzt „aus einem Stein“.

³⁶ Ein Layer stellt im Sprachgebrauch dieses Buchs eine vertikal logische Unterteilung, eine Schicht, eine tatsächliche vertikal physische Trennung der Software dar, das heißt ein Layer kann durchaus mehrere Schichten überdecken und eine Schicht kann aus vielen Layern bestehen.

Die Client-Server-Architektur lässt sich zu einer n-Tier oder auch Layerarchitektur verallgemeinern, die meist genutzte Unterteilung der verschiedenen Layer ist (s. Abb. 2.8):

- Benutzerinterface – Hier findet die gesamte Benutzerinteraktion statt, wobei der Begriff Benutzerinterface generisch zu sehen ist. Auch ein Drucker oder ein Archivsystem bildet, logisch gesehen, ein Benutzerinterface.
- Präsentationslayer – Der Präsentationslayer stellt alle notwendigen Informationen zu Verfügung, die an das Benutzerinterface geschickt werden.
- Prozesslayer – Dieser Layer ist optional in Architekturen vorhanden, hier wird der Prozessfluss gesteuert. Wichtig ist dieser Layer für Workflow und Kollaborationssysteme.
- Applikationslayer – Dieser Layer stellt die eigentliche Geschäftslogik dar.
- Datenzugriffslayer – Dieser stellt die Daten für die Objekte zur Verfügung.

Alle komponentenbasierten Architekturen haben neben einer vertikalen Schichtung auch eine horizontale Aufteilung. Eine solche Architektur hat sehr viel Potenzial für Wiederverwendung, Qualität und Robustheit. Die Komponenten selbst werden zur führenden Designrichtlinie bei der Entwicklung von Softwaresystemen. Die zentrale Idee hinter der komponentenbasierten Architektur ist die Zusammenstellung von Softwaresystemen mit Hilfe von diskreten, meist disjunkten, hochgradig wiederverwendbaren Komponenten. Diese Komponenten sind Einheiten, die Daten und Funktionen kapseln, ihre Interfaces ermöglichen eine Zusammenstellung zu einem kompletten Softwaresystem (Applikation). Obwohl in der Theorie die Wiederverwendung von Komponenten eine der zentralen Aufgaben der Komponenten ist, spielt diese Fähigkeit in der Praxis nur eine untergeordnete Rolle. Die meisten Softwareentwicklungen nutzen die Zerlegung in Komponenten primär als Strukturierungstechnik.

Innerhalb einer Service Oriented Architecture³⁷ (SOA) stellen alle Softwaresysteme und Applikationen Services³⁸ dar, welche in einer definierten Umgebung erreichbar sind. Die tatsächlich vorhandene Infrastruktur wird im Rahmen einer SOA so stark abstrahiert, dass nur noch die Funktionalität der Services, nicht jedoch ihre Implementierung oder die notwendigen Transportmechanismen in den Vordergrund treten. Diese Zerlegung ist unabhängig von eventuell vorhandenen Layern einer Softwarearchitektur, indem zusätzliche abstrakte Servicelayer eingeführt werden. Aus Sicht der Softwareentwicklung ist eine SOA eine Weiterführung der Idee der Komponentenbauweise. SOA ist kein wirklich neues Konzept, sondern eine Spezialform einer verteilten Systemarchitektur. Insofern enthalten „klassische“ verteilte Architekturen wie CORBA, DCOM oder J2EE Anklänge von SOA. Eine SOA kann daher wie eine Fortsetzung der bisherigen Softwarearchitekturen verstanden werden. Es existiert jedoch ein fundamentaler Unterschied zwischen einer SOA und

³⁷ Die erste Erwähnung von SOA geschah in einer Veröffentlichung der Gartner Group im Jahre 1996 mit dem Titel: *Service-Oriented Architecture Scenario*. Interessanterweise war das Szenario überhaupt nicht technisch, sondern hatte das Ziel, die Geschäftsprozesse in den Vordergrund zu stellen.

³⁸ Siehe Servicedefinition S. 15.

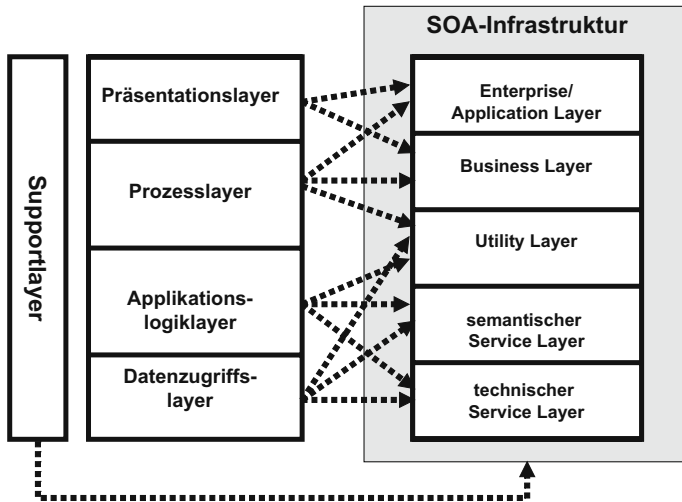


Abb. 2.8 Die fünf Layer einer SOA im Vergleich zu einer klassischen Layerarchitektur

anderen Architekturen: Traditionelle Architekturen werden in aller Regel universell – ohne Berücksichtigung der Organisation oder Struktur des zu unterstützenden Geschäftsprozesses – eingesetzt. Eine SOA ist jedoch nur dann sinnvoll, wenn alle Prozesse dem Serviceparadigma genügen und die Organisation sich an einem SOE ausrichtet. Außerdem repräsentiert eine SOA ein sehr viel größeres Maß an Abstraktion als die meisten bisherigen Architekturen (s. Abb. 2.9). Eine aus Services bestehende Architektur unterscheidet sich von objektorientierten und komponentenbasierten Architekturen. Services nehmen eine Art Zwischenstellung zwischen den Objekten und den Komponenten an, obwohl sie im Grunde frei skalieren können. Objektbasierte Architekturen (CORBA und DCOM) tendieren dazu sehr kleinteilig zu sein, da jedes Objekt einzeln zugänglich ist. Komponentebasierte Systeme hingegen sind meistens grobgranularer, da hier eine einzelne Komponente eine mehr oder minder geschlossene Einheit darstellt. Services liegen zwischen diesen beiden Extremen und können sich in einem Grenzfall wie Objekte und im anderen wie Komponenten darstellen.

Die hier verwendete Definition einer SOA ist analog zu der in diesem Buch verwendeten Servicedefinition (s. S. 15):

Eine SOA ist das Modell eines Systems, welches vollständig aus autonomen Services aufgebaut ist, deren Interaktion über dasselbe öffentliche Protokoll abläuft und im Modell stets die drei Rollen Provider, Consumer und Broker³⁹ vorhanden sind.

Diese SOA-Definition ermöglicht es, beliebige Protokolle und Rollen innerhalb eines Systems zu haben, allerdings wird man in der Praxis von einigen Forderungen

³⁹ Ein Broker ist ein „Informationshändler“, der dem Consumer Zugang zu den Providern oder den Services gewährt, s. Abschn. 2.6.2.

abweichen. So wird im Allgemeinen die Autonomie nicht vollständig durchgehalten und der Broker wird nicht in allen Implementierungen vorhanden sein. Die obige SOA-Definition ist so generisch gehalten, dass nicht nur „klassische“ Applikationen, sondern auch viele Formen der verteilten Soft- oder Hardware mit ihr beschrieben werden können.

Die Einführung von Layern stellt ein wichtiges Mittel zur Abstraktion in Softwaresystemen dar. Ein Layer enthält Services, die einen ähnlichen Abstraktionsgrad haben. Normalerweise bauen die einzelnen Layer aufeinander auf. Ein Layer implementiert die für den jeweiligen Abstraktionsgrad spezifischen Teilaspekte und nutzt dazu nur Services des direkt darunterliegenden Layers. Die Einteilung in Layer besitzt eine Reihe von Vorteilen. Sind die einzelnen Layer klar voneinander abgegrenzt und besitzen wohl definierte Interfaces, können diese eventuell wieder verwendet werden.

Wie viele Layer optimal sind⁴⁰, hängt stark von der jeweiligen Applikation ab. Wenn zu wenig Layer vorhanden sind, sinkt die mögliche Wiederverwendbarkeit, zu viele Layer führen zu Performancenachteilen, da eine Anfrage durch jeden Layer transportiert werden muss. Dieser Overhead kann bei einer Verwendung vieler Layer sehr viel größer sein als der Aufwand zur Ausführung der eigentlichen Operation. Dies gilt insbesondere dann, wenn die unterschiedlichen Layer auf verschiedenen Rechnern verteilt sind. Das Kernproblem von Layern ist, dass ein Layer nur in seinem lokalen Kontext optimieren kann, wobei für ein globales Optimum jedoch der gesamte Kontext notwendig wäre.

Bei den meisten Softwaresystemen hat sich eine 4-Layerarchitektur herausgebildet, die man mit den Layern einer SOA vergleichen kann. Eine SOA lässt sich analog in fünf Layer einteilen (s. Abb. 2.8):

- Enterprise- oder Application Layer – Dies ist der oberste Layer in einer SOA. Auf ihm sind die eigentlichen Endpunkte zur Kommunikation mit dem gesamten System angesiedelt. Ein Benutzer interagiert mittels eines sogenannten Application Frontends mit dem System. Der organisationsweite Zugriff auf dieses System erfolgt über die öffentlichen Enterpriseservices.
- Business Layer – Dieser enthält die prozesszentrierten Services.
- Utility Layer – Dieser Layer enthält alle Zwischenservices, das heißt: Façaden, Adapter, Gateways und funktionserweiternde Services.
- Basislayer – Der Basislayer ist der wichtigste Layer in einer SOA und enthält alle Basisservices. Er realisiert die Datenhaltung und Geschäftslogik. Zusätzlich enthält dieser Layer Proxies für Services, die in einem anderen System auf Ebene des Enterpriselayers angesiedelt sind. Der Basislayer zerfällt seinerseits in zwei Sublayer: Semantischer Service Layer und technischer Service Layer.

Ein wesentlicher Unterschied besteht in der Verteilung der unterschiedlichen Layer. Bei traditionellen Architekturen entspricht der logische Layer meist der physikalischen Unterteilung, welche sich später so auch in einer Laufzeitumgebung wiederfindet. So kann der Präsentationslayer auf einem Webserver, der Steuerungslayer

⁴⁰ Eine Applikation, die aus genau einem Layer besteht, ist eine monolithische Applikation (s. Abb. 2.9).

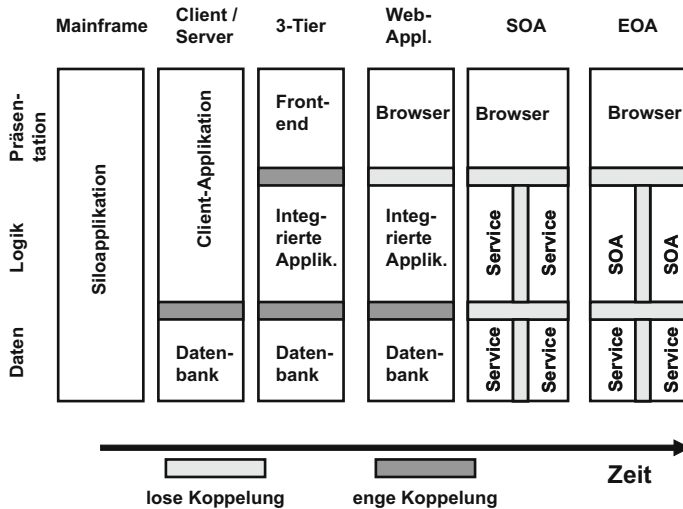


Abb. 2.9 Evolution der Architektur der Informationssysteme

auf einer Workflowengine sowie Applikations- und Datenlayer auf einem Applikationsserver ablaufen. Bei einer SOA bestimmt die Zugehörigkeit zu einem Layer nicht automatisch dessen physikalische Verteilung. Bei einer SOA sind Software- und Systemarchitektur voneinander entkoppelt. An dieser Stelle soll darauf hingewiesen werden, dass eine SOA nicht in Konkurrenz zur 4-Layerarchitektur, welche nach wie vor ihre Berechtigung hat, steht. So können etwa einzelne Services in dieser Form aufgebaut und realisiert sein. Der Vorteil einer SOA zeigt sich erst, wenn Services von mehreren Applikationen wiederverwendet werden. Klassische Layerarchitekturen partitionieren die verschiedenen Layer horizontal, wohingegen eine SOA zusätzlich eine vertikale Partitionierung vorsieht, welche die SOA aus der Komponentenarchitektur entlehnt.

Für den Einsatz einer Service Oriented Architecture in einer Organisation werden immer wieder folgende Vorteile behauptet:

- Implementierung großer Enterprisesysteme – Mittels SOA und den entsprechenden Managementwerkzeugen können auch große Systeme einfach organisiert werden. Services können leicht ausgetauscht werden, neue Services können leicht eingebunden werden und es bereitet keine allzu große Schwierigkeit, interne Services extern zugänglich⁴¹ zu machen. Im Gegensatz dazu sind mit enger gekoppelten Architekturen, wie bei objektorientierten Applikationen, derartige Änderungen am System mit einem weitaus höheren Programmier- und Konfigurationsaufwand verbunden.
- Skalierbarkeit – Eine SOA trifft nur wenige Annahmen über das Netzwerk und minimiert die Abhängigkeit von anderen Services und sollte daher im Netzwerk skalieren können.

⁴¹ Allerdings sollte beachtet werden, dass das Problem der Sicherheit vollständig gelöst sein muss.

Tabelle 2.6 Vergleich zwischen unterschiedlichen Architekturformen

	Komponenten	CORBA	SOA
Interface	Keine getrennte Interfacedefinition	IDL	Servicevertrag
Partitionierung der Applikationslogik	Horizontal	Horizontal & vertikal	Vertikal
Discovery	–	Repositorylookup der Objekte	Registrylookup der Services
Autonomie	–	Hinreichend	Hinreichend
Zusammensetzbarkeit	–	Gut	Gut
Koppelung	Sehr eng	Eng	Lose
Zustandsbehaftet	Ja	Abhängig vom Design	Sollte zustandslos sein
Granularität	Objekte/Funktionen	Objekte	Beliebig
Verteilungsreichweite	Applikation	Enterprise	Enterpriseübergreifend
Offenheit	Externe Dokumentation, Design und Policies	Ausdrucksfähigkeit des Interfaces, externe Dokumentation	Ausdrucksfähigkeit des Service, SLAs, externe Dokumentation
Interaktionsmodus	Synchron, blockend	Primär synchron, asynchron selten	Synchron und asynchron
Datenlebenszyklus	–	Objektlebenszyklus	–

- Hohe Wiederverwendbarkeit und lose Koppelung – Durch die lose Koppelung in der SOA wird die Integration neuer Services erleichtert. Services können auch ohne großen Aufwand in einem anderen Kontext verwendet werden. Code, der bereits für andere Applikationen entwickelt wurde, kann als Service veröffentlicht werden. Durch Wiederverwendung von Code und Services kann das Wachstum der Applikation verlangsamt werden und der Entwicklungsaufwand sinkt.
- Abstraktion – Durch die Modellierung von Applikationen in offenen Umgebungen kann eine Abstraktion der Funktionen geschaffen werden.
- Flexibilität – Das einfache Austauschen und Zusammensetzen der Services ermöglicht es, einfacher und rascher Änderungen am System durchzuführen als bei traditionellen oder objektorientierten Systemen.
- Standardisierung und Unterstützung durch viele Lieferanten – Viele Organisationen beschäftigen sich mit dem Entwickeln von Standards für SOA. Implementierungen der Standards sind von den meisten großen Softwareanbietern wie *IBM*, *Microsoft*, *Sun* und so weiter zu finden. Eine breite Unterstützung durch Werkzeuge erleichtert das Erstellen und Veröffentlichen von Services.

Eine SOA ist keine Fortführung des Webgedankens, technisch gesehen hat das Web nur eine einzige Repräsentation (HTML), im Gegensatz dazu kommen in den Organisationen diverse zum Teil überlappende⁴² Repräsentationen vor. Das Beste, was in diesem Fall erreicht werden kann ist, sich der Standardisierung zu nähern und überlappende Interfaces zu erlauben, mit der Maßgabe, dass eine Transformation

⁴² Obwohl alle Beteiligten fest der Ansicht sind, eine einzige Repräsentation im Sinne einer Standardisierung sei unbedingt notwendig.

zwischen diesen Interfaces bekannt ist. Damit eine SOA sehr stark ereignisorientiert sein kann, muss den Anforderungen an Flexibilität und Anpassbarkeit auch Genüge getan werden – die Ereignisorientierung muss sich auch in der Architektur widerspiegeln.⁴³

Am strukturell einfachsten ist eine Dreiteilung in die Bereiche Event Layer, Access Control Layer und Business Process Layer. Dies stellt eine logische und keine physische Einteilung dar, sie ist durchaus komplementär zu den Layerarchitekturen. Jeder der Layer (Event und Access Control sind parallel) konzentriert sich auf die speziellen Anforderungen und sollte dann auch durch spezielle Infrastrukturwerkzeuge unterstützt werden:

- Event Layer – Von dem Zeitpunkt an, wo Services eine reine P2P⁴⁴-Struktur in ihren Interaktionen verlassen⁴⁵, ist eine verteilte Steuerung für die Serviceaufrufe notwendig. Ein möglicher Mechanismus, dies zu erreichen, ist die Nutzung von Events. Ein eventgetriebener Ansatz besitzt lose Koppelung sowie Mechanismen, um auf unstrukturierte oder unzusammenhängende Interaktionen reagieren zu können. Der Event Layer in Form eines Eventservice stellt die Infrastruktur zur Erzeugung, Verfolgung und Übertragung von Events dar. Jeder Service kann Events produzieren und konsumieren. Zum einen sind diese Events Vorfälle, die sich irgendwann einmal ereignen und voneinander unabhängig erzeugt werden, zum anderen dienen sie auch zur Auslösung einer spezifischen Fähigkeit, welche durch einen Service ausgeführt werden kann. Die Events können typisiert werden und selbst Informationen enthalten. Innerhalb der Eventarchitektur werden die Events durch den Aufruf des Event Layers produziert, umgekehrt registrieren sich Prozesse und Services des Business Process Layers im Event Layer, um über das Auftauchen bestimmter Typen von Events informiert zu werden. Üblicherweise werden die Events in die Kategorien Business Events und System Events eingeteilt: Obwohl beide Kategorien sich syntaktisch sehr ähneln, unterscheiden sie sich semantisch sehr stark. Die System Events dienen dazu, die SOA als Infrastruktur zu steuern, während die Business Events zur Durchführung der fachlichen Prozesse innerhalb der Organisation dienen.
- Access Control Layer – Eine der Grundlagen des Serviceorientierungsparadigmas ist die Annahme, dass Geschäftsprozesse durch den Aufruf diverser Services realisiert werden können, welche eventuell auch verteilt und heterogen sind. Da Services auf Grund ihrer fachlichen Spezifikationen ermittelt und aufgerufen werden, ist der Provider nicht unbedingt ex ante bekannt, folglich kann ihm auch nicht a priori vertraut werden.⁴⁶ Das Vertrauen im Sinne der Prognosefähigkeit bezüglich des zukünftigen Verhaltens des Providers ist aber eine der notwendigen Vorbedingungen, um überhaupt eine Geschäftsbeziehung einzugehen. Zur Reduktion des Risikos, einen nichtvertrauenswürdigen Provider zu nutzen, müssen

⁴³ Architektur ist die strukturelle Kraft, welche bestimmte Fähigkeiten besonders gut unterstützt.

⁴⁴ Peer-to(2)-Peer.

⁴⁵ Dies ist eine oft zu beobachtende Anfangsform des Einsatzes von Services. Services werden als Peers direkt („hardcoded“) miteinander verknüpft.

⁴⁶ Auch bekannten Providern kann nicht unbedingt vertraut werden. Bekanntheitsgrad und Dauer der Beziehung beeinflussen die Vertrauenswürdigkeit oft nur subjektiv.

die Messages als auch die Zugriffe entsprechend verwaltet werden; mit dieser Aufgabe ist der Access Control Layer betraut.

- **Business Process Layer** – Der Business Process Layer befasst sich mit der Steuerung und Optimierung des Alignments zwischen Geschäftsprozessen und Services, sowie der Koordination von Aufrufen von Services.

2.5.1 Services

Die Services sind, analog zu den Komponenten, unabhängige Bausteine, welche gemeinsam eine Applikationsumgebung repräsentieren. Im Unterschied zu Komponenten haben Services besondere Charakteristika, die es ihnen gestatten, als Teil einer SOA zu agieren. Eine dieser Eigenschaften ist, dass ein Service zunächst völlig unabhängig von jedem anderen Service konzipiert sein sollte. Ein Service ist nur verantwortlich für seinen eigenen Ausschnitt der Fachlichkeit⁴⁷, welcher dann auf eine bestimmte Funktion oder Menge von Funktionen abgebildet wird. Diese Funktionen können von diesem Service auch anderen Services zur Verfügung gestellt werden. Eine SOA ist im Wesentlichen eine Sammlung von Services. Diese interagieren miteinander, indem sie Daten austauschen oder gemeinsame Aktivitäten durchführen; zur Koordination gemeinsamer Aktivitäten wird ein Mechanismus benötigt, mit dem Services zusammengefügt werden können.

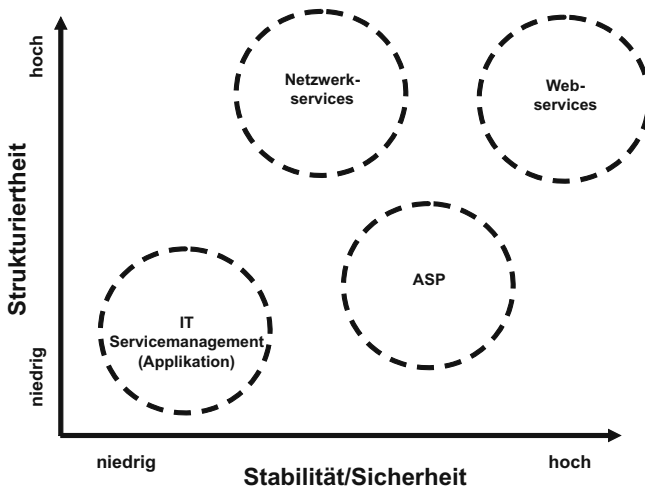


Abb. 2.10 Verschiedene Formen von Services

⁴⁷ Diese Ausschnitte der Fachlichkeit müssen nicht disjunkt sein, oft ist ein gewisser Grad an Redundanz notwendig.

Der entscheidende Vorteil einer SOA liegt darin, dass eine SOA die Implementierung eines Service von dessen Interface trennt. Das heißt, es wird das „Was“ vom „Wie“ getrennt. Der Consumer braucht nicht zu wissen, wie der Service funktioniert und implementiert ist, für ihn ist nur interessant, dass er ihm zur Verfügung steht und dass er ihm die Serviceleistung liefert, die er benötigt. Die einzige Anforderung, die der Consumer an den Service hat, ist, dass der Service eine Antwort zurückliefert und zwar in einem vereinbarten Format. Die Eigenschaften eines Service führen dazu, dass man den Service als eine isolierte Teilfunktion einer Organisationsfunktion ansehen kann; die Organisationsfunktion hingegen kann als eine lockere Verbindung von Services im Rahmen einer SOA gesehen werden. Durch die Unabhängigkeit, welche die Services in dieser SOA haben und die gekapselte Programmlogik, die sie mitbringen, sind die Services nicht an irgendeine Plattform oder Technologie gebunden.

Alle Services innerhalb einer SOA haben eine Reihe von technischen Eigenschaften, welche die SOA als Framework aber auch als Designmodell repräsentieren. Zu diesen technischen Eigenschaften der Services zählen:

- lose Koppelung – Wechselwirkende Services sind bedingt durch ihre Natur immer lose gekoppelt. Dies ist notwendig, da sie a priori plattform- und implementierungsneutral sein müssen, was dazu führt, dass sie nur ihr Interface, nicht jedoch ihre Implementierung bekannt geben. Eine der Ideen von Services ist es, den Provider wechseln zu können. Hierfür ist eine lose Koppelung notwendig, da eine enge Koppelung den Wechsel des Providers stark erschwert.
- messagebasierte Interaktion – Die Kommunikation zwischen einzelnen Services geschieht immer messagebasiert. Üblicherweise werden diese Messages asynchron ausgetauscht, beziehungsweise der Requestor simuliert eine Synchronität, wenn dies durch den Consumer⁴⁸ gewünscht ist. Messagebasierte Kommunikation vereinfacht es auch, eine lose Koppelung zu unterstützen.
- dynamische Discovery – Services sind stets Softwareteile, die anderen Consumern zur Verfügung stehen. Da sich a priori nicht festlegen lässt, dass sich der gewünschte Service auf demselben Rechner wie der Consumer befindet, ist es wichtig zu wissen, welche Services zurzeit zur Verfügung stehen und was diese tatsächlich leisten können. Daher muss ein Modell zum dynamischen Auffinden und Entdecken von Services unterstützt werden. Besonders wichtig ist eine solche Fähigkeit offensichtlich bei mobilen Systemen. In einer solchen Umgebung ist es überhaupt nicht möglich zu wissen, welche Services einem in Zukunft zur Verfügung stehen.
- selbstständiges Deployment – In sehr großen Umgebungen ist ein automatisches Deployment des Service ein großer Vorteil.⁴⁹
- Portabilität – Die Services sollten von der konkreten Implementierung als auch der konkreten Umgebung so unabhängig wie möglich sein.

⁴⁸ Oft wird nicht von den Consumern, sondern von den implementierenden Programmierern die Synchronität gefordert, da synchrone Programme sich viel einfacher bauen lassen als asynchrone.

⁴⁹ Computerviren und -würmer haben ähnliche Eigenschaften und können sich erfahrungsgemäß selbst fast überall und rasch installieren. Leider!

- Implementierungsneutralität – Services definieren sich ausschließlich über ihre Qualitäten und ihre eigenen Interfaces, nicht jedoch über ihre jeweilige Implementierung. Ohne diese Prämisse wäre ein Provider- oder Servicewechsel nur schwer möglich.
- Autonomie – Services müssen sich entkoppeln und werden daher auch unabhängig von anderen Services implementiert. Ausschließlich die Interfaces eventuell nutzbarer Services sind bekannt.
- policybasiertes Verhalten – Bestimmte technische Eigenschaften wie Transaktionsicherheit, Verschlüsselung und Kontext sollten keine festen Bestandteile der Serviceimplementierung sein. Diese Forderung ist zwar per se nicht zwingend, erleichtert aber die Nutzung unterschiedlicher Qualitäten (s. Abschn. 2.5.4) für eine gegebene Implementierung.
- Konfigurierbarkeit – Services werden nur abstrakt definiert und später implementiert. Die Konfiguration eines Service sollte dynamisch möglich sein, um auf Dauer möglichen Veränderungen folgen zu können.
- late Binding – Die Services werden dynamisch aufgerufen, mit der Folge, dass das Binding der Services erst zur Laufzeit geschehen kann. Eine Technik, die heute in diversen Betriebssystemen eingesetzt wird, so unter *Windows* durch die DLLs.

2.5.2 Servicemodell

Eine SOA basiert auf den Wechselwirkungen zwischen drei verschiedenen Beteiligten, dem Provider, auch Server genannt, dem Servicerequestor, auch Client oder Consumer genannt, und der Registry (s. Abb. 2.11) oder Broker. Der Provider stellt die einzelnen Implementierungen zur Verfügung und publiziert ihre Eigenschaften in der Registry. Der Consumer wiederum findet seine gesuchten Services mit Hilfe einer Interfacebeschreibungssprache innerhalb der Registry und nutzt die dortigen Interfacedefinitionen, um sich gegen den Provider zu binden. Die konkrete Nutzung der Services läuft dann transparent über das Netzwerk, mit Hilfe einer Protokollsprache⁵⁰ zwischen dem Consumer und dem Provider.

Diese Art des Auffindens der Services trennt den Consumer vom Provider und führt zu einer Entkoppelung. Da nun der Provider nicht mehr ab initio bekannt ist, können auch keine Annahmen über die Serviceimplementierung außer dem Interface getroffen werden, es entsteht mehr oder minder automatisch eine lose Koppelung zwischen Consumer und Provider. Durch diesen Mechanismus ist der Consumer gezwungen, sich auf unterschiedliche Provider für den gleichen Service einzustellen, dasselbe gilt umgekehrt auch für den Provider.

Das Modell einer SOA wird am einfachsten durch das „find-bind-execute“-Paradigma⁵¹ beschrieben, wobei der jeweilige Consumer sich auf der Suche nach

⁵⁰ Meistens SOAP oder CORBA. CORBA ist streng genommen kein Protokoll, diese Tatsache ist aber hier irrelevant.

⁵¹ Oft auch SOA-Paradigma genannt.

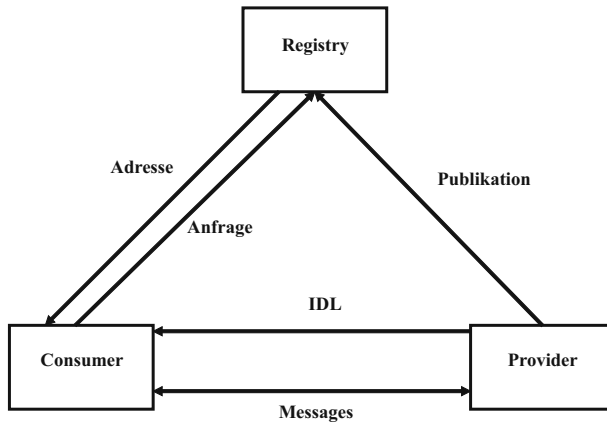


Abb. 2.11 Die drei Rollen: Provider, Consumer und Registry

einem bestimmten Service an die Registry wendet. Ist der Registry der Service bekannt, so antwortet er dem Consumer mit einem Servicevertrag (Service Contract) und einer Adresse, unter dem der Consumer beim Provider den Service in Anspruch nehmen kann. Hat der Provider den Vertrag und die Serviceadresse erhalten, so kann er zu dieser Adresse verbinden und den Service anfordern. Der Provider erlaubt dem Consumer, den Service bezüglich des Vertrages zu nutzen. Jeder Provider kann seine Services bei der Registry registrieren lassen, damit sie von den Consumern gefunden werden können.

- **Consumer** – Der Consumer selbst kann ein Service, eine Applikation, ein Programm, ein Softwaremodul, ein Hardwaremodul oder auch ein Mensch sein, der diesen Service in Anspruch nimmt. Der Consumer sucht den Service bei der Registry und erhält die Serviceadresse und den Vertrag. Damit bindet der Consumer den Service über ein Transportprotokoll und führt danach den Service aus, indem er eine entsprechend des Vertrags formatierte Anfrage an den Provider schickt.
- **Provider** – Oft ist der Provider gemeint, wenn man vom Service spricht. Der Provider ist die Entität, welche den Service anbietet und über das Netzwerk zugänglich macht. Er akzeptiert die Anfragen des Consumers und führt sie aus. Der Provider veröffentlicht seinen Vertrag bei der Registry, damit die Consumer den Service finden und nutzen können.
- **Registry** – Eine Registry basiert auf einem Softwaresystem, welches die Adressen von den verfügbaren Services enthält. Die Registry akzeptiert und speichert die Verträge der Provider und stellt sie dem Consumer zur Verfügung, falls dieser den entsprechenden Service nutzen möchte.
- **Servicevertrag** – Der Servicevertrag ist eine Spezifikation, die festlegt, wie ein Consumer in Kontakt mit dem Provider des Service zu treten hat. Der Vertrag legt das Format der Anfrage und der Antwort des Service fest. Die Vorbedingungen und Nachbedingungen sind ebenfalls dem Vertrag zu entnehmen. Sie spezifizieren den Zustand, in dem sich der Service befinden muss, um eine be-

stimmte Funktion ausführen zu können. Der Servicevertrag beschreibt außerdem die verschiedenen „Quality of Service Levels“, die unterstützt werden.

- **Serviceproxy** – Der Provider stellt dem Consumer ein Serviceproxy zur Verfügung. Dieses Proxy befindet sich zwischen Consumer und Provider und zwar lokal auf dem Client des Consumers. Der Consumer kann eine Anfrage ausführen, indem er die API-Funktion des Proxys verwendet. Der Vorteil besteht darin, dass das Proxy direkt bei der Registry den Provider und den zugehörigen Vertrag erhält und damit die formgerechte Serviceanfrage im Auftrag des Consumers ausführt. Das Proxy erhöht die Leistung des Service, da es die Referenzen der Provider und die zugehörigen Verträge lokal speichert. Wird also der gleiche Service öfters ausgeführt, so müssen nicht jedes Mal bei der Registry die entsprechenden Daten angefordert werden, sondern der Service kann direkt angefragt werden. Dadurch reduziert sich die Anzahl an Netzwerkverbindungen, die der Consumer tätigen muss. Alle Servicemethoden, die keine Daten vom Provider benötigen, können lokal auf dem Proxy ausgeführt werden, dadurch wird der Provider entlastet. Benötigt ein Service nur einen kleinen Datenteil, so kann dieser ebenfalls beim Provider heruntergeladen und im Proxy gespeichert werden. Für den Consumer ist es nicht von Belang, ob die Funktionen lokal im Proxy oder remote beim Provider ausgeführt werden. Das Proxy kann nur Funktionen zur Verfügung stellen, die der Provider auch unterstützt.
- **Servicelease** – Die Registry bestimmt, mit dem sogenannten Servicelease, wie lange ein Servicevertrag gültig ist. Der Consumer stellt an die Registry eine Anfrage und bekommt dann neben dem eigentlichen Vertrag noch einen Mietvertrag, der beschreibt, wie lange das Servicelease gültig ist. Der Vertrag ist gültig ab dem Zeitpunkt, an dem der Consumer das Servicelease erhält, bis zu dem Zeitpunkt, den die Registry im Servicelease festgelegt hat. Wenn das Servicelease abgelaufen ist, muss der Consumer eine neue Anfrage an die Registry stellen, um ein neues Servicelease zu erhalten. Das Servicelease ist unabdingbar für Services, die Zustandsinformationen über die Bindung zwischen Consumer und Provider benötigen. Das Servicelease definiert die Zeit, in welcher der Zustand unverändert bleibt. Außerdem wird damit die Koppelung zwischen Consumer und Provider reduziert, indem die Zeit begrenzt wird, welche den Consumer an den Provider bindet. Ohne die Idee eines Vertrags auf Zeit könnte ein Consumer für immer an einen speziellen Service oder Provider gebunden werden und nie wieder von dem Vertrag zurücktreten. Das würde dazu führen, dass zwischen Consumer und Provider eine viel engere Beziehung bestehen würde als eigentlich beabsichtigt. Das würde dem Kerngedanken einer SOA, der losen Koppelung von Services, widersprechen. Möchte der Provider seine Implementierung und damit auch den Service ändern, so kann er das, indem er wartet, bis der Mietvertrag ausläuft und der Consumer dann einen Vertrag mit den neuen Konventionen erhält. Damit kann die Implementierung geändert werden, ohne die Ausführung des Services zu beeinträchtigen, da der Consumer eine neue Anfrage für einen neuen Service und Mietvertrag stellen muss. Wenn die neuen Verträge in Kraft treten wird allerdings nicht garantiert, dass der Service identisch zum Vorherigen ist.

2.5.3 *Komposition*

Obwohl es sehr oft von Interesse ist einzelne Services anzusprechen, liegt ein Teil der Mächtigkeit des Serviceorientierungsparadigmas in der Fähigkeit begründet, bestehende Services zu neuen Services zusammenzufassen, einen Prozess, den man Servicekomposition oder kurz Komposition nennt. Eine solche Flexibilität ist unbedingt nötig, damit sehr rasch auf Veränderungen der Umgebung reagiert werden kann.⁵² Die Frage der Komposition von Services kann aus zwei Blickwinkeln betrachtet werden:

- Geschäftsprozesskomposition – Diese Abstraktionsebene komponiert völlig neue Geschäftsprozesse aus bestehenden Teilprozessen oder Services. Eine solche Komposition ist für den Endbenutzer transparent und kundenorientiert, da sie hochgradig interaktiv ist und eine intensive Beteiligung von Domänenexperten benötigt. Eine solche Kompositionsform ist ohne eine explizite Betrachtung oder zumindest eine partielle Veränderung der Organisation nicht möglich.
- Servicelevelkomposition – Der Aufbau von Services aus anderen Services ist eine Technik, welche in ähnlicher Form schon bei Workflowsystemen eingesetzt wird. Im Gegensatz zur Geschäftsprozesskomposition steht hier die Interoperabilität und technische Machbarkeit im Vordergrund. Fast immer wird diese Kompositionsform ohne jegliche Berücksichtigung organisatorischer Abläufe vorgenommen.

Die Mechanismen zur Komposition von Services zu neuen (in der Regel komplexeren) Services haben zwei unterschiedliche Zielrichtungen:

- komplexe Services,
- semantische Services.

Das Interesse hinter dem Kompositionsprozess für Services wird hauptsächlich durch zwei Problemgebiete getrieben, zum einen der Kommunikation zwischen Organisationen und zum anderen durch die **Enterprise Application Integration (EAI)**. In dieser Richtung wurde eine Anzahl von XML-basierten Spezifikationen entwickelt. Eine andere Entwicklung findet im Bereich der semantischen Services, hier besonders bei den semantischen Webservices, statt. Bei einer statischen Komposition stellt sich die Frage: Wie kann ein Service genutzt werden, um eine komplexe Aufgabenstellung zu lösen? Ein solches Problem wird sich in Zukunft immer häufiger stellen, denn je mehr Services (meist feingranularer Natur) schon vorhanden sind, desto größer ist die Chance der Wiederverwendung in einem komponierten Service. Es gilt, diese vorhandenen Services zu nutzen und sie zu neuen Services zusammenzubauen. Kompositionsregeln behandeln das Vorgehen, um aus einfacheren Services komplexere aufzubauen. Hierfür gibt es zwei Möglichkeiten:

- Orchestrierung – Bei der Orchestrierung wird ein neuer Service dadurch geschaffen, dass vorhandene Services durch einen zentralen Koordinator (Orchestrator)

⁵² In gewisser Weise setzt dies die Annahme von stabilen „atomaren“ Services voraus, denn wenn die Bestandteile sich schneller ändern als die Umgebung der Komposition, wird diese ad absurdum geführt.

gesteuert werden. Der Orchestrator nimmt die Aufrufe von außerhalb entgegen und verteilt die Aufgaben an die einzelnen benutzten Services. Im Bereich der Webservices sind für die Orchestrierung die Protokolle BPML und BPEL vorhanden.

- Choreographie – Bei der Choreographie gibt es keinen zentralen Koordinator, sondern es wird die Kommunikation zwischen den einzelnen Services und Aufgaben beschrieben. Insofern resultiert der Gesamtservice aus einer Reihe von P2P-Interaktionen zwischen den beteiligten (Sub)-Services. Für die Choreographie von Webservices wurde bisher das Protokoll WS-CDL vorgeschlagen.

Eine besondere Berücksichtigung bei der Komposition sollte die Schaffung von Layern haben, da eine layerübergreifende Komposition Teile der Architektur zunichte macht. Typischerweise setzt sich ein Kompositorservice aus Services des gleichen oder des direkt darunterliegenden Layers zusammen.

Bei der Orchestrierung existiert ein zentraler Prozess, welcher die Kontrolle über alle beteiligten Services hat. Dieser Prozess, der seinerseits ebenfalls ein Service sein kann, koordiniert die Ausführung verschiedener Operationen der Services. Die involvierten Services wissen nicht, dass sie komponiert werden und Teil eines darüberliegenden Geschäftsprozesses sind und dürfen es auch nicht wissen. Nur der zentrale Koordinator der Orchestrierung hat die entsprechende Information. Die Choreographie dagegen besitzt keinen zentralen Koordinator. Vielmehr weiß jeder involvierte Service, wann seine Operationen auszuführen sind und mit wem er zu interagieren hat. Choreographie ist als eine Kollaboration zu verstehen, deren Ziel der Austausch von Messages in einem Prozess ist. Alle Beteiligten müssen dazu den Prozess, die auszuführenden Operationen und das Timing für den Messageaustausch kennen. Oder anders formuliert: Die Orchestrierung beschreibt die ausführbaren Aspekte eines Service aus Sicht des Orchestrators, während die Choreographie die Zusammenarbeit mehrerer Services aus Sicht des Prozesses darstellt. Durch die Kapselung wurde der Prozess in einzelne Funktionalitäten aufgespalten. Die eigentliche Prozessstruktur und die Prozesslogik sind dabei verloren gegangen, da die einzelnen Kapseln durch lose gekoppelte Services realisiert werden können, die in keinerlei Bezug zueinander stehen müssen. Mit Hilfe der Orchestrierung der funktionalen Kapseln soll der Ausgangsprozess wieder „zusammengesetzt“ werden, ohne dass der Aspekt der losen Koppelung, welcher eine zentrale Rolle in einer SOA darstellt, verloren geht.

Auch wenn in der funktionalen Kapselung Ereignisse wie Funktionen jeweils einem Service zugeordnet werden, gehören sie nicht zwingend zu diesem Service selbst, denn Ereignisse stellen vielmehr die Verbindungsglieder zwischen den gekapselten Funktionalitäten dar, wobei für die Orchestrierung nur diejenigen Ereignisse betrachtet werden müssen, welche Funktionalitäten verschiedener Services miteinander verbinden. Ereignisse, welche innerhalb eines Services zwei oder mehrere Funktionen miteinander verbinden, beschreiben die innere Logik des funktionalen Services, welche durch die Implementierung des zu realisierenden Service umgesetzt werden muss.

Die Orchestrierung lässt sich aus einer prozessorientierten Sicht eines einzelnen Kooperationspartners auf zwei Ebenen betrachten:

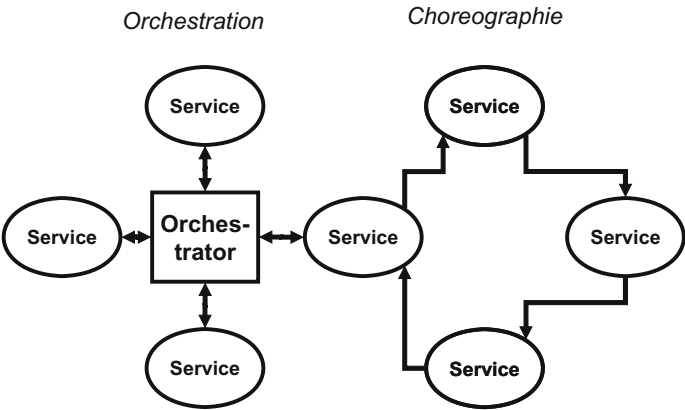


Abb. 2.12 Orchestrierung im Vergleich zur Choreographie

Tabelle 2.7 Vergleich der Choreographie- und Orchestrationssprachen. (✓) für indirekte Unterstützung

Feature	BP EL	BP ML	WS-CDL	WSCI	OWL-S
Kollaborationsmodellierung	✓	(✓)	✓	✓	✓
Ausführungsmodellierung	✓	✓	–	–	✓
Rollenkonzept	Schwach	–	✓	✓	–
Transaktion & Kompensation	(✓)	✓	(✓)	✓	(✓)
Exceptionhandling	✓	✓	✓	✓	✓
Semantik	–	–	–	–	✓
Quality of Service	–	–	–	–	–

- Interfaceebene – Die Interfaces aller anderen Kooperationspartner werden aus Sicht des ausgewählten spezifiziert.
- Ablaufebene – Die Interfacespezifikation wird um genaue Ablaufbeschreibungen ergänzt, welche notwendig sind, um die gewünschte Funktionalität zu ermöglichen, dabei werden nur die eigenen Abläufe, nicht jedoch die der Kooperationspartner beschrieben.

Im Unterschied dazu wird bei der Choreographie ein Vertrag über den Messageaustausch und dessen Ablauf festgelegt, wobei beschrieben wird, wann welche Message gesendet und verarbeitet werden muss. Außerdem sollte auch das Ausnahmeverhalten bei einer Choreographie stets festgelegt sein.

2.5.4 Quality of Service

Ein wichtiger, aber schwer zu definierender Aspekt ist die Qualität eines Service. Die Summe der Qualitäten eines Service, das heißt die Summe der nichtfunktionalen, aber zugesicherten und messbaren Eigenschaften, bezeichnet man als **Quality of**

Service (QoS). Die Feststellung, was tatsächlich benötigt wird, um eine vordefinierte Qualität eines Service überhaupt erreichen zu können, ist eine schwierige Aufgabe. In den meisten Fällen wird dabei die bestehende Erfahrung extrapoliert und als Quelle einer Schätzung genutzt. Innerhalb einer SOA wird dieses Problem noch dadurch forciert, dass es in einer solchen Umgebung möglich sein muss, dieselben Services mit unterschiedlichen Qualitätsstufen auf einer identischen Infrastruktur zu betreiben.⁵³ Neben den Problemen des Betriebs ergeben sich noch weitere Fragestellungen auf Grund der Architektur:

- Consumerperspektive – Typischerweise wollen die Consumer die Qualität des Service definieren. In traditionellen Systemen ist dies kein so großes Problem, da hier meistens das gesamte System unter der Hoheit des Consumers abläuft. Nicht jedoch in einem vollständig serviceorientierten System: In diesem kann nicht mehr davon ausgegangen werden, dass das gesamte System unter der Regie des Consumers abläuft, schließlich können die Services von beliebigen Providern geliefert werden. Die Kunden – sprich Consumer – wollen festlegen, was sie von dem Service erwarten, in ihrer eigenen Terminologie mit einer eigenen Bewertung von Qualitäten. Allerdings ist zu vermuten, dass in den meisten Fällen der Provider einfach einen „alten“ Service recycelt und damit gerade nicht die Consumerperspektive einnimmt.
- Kompositerservices – Eines der Ziele hinter einer SOA ist die Fähigkeit, neue Services aus dem Zusammenbau von bestehenden Services zu erzeugen. Obwohl die Komposition fachlich relativ einfach erscheint, stellt sich die Frage, wie die Qualitäten der Services bei einer solchen Zusammenlegung propagieren. Speziell, wenn ein definiertes stochastisches Verhalten gefordert wird, ist unklar, ob der neue Gesamtservice sich nach einfachen statistischen Gesetzen verhält.
- QoS-Degeneration – Selbst wenn es gelingt einfache Kompositerservices zu modellieren, auf Dauer werden die Beziehungen zwischen den Services von inhärenten Rekursionen geprägt sein. Solche Rekursionen führen dann sehr schnell zu sinnentleerten Aussagen über die jeweiligen Qualitäten. Ab einer gewissen Größe verhält sich der Service als Ganzes nur noch bedingt deterministisch; an dieser Stelle sind Qualitäten gefragt, welche sich aus systemtheoretischen Betrachtungen (s. Anhang A) ableiten lassen.

Qualitäten der Services können und müssen auch genutzt werden, um spezifische Services zu selektieren, neben mehr fachlich getriebenen Qualitäten, welche in aller Regel schon über das Interface und die Policies abgebildet werden, existieren serviceübergreifend eine Reihe quantifizierbarer Qualitäten, welche zur Auswahl dienen können:

- Ausführungspreis –
- Ausführungsdauer –
- Reputation – Die Reputation eines Services ist ein Maß dafür, welches Vertrauen ihm entgegen gebracht wird. Am einfachsten wird die Reputation über ein Ranking durch eine große Zahl von Consumern bestimmt.

⁵³ Schließlich möchte der Provider in der Lage sein, bei fachlich identischen Services ein Differenzierungsmerkmal für den Consumer zu liefern.

- Transaktionsfähigkeit –
- Kompensationsrate –
- Strafrate –

Zusätzliche Möglichkeiten, Services zu bewerten, sind:

- Performanz –
- Zuverlässigkeit – Die Zuverlässigkeit ist die Wahrscheinlichkeit, dass die Software keinen Fehler im System für eine spezifizierte Zeit und spezifizierte Randbedingungen produziert. Vorausgesetzt der Service erfüllt seine Aufgaben zum Zeitpunkt $t = 0$, dann ist die Zuverlässigkeit R gegeben durch:

$$R(t) = p_{\text{Failure}}(\tau > t) ,$$

wobei τ eine kontinuierliche Variable ist, welche die Zeit festlegt, bis der erste Fehler⁵⁴ auftritt.

- Verfügbarkeit – Die Verfügbarkeit ist die Fähigkeit eines Service, seine regulären Funktionen auszuführen. Die Verfügbarkeit steht in engem Zusammenhang mit der Zuverlässigkeit R . Wenn neue Instanzen des Service oder „reparierte“ Instanzen mit einer Rate von ρ auftauchen, dann ergibt sich die Verfügbarkeit A zu:

$$A(t) = R(t) + \int_0^t R(t - \tau) \rho(\tau) d\tau .$$

Oft ist die mittlere Verfügbarkeit $\bar{A}$ eine publizierte Größe:

$$\bar{A}(t) = \frac{1}{t} \int_0^t A(\tau) d\tau .$$

Eine permanente Verfügbarkeit ist dann durch $\lim_{t \rightarrow \infty} \bar{A}(t) \approx 1$ gegeben.

- Verlässlichkeit – Ist die Fähigkeit einen Service zu liefern, dem vertraut werden kann.

Welche konkreten Maße sich auf Dauer durchsetzen werden bleibt noch abzuwarten, aber die QoS werden immer wichtiger, je stabiler in Zukunft die fachlichen Interfaces sind.

2.5.5 Policy

Die Policies sind Richtlinien, welche von Menschen vorgegeben werden, um damit das Verhalten der Services zu beeinflussen. Services brauchen stets eine Policy, nach der sie operieren, in den meisten Fällen wird diese implizit durch die Umgebung gegeben, aber es gibt auch Anstrengungen, das Verhalten von außen steuerbar

⁵⁴ In diesem Kontext bedeutet Fehler neben fachlichen Fehlern auch das Nichtvorhandensein oder die Unerreichbarkeit des Service.

zu machen. Ein solches Verhalten muss dann in einer Sprache spezifiziert werden. Sprachen oder Standards, die explizit Policies unterstützen, sind:

- WS-Policy,
- WS-Security,
- XACML⁵⁵.

Das Aufkommen dieser Sprachen hat das Ziel, vorhandene Policyengines einzusetzen und damit den Selektions- und Ablaufprozess der Services deutlich zu verbessern.⁵⁶ Alle heutigen Policy Sprachen haben jedoch eine Reihe von Problemen:

- Die meisten Policy Sprachen werden von ganz unterschiedlichen, in sich heterogenen Organisationen vorangetrieben, mit der Folge, dass jede dieser Sprachen eine mehr oder minder eigenständige Semantik und Syntax besitzt und die unterschiedlichsten Konzepte implementiert.
- Es ist unklar, wo Policies aufhören und Applikationen oder Services beginnen. Typischerweise würde man die Sicherheit und Transaktionseigenschaften als Policies eines Service verstehen, aber Kosten oder andere „Quality of Service“-Spezifika sind viel schwieriger einzuordnen, da die heutigen Policyframeworks sich fast ausschließlich auf die technischen Eigenschaften der Services fokussieren.
- Häufig sind die Eigenschaften, die für Verhandlungen über die Nutzung von Services notwendig sind, viel feingranularer als es die Policy Sprachen ermöglichen.⁵⁷

Speziell im Rahmen der Entwicklung von Plattformen für Services werden Policies und die entsprechenden Sprachen immer wichtiger werden, da sie in der Lage sind, alle Services nach identischen Mustern (abhängig vom Consumer) zu steuern.

Aus systemtheoretischer Sicht sind Policies der Versuch, die Umgebung (Kontext) eines Service aktiv zu steuern, so dass sich dadurch das Verhalten des Service und damit nach der Servicedefinition (s. S. 15) der Service selbst verändert. Aus Sicht des Consumers agieren Policies wie Parametrisierungen des Interfaces, aus Sicht des Providers jedoch wie Eigenschaften der Ablaufplattform des Services. Architektonisch gesehen kann die Ausführung einer Policy als eine Art Metaservice verstanden werden und wird auch so in einigen Architekturen implementiert (s. Anhang B.1).

⁵⁵ eXtensible Access Control Markup Language (XACML) ist ein XML-Schema, welches die Darstellung und Verarbeitung von Autorisierungen zu standardisieren versucht.

⁵⁶ Eine Situation, wie sie sich in der Vergangenheit bei Protokollen wie LDAP (Lightweight Directory Access Protocol) zeigte, vermutlich können die LDAP-Erfahrungen auf den Sektor der Policies übertragen werden.

⁵⁷ Da die Feinsteuerung der Nutzung theoretisch jede beliebige Parameterkombination des Interface als auch des Servicevertrags betreffen kann, ist dies vermutlich ein nicht lösbares Problem.

2.5.6 Webservices

Webservices können als ein Produkt der Entwicklung des **World Wide Web (WWW)** gesehen werden, da neben der reinen Darstellung und Verlinkung von Dokumenten auch der Informationsaustausch recht schnell in den Vordergrund des WWW rückte. Die Nutzung von Webservices führt nicht automatisch zu einer SOA, ganz im Gegenteil, frühe Webserviceimplementierungen haben die Tendenz, sich zu fest codierten Aufrufen von Services zu entwickeln, die einfach ein Webserviceprotokoll einsetzen. Genauso gut könnte auch direkter Call gemacht werden. SOA ist eine Architekturidee und Webservices sind ein bestimmtes Protokoll!

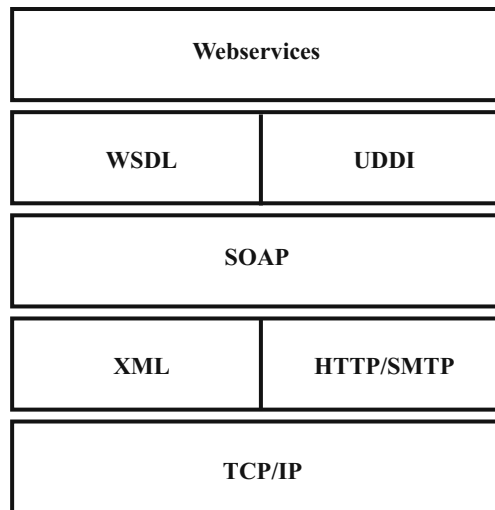


Abb. 2.13 Protokollstack für Webservices

Oft werden Webservices mit dem Konzept der mobilen Agenten verwechselt. Mobile Agenten werden üblicherweise eingesetzt, um auf einem anderen System Operationen durchzuführen. Sehr oft steckt dahinter die Idee des autonomen mobilen Agenten, welcher sich wie ein Wurm⁵⁸ oder Virus⁵⁹ völlig unabhängig in dem anderen System bewegt. Typisch für solche mobilen Agenten sind Aufgaben im Bereich der Informationsbeschaffung⁶⁰. Mobile Agenten können auch als Webservices implementiert sein, jedoch ist nicht jeder mobile Agent ein Webservice. Markante Unterschiede sind:

⁵⁸ Würmer wandern von einem System zum anderen und können sich dabei ähnlich den Viren oft auch reproduzieren.

⁵⁹ Ein Virus vermehrt sich innerhalb eines Systems und infiziert andere Programme, indem er sie verändert.

⁶⁰ Internetsearchengines oder Datamining.

- Ein Webservice hat nur Wissen über sich selbst, nicht jedoch Wissen über seine Consumer⁶¹ oder andere Webservices in seiner Umgebung. Im Gegensatz dazu sind mobile Agenten meistens damit beschäftigt, Informationen über ihre Umgebung und eventuellen Consumer zu sammeln. Diese Form der „Selfawareness“ ist zwar manchmal nur auf einer Metaebene vorhanden, trotzdem ist sie immanent.
- Agenten streben von sich aus Aktivitäten an – Webservices sind stets passiv.
- Agenten sind bis zu einem gewissen Grad autonom – Webservices sind dies heute nicht, obwohl die Autonomie eine zentrale Forderung an die Services ist.
- Agenten werden meist a priori kooperativ konzipiert – Webservicestandards berücksichtigen dies heute nicht.
- Agenten müssen sich auf diverse Umgebungen einstellen können – bei Webservices ist die Umgebung faktisch fixiert.

Im Vergleich zu den mehr allgemein orientierten Komponenten sind die Webservices ein spezieller Fall, denn hierunter versteht man lose gekoppelte, ausführbare Applikationen, welche dynamisch über ein TCP/IP-Protokoll eingebunden werden. Aus einer anderen Perspektive beurteilt, sind Webservices eine mögliche Implementierungsform von einer SOA. Die offizielle Definition von Webservices ist laut dem *World Wide Web Consortium*:

...software application identified by a URI, whose interfaces and binding are capable of being defined, described and discovered by XML artifacts and supports direct interactions with other software applications using XML based messages via Internet based protocols.

Die Nutzung von SOAP⁶² ist auch eines der Probleme der Webservices. Ursprünglich war SOAP entwickelt worden, um **Remote Procedure Calls (RPC)** zu ermöglichen. Beim RPC wird ein lokaler direkter Aufruf eines Moduls durch einen Aufruf eines Moduls in einem anderen Rechner ersetzt. Diese Herkunft führte dazu, dass die ersten Webservices SOAP als „Tunnel“ für applikationsspezifische verteilte Aufrufe genutzt haben und damit keinen Servicecharakter zeigten. Diese Situation setzt sich heute fort, da Applikationen, die auf SOAP aufbauen, nicht per se interoperabel sind. Webservices haben keine eigenen Benutzeroberflächen, mit der Folge, dass solche Benutzeroberflächen durch andere Mechanismen erst gebaut werden müssen. Daraus resultiert eine geringe Sichtbarkeit der Webservices für die Endbenutzer, da die Endbenutzer Software direkt nur durch die Benutzeroberflächen erleben.

Welche Voraussetzungen an Technik auf der Protokollebene sind für die Webservices notwendig? Obwohl Webservices auf Standardprotokollen aufbauen, brauchen sie eine gewisse Menge von Voraussetzungen. Diese Voraussetzungen bauen systematisch aufeinander auf. Der sogenannte Webservicestack erfüllt diese Voraussetzungen, er braucht, von unten nach oben betrachtet:

⁶¹ Im Sprachgebrauch der Webservices oft auch Requestor anstatt Consumer genannt.

⁶² Ursprünglich stand SOAP für **S**imple **O**bject **A**ccess **P**rotocol, aber mittlerweile wird dies für missverständlich gehalten, so dass SOAP keine weitere Bedeutung hat.

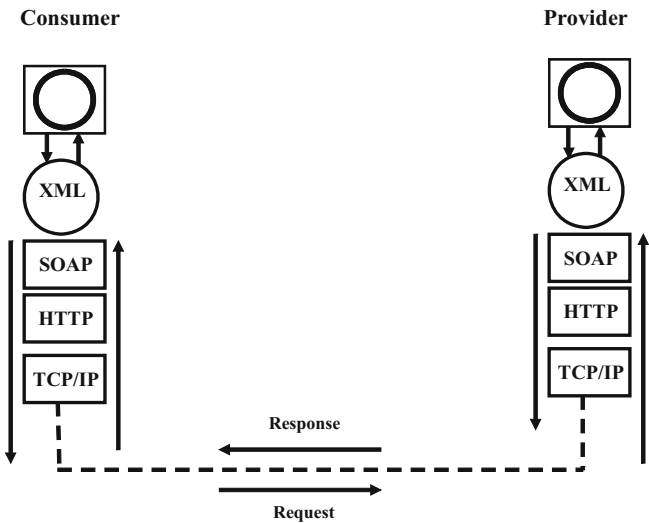


Abb. 2.14 Aufrufstack für Webservices mit XML und SOAP

Tabelle 2.8 Vergleich zwischen CORBA und Webservices

Eigenschaft	CORBA	Webservice
Datenmodell	Objektmodell	SOAP Messages
Koppelung	Eng	Eng/lose
Lokationstransparenz	Objektreferenz	URL
Typsystem	IDL	XML
Exceptionhandling	IDL Exception	SOAP-Fault Message
Serialisierung	ORB	–
Parameterübergabe	Referenz/Wert	Wert
Transfertyp	Binär	Unicode
Zustand	Zustandsbehaftet	Zustandslos
Laufzeitkomposition	DII	UDDI/WSDL
Registry	Interface- und Implementation-Repository	UDDI/ESB
Servicediscovery	CORBA Namensservice	UDDI
Sprachunterstützung	Jede Sprache mit IDL-Binding	Jede Sprache
Events	CORBA Eventservice	–

- **TCP/IP** – Diese logische Basisverbindung stellt das Rückgrat jeder Kommunikation im Webservice-Umfeld dar.
- **XML** – Die Protokollsprache XML dient zum Messageaustausch der einzelnen Webservice-Aufrufe.
- **HTTP** – Das HTTP nutzt das TCP/IP als darunterliegendes Transportprotokoll. Durch SOAP wird HTTP sowohl für den Aufruf als auch den Austausch der XML-Dateien, beziehungsweise XML-Datenströme genutzt.
- **SOAP**,

- UDDI⁶³,
- WSDL⁶⁴.

Aufbauend auf diesem Protokollstack werden die einzelnen Webservices implementiert. Interessanterweise sind die Teile TCP/IP, HTTP, XML und SOAP so weit verbreitet, dass sie in vielen Bereichen den heutigen De-facto-Standard darstellen. Im Fall von TCP/IP ist der Übergang zur Hardware in Form von embedded Chips auf den Netzwerkkarten heute schon fließend⁶⁵. Obwohl eine Reihe von verschiedenen Internetprotokollen existiert, hat sich XML als der De-facto-Standard für Webservices herauskristallisiert.

2.5.7 Präsentationsservices

Im Bereich der Service Oriented Architecture gibt es keine Services für die Benutzeroberfläche, da in einer SOA ein Service auf eine zustandslose Funktion reduziert wird. Aus ökonomischer Sicht betrachtet ist dies nicht besonders hilfreich, da schätzungsweise 50–70% der Softwareentwicklungskosten heute direkt oder indirekt Kosten der Benutzeroberfläche sind.⁶⁶ Der Bau dieser Benutzeroberflächen ist, bedingt durch die heute übliche Eventsteuerung und der großen Zahl an Kombinationsmöglichkeiten in Design, Implementierung und Test sehr teuer. Auf der anderen Seite ist die Benutzeroberfläche der Teil eines Softwaresystems, welcher für den Anwender direkt zugänglich ist und ihm damit einen Eindruck vom System vermittelt. Speziell die Akzeptanz eines neuen Softwaresystems steht und fällt mit der Benutzeroberfläche und der Erwartungshaltung der Anwender an diese Oberfläche. Dieses wichtige Kriterium wird von der Idee der SOA überhaupt nicht adressiert. Im Gegenteil, in der Literatur erscheinen Oberflächen meist als lästig und kompliziert oder es wird lapidar auf die Existenz von Browsern verwiesen.

Eine SOA benötigt eine generische Benutzeroberfläche, da sonst die Basisidee der Serviceorientierung verletzt wird. Insofern ist eine „reine“ SOA nur mit Brow-

⁶³ Universal Description Discovery and Integration.

⁶⁴ Web Services Description Language, s. Abschn. 2.7.1.

⁶⁵ Ein Phänomen, welches als Sedimentation bezeichnet wird. Die Sedimentation kann man ausnutzen, um effektiver zu produzieren. Bei der Sedimentation bewegt sich ein Teil der Supportfunktionalität von der Applikation in die Middleware, von der Middleware in das Betriebssystem und von dort in die Hardware. Beispiele hierfür sind Netzwerkverbindungsprotokolle, die heute Bestandteile aller modernen Betriebssysteme sind. Aber auch Supportfunktionalität, wie beispielsweise ein Printspooler oder ein Jobscheduler, welche früher Bestandteil einer Software waren, sind heute oft in der Infrastruktur wiederzufinden. Ein besonders schönes Beispiel für Sedimentation ist Lastverteilung in Webservern; diese wanderte zunächst in die Betriebssysteme und ist heute sogar in der Netzwerkhardware implementiert.

⁶⁶ Die besonders hohe Vielfaltigkeit (s. Anhang C.3.1) einer Benutzeroberfläche (nicht nur die möglichen Zustände der einzelnen Oberflächenelemente spielen eine Rolle, auch die Reihenfolge ihrer Aktivierung und Deaktivierung ist oft entscheidend), welche berücksichtigt werden muss, führt zu dieser großen Zahl.

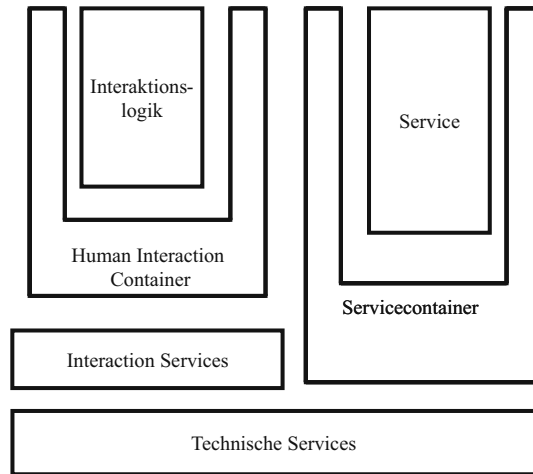


Abb. 2.15 Der Human Interaction Container in einer Gesamtarchitektur

sern oder Portlets⁶⁷ möglich. Eine andere Variante ist, die Benutzeroberfläche an dieselbe Ontologie (s. Abschn. 2.8.4) anzubinden wie den genutzten fachlichen Service, eine solche Strategie setzt keine heutigen Styleguides voraus, sondern standardisierte Repräsentationen fachlicher Objekte. In sehr engen Grenzen ist dies schon heute möglich, so zum Beispiel bei Adressen oder Kontendaten. Die funktionalen Anforderungen an einen Präsentationsservice sind:

- **Daten** – Daten müssen für eine Darstellung und Nutzung im Präsentationsservice in eigenem weit verbreitetem, offenem, standardisiertem und einfach verwendbarem Format vorliegen. Um eine gute Integration zu ermöglichen, sollten die Daten einer integrierten Ontologie folgen. Aufgabe der Präsentationsservices ist es, die Daten funktional und ansprechend nutzbar zu präsentieren. Dazu kann der einzelne Präsentationsservice präsentationsspezifische Daten transparent cachen⁶⁸, aber nicht explizit speichern wie die Basisservices; Präsentationsservices sind somit zustandsbehaftet.
- **Funktionen** – Die Komponenten zur Implementation der Funktionalität des Präsentationsservice sollten eine möglichst hohe Kohäsion und eine möglichst geringe Koppelung aufweisen. Je nach gewähltem Entwurfsmuster für die Implementation der Präsentationsservices gliedern sich die Funktionalitäten unterschiedlich. Eines der Ziele dabei ist es, eine möglichst gute Abstraktion der angebotenen Funktionalität bei weitgehendem Erhalt oder sogar einer Verbesserung der Performanz der implementierten Komponenten und der Gesamtfunktionalität des Präsentationslayers zu erreichen.

⁶⁷ Teile eines Portals.

⁶⁸ Das Caching verletzt die Forderung der Zustandslosigkeit, ist aber in der Praxis fast immer notwendig.

- Verhalten – Das Verhalten des Präsentationsservices muss korrekt, konsistent und jederzeit überprüfbar sein. Insbesondere müssen Transaktionen unterstützt werden. Das Verhalten sollte für Consumer des Systems so transparent und deterministisch wie möglich sein.
- Ausnahmen – Ausnahmen, welche für die Consumer relevant sind, müssen diesen in angemessener Form präsentiert werden. Alle Ausnahmen müssen protokolliert werden, um eine spätere Rückverfolgbarkeit bei Ausfällen der Funktionalität oder sonstigen Problemen zu ermöglichen. Bei Auftreten von nicht behebbaren, systemkritischen Fehlern und darauf folgendem Systemausfall sollte das System stets in einen gesicherten, deterministischen Systemzustand übergehen.

Anstelle eines Präsentationsservices lässt sich die Wechselwirkung eines Menschen mit Software auch anders modellieren. Ein Ansatz ist der **Human Interaction Container (HIC)**. Dieser Container, analog zum Servicecontainer (s. Abschn. 2.6.5), adaptiert sich dynamisch an eine geänderte Serviceumgebung und stellt dem Consumer stets die gleichen Funktionen zur Verfügung. Ein solcher Container führt zu einer völlig anderen Sicht auf ein IT-System (s. Abb. 2.15). Ein solcher HIC ist gleichwertig zu einem Servicecontainer und macht den Menschen zu einem Teil des IT-Systems, oder anders formuliert durch einen HIC wird der Mensch zu einem Subsystem des Systems und nicht zu einem Bestandteil der Systemumwelt (s. Anhang A). Die Interaction Services befassen sich mit dem Management der Consumeraufgaben und der Evaluierung des aktuellen Zustands von Dialogen und Interaktionsmustern.⁶⁹

2.6 Service Oriented Platform

Die Flexibilität, welche eine Serviceorientierung innerhalb der Organisation und der Software produziert, muss sich in der Infrastruktur widerspiegeln und von dieser explizit unterstützt werden. Die Schwierigkeit der Unterstützung liegt darin begründet, dass durch eine SOA ein hohes Maß an Interoperabilität erzeugt wird, die infrastrukturell erst ermöglicht werden muss, ein **Enterprise Service Bus (ESB)** ist ein Weg in diese Richtung. Die Heterogenität der Services als auch der ihrer Provider läuft dem jahrzehntealten Trend zur Homogenität in der Infrastruktur entgegen. Trotzdem lässt sich die Serviceorientierung auch auf die IT-Infrastruktur übertragen, denn dann kann die Infrastruktur wohldefinierte Services zur Verfügung stellen. Zwar ist es theoretisch möglich, Services ohne eine Serviceinfrastruktur durch direkte Aufrufe oder mittels eines Applikationsservers zu betreiben, aber diese Modelle haben keinerlei Vorteile gegenüber herkömmlichen Applikationen. Erst der Einsatz einer entsprechenden Serviceplattform ermöglicht es, die typischen Eigenschaften der Interoperabilität und Flexibilität, die von Services erwartet werden, wirklich zu nutzen.

⁶⁹ In gewisser Weise steht der HIC im Gegensatz zum „Affective Computing“ oder „Instinctive Computing“, bei dem explizit auf die emotionalen Bedürfnisse des einzelnen Menschen eingegangen wird und so eine erhöhte Produktivität erzielt werden kann.

Historisch betrachtet sind die meisten IT-Infrastrukturen stückweise, bottom-up und sehr technikzentriert entstanden, mit der Folge, dass sich die Infrastruktur und die Applikationen sehr eng koppeln. Diese enge Koppelung führt seit vielen Jahren zu einer Überbewertung der Technologie im Vergleich zu den eigentlichen Informationen, aber gerade diese Informationen sind der wirkliche Sinn und Zweck der IT. Dies ist eines der Risiken des pervasiven Computings (s. Kap. 4), speziell im häuslichen Umfeld existiert Infrastruktur im IT-Sinne meist nur rudimentär. Eine Folge der Serviceorientierung muss die Entkoppelung von Geschäftsprozessen und der darunterliegenden IT-Infrastruktur sein. Neben der Ausrichtung der IT-Infrastruktur auf mehr Services, Entkoppelung von Applikationen, muss auch das Infrastrukturmanagement in Form von Services durchgeführt werden. Dieser Teil wird als **Service Oriented Platform (SOP)** bezeichnet. Eine solche SOP muss eine Reihe von Aufgaben erfüllen. Dazu zählen unter anderem:

- **Sicherheit** – Isolierte Server können auch schon heute sichere Webservices zur Verfügung stellen; dies wird aber in Zukunft nicht ausreichen, da die Sicherheit über mehrere Services hinweg und simultan auf diversen Plattformen erreicht werden muss. Außerdem muss eine SOP in der Lage sein, sich an ein externes Identity Management System andocken zu können.
- **Routing** – Die Infrastruktur muss es ermöglichen, alle Messages zu interpretieren und an einen entsprechenden Provider weiterleiten zu können. Ohne diese Routingfunktionalität lässt sich ein dynamisches Kompositionsmodell (s. Abschn. 2.5.3) nur schwer realisieren.
- **Transformation** – Die Transformation von XML-Strukturen ist zwar nicht unbedingt eine der originären Aufgaben einer SOP, trotzdem muss sie eine solche Funktionalität besitzen, ansonsten ist eine SOP nicht in der Lage, Routing entsprechend zu unterstützen. Daher liegt es nahe, eine Reihe von Transformationservices innerhalb der SOP anzusiedeln.
- **Prozessmanagement** – Die Fähigkeit, Abläufe und Prozesse als ein Netz von Serviceaufrufen und Zustandsübergängen ablaufen zu lassen, ist eine der zentralen Fähigkeiten jeder SOP.
- **Monitoring und Administration** – Neben den Abläufen muss die SOP es ermöglichen, diese zu kontrollieren, messen und administrieren zu können.

Zusätzlich sollte eine SOP eine Reihe von Charakteristika aufweisen, um die Infrastrukturaufgaben einfacher bewerkstelligen zu können:

- inkrementelles Deployment,
- explizite Unterstützung öffentlicher Protokolle,
- dezentralisiert, dezentral verwaltet und ausbaubar.

Diese Aufzählung von Kennzeichen einer SOP zeigt, welche Mächtigkeit eine voll ausgebaute SOP besitzen muss. Das Phänomen der Sedimentation (s. Fußnote S. 55) setzt sich auch in einer SOP fort; bei einer SOP wandert immer mehr an Funktionalität, die heute in Applikationen angesiedelt ist, in die Infrastrukturschicht, außerdem übernimmt eine SOP in Zukunft eine Reihe von Funktionen, die heute ein Applikationsserver durchführt.

2.6.1 Komponenten

Eine SOP selbst besteht wiederum aus einer Reihe von Komponenten (s. Abb. 2.16), welche eine mehr oder minder klar umrissene Funktionalität besitzen:

- **Servicedirectory** – Das Servicedirectory liefert Informationen über die Services (Interfaces und Implementierungen), Provider und Rahmenbedingungen. Es besteht dabei aus zwei Blöcken:
 - **Registry** – Eine Registry gibt dem Consumer Angaben über die der Registry bekannten Services innerhalb der Reichweite der Infrastruktur. Da eine Registry und eine ESB nicht zwanghaft benutzt werden müssen, ist es auch denkbar, dass Services außerhalb der Organisation über Mechanismen angesteuert und genutzt werden, die nicht zur SOP gehören (zum Beispiel direkter Aufruf). Zu diesen Angaben zählen neben den syntaktischen Beschreibungen der Interfaces auch Informationen über die nichtfunktionalen Eigenschaften der Services (QoS, s. Abschn. 2.5.4), sowie Daten zur taxonomischen Klassifikation (s. Abschn. 2.8.4) oder ontologischen Positionierung (s. Abschn. 2.8.4) des entsprechenden Service. Die Registry enthält aber keine Serviceimplementierungen, sondern nur Verweise (meist in Form einer URI) auf die Implementierungen.
 - **Repository** – Ein Repository steuert den kompletten Lebenszyklus eines Service während seiner Entwicklung und Laufzeit und arbeitet mit der Registry zusammen. Das Repository verwaltet die Serviceinterfaces und Implementierungen (Konfigurations- als auch Versionsverwaltung). Das Repository enthält neben den Implementierungen auch eine Reihe von Metainformationen über die Services. Die Rolle, welche die Registry für den regulären Betrieb von Services hat, übernimmt das Repository für die Entwicklung von Services. Während der Entwicklung sind nicht nur Informationen über die Interfaces, sondern auch über die Art und Weise, wie die entsprechende Aufgabe gelöst worden ist und welcher Service von wem genutzt wird, im Repository zugänglich.

Registry und Repository können auch gemeinsam in einem Paket implementiert werden, in diesem Fall versteht man das Repository mit einem UDDI-Interface und verweist dabei auf sich selbst.

- **Werkzeuge** – Eine SOP benötigt Werkzeuge für ihren sinnvollen Einsatz. Eine Vielzahl von heute noch nicht standardisierten Werkzeugen sind hier denkbar, die in zwei Kategorien eingeteilt werden können:
 - **Administration** – Zurzeit sind noch keine SOP-dedizierten Administrationswerkzeuge entwickelt worden. Es existieren aber Werkzeuge, die einen Teil des notwendigen Spektrums abdecken, aber sie entstammen einer anderen Applikations- und Infrastrukturhistorie. Wichtige zu unterstützende administrative Aufgaben sind:
 - Steuerung des ESBs,
 - Verwaltung von Rollen und Rechten für die Servicenutzung,

- Verwaltung von Policies zur Servicenutzung,
 - Bepreisung⁷⁰,
 - Verfolgung von Fehlern und Ausnahmen,
 - Verfolgung des Lebenszyklus der Serviceimplementierungen,
 - Kontrolle der Performanz der SOP.
- Entwicklung – Werkzeuge in dieser Kategorie umfassen faktisch alle typischen Softwareentwicklerwerkzeuge:
 - Generatoren für die Interfaces,
 - Modellierungswerkzeuge für Abläufe und Servicekomposition,
 - High-Level-Debugger⁷¹,
 - Simulatoren für Abläufe und Servicekomposition,
 - Testwerkzeuge,
 - Analysewerkzeuge für Abhängigkeiten.
- Enterprise Service Bus – Der ESB bildet den eigentlichen Kern einer SOP (s. Abschn. 2.6.3).

2.6.2 Broker Architekturen

Heutige Ansätze für eine SOP nutzen meist Brokerarchitekturen, ein Broker ist eine Art softwaretechnischer Makler, der zwischen zwei oder mehreren Systemkomponenten vermittelt. In einer SOA übernimmt ein solcher Broker diverse Rollen, unter anderem:

- Übertragung von Informationen zwischen den verschiedenen Services,
- Anlegen und Zurverfügungstellung einer Registry,
- Abgleich zwischen Aufruf eines Service und dem implementierten Interface,
- Unterstützung der Verhandlung zwischen Consumer und Provider.

Da der Broker eine zentrale Rolle spielt, ist es notwendig, sein grundsätzliches Design näher zu beleuchten (s. Tabelle 2.9). Brokerarchitekturen sind nicht a priori festgelegt, außer sie werden als Produkt eingekauft, aber selbst dann können sie meist noch konfiguriert werden, von daher ist es sinnvoll, die verschiedenen Designalternativen für Brokerarchitekturen zu betrachten:

- zentral versus dezentral – In der Weise, wie die Größe und die Ansprüche an ein serviceorientiertes System wachsen, wird die Skalierbarkeit eine immer wichtigere Rolle in einer Organisation spielen. Hier sind dezentrale Architekturen vorteilhafter als zentrale, zumindest kann die Last in bestimmten Bereichen auf mehrere Broker verteilt werden. Im Gegensatz dazu bildet in einer zentralen

⁷⁰ Einige neue Services im Mobilfunkumfeld scheiterten in der Vergangenheit daran, dass sie nicht nachvollziehbar zu bepreisen waren.

⁷¹ Ein High-Level-Debugger ermöglicht es, die Aufrufe der Services und die ausgetauschten Daten zu verfolgen und gegebenenfalls die einzelne Serviceimplementierung zu debuggen.

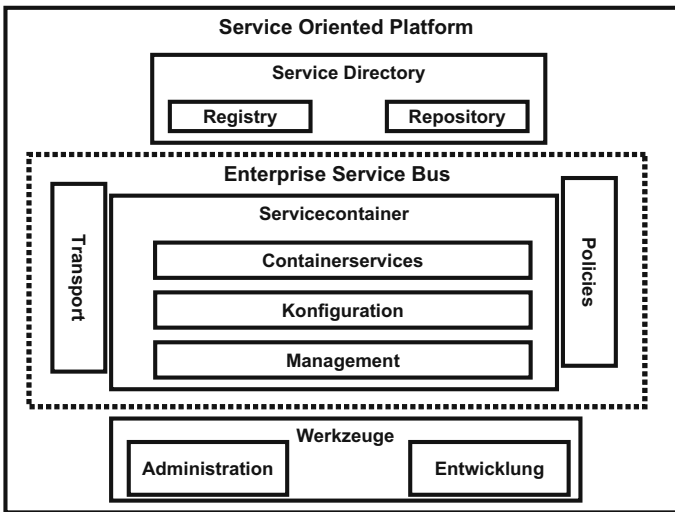


Abb. 2.16 Die Komponenten einer Service Oriented Platform

Tabelle 2.9 Unterschiedliche Brokerdesigns

Design	Alternativen	Kommentar
Verteilung	Zentral	Ein einzelner Broker, der alle Entscheidungen trifft.
	Dezentral	Mehrere Broker mit lokalen Entscheidungen.
Aufruf	In-band	Requests und Response werden übertragen.
	Out-of-band	Nur Requests existieren.
Brokertyp	Informationbroker	Gibt Hinweise über die Serviceorte. Fokus auf Registry.
	Servicevirtualizer	Gibt Services zusätzliche Funktionalität, zum Beispiel Transaktionen.
Fokus	Brokerfokus	Unabhängigkeit und Stabilität des Brokers. Services sind sekundär.
	Servicefokus	Services werden primär. Risiko der Verquickung.

Umgebung ein einzelner Broker das Rückgrat und eventuell auch das Bottleneck für die Skalierbarkeit. Außerdem können Probleme an einer zentralen Stelle das Gesamtsystem lahm legen. Ein dezentraler Broker hat dagegen den Vorteil, dass dieser nur seine lokalen Services kennt und auch nur von einer Teilmenge von Services genutzt werden kann. Bei dezentralen Lösungen gibt es neben der Möglichkeit, verschiedene Typen von Brokern einzusetzen auch die Chance, multiple Instanzen desselben Brokers zu nutzen. Aus Betriebssicht wird die zweite Variante bevorzugt, mit dem zusätzlichen Effekt, dass unterschiedliche Policies auf den verschiedenen Brokerinstanzen zu einer besseren Anpassung an die jeweilige Consumergruppe führen. Die dezentralen Lösungen führen zu spezifischen Problemen in der Infrastruktur. Eine Verteilung der Funktionalität auf mehrere Broker führt zu einer Fragmentierung der Information über das System.

Außerdem ist unklar, ob die Summe aus lokalen Optima auch immer ein globales Optimum darstellt. Um dieses Problem zu vermeiden ist bei einer dezentralen Lösung eine Koordination zwischen den Brokern notwendig, um zumindest bei den Policies und der Ressourcenallokation eine gemeinsame Lösung zu finden. Die Allokation knapper Ressourcen ist die primäre Stelle, an denen zentrale Systeme dezentralen überlegen sind. Folglich müssen alle Broker Informationen über sich selbst mit den anderen Brokern austauschen können. Diese zusätzlichen Informationen stellen für das Gesamtsystem ein Dilemma dar. Je mehr an zusätzlicher Information ausgetauscht wird, desto besser ist die Ressourcenallokation im Gesamtsystem. Aber je mehr Information ausgetauscht wird, desto mehr Overhead an Austausch entsteht, der von den tatsächlichen Services weggenommen wird. In dem Extremfall sehr vieler Services und ganz weniger Broker nähert man sich der zentralen Lösung, mit einer Kommunikationskomplexität von:

$$\mathcal{I}_{\text{Broker}} \sim \mathcal{O}(1) .$$

Im umgekehrten Fall – jeder Service hat seinen eigenen Broker – sind die Broker nur noch mit sich selbst beschäftigt. In diesem Fall müssen alle Broker miteinander kommunizieren:

$$\mathcal{I}_{\text{Broker}} \sim \mathcal{O}(n^2) .$$

- In-band versus Out-of-band – Die Kategorisierung zwischen in- und out-of-band gibt an, wie ein Broker auf Requests reagiert. Die In-band-Broker sind für den Request als auch den Response verantwortlich, damit stellen sie eigentlich einen Kanal zwischen Consumer und Provider dar. Bei dieser Variante sind die Provider für den Consumer anonym, da jede Kommunikation nur über den Broker funktioniert. Die Out-of-band-Broker sind nur auf einer Seite der Kommunikation beteiligt. In diesem Fall nutzen die Consumer den Broker, um einen Provider zu finden, entkoppeln sich aber danach vom Broker und kommunizieren direkt mit dem Provider über Folgeaktivitäten.⁷² Der Vorteil dieser Variante ist eine deutlich niedrigere Basislast auf dem Broker, der jetzt fast nur noch wie eine Registry agiert.
- Informationbroker versus Servicevirtualizer – Die Informationbroker führen nur eine Softallokation durch. Dies bedeutet, dass der Broker Informationen über den Ort des Services bekannt gibt, aber keine Kontrolle über den tatsächlichen Service besitzt. Ein solcher Broker kann keine Garantien für die Verfügbarkeit oder die Qualität des jeweiligen Service geben. Typische Beispiele sind Broker, die wie Suchmaschinen agieren und dann dem Consumer eine URI zur Verfügung stellen. Ein Servicevirtualizer ist ein Broker, der eine harte Allokation durchführt, der Broker gibt Garantien für die Services, ihre Verfügbarkeit als auch ihre Qualitäten ab. Außerdem übermittelt er Informationen über die Ser-

⁷² Die Basisidee des Servicemodells (s. Abschn. 2.5.2).

vicebeschreibung. Der Broker erscheint einem Consumer wie der Service selbst und kann sogar einfache Funktionalitäten, wie Serviceaggregation, anbieten.

- **Brokerfokus versus Servicefokus** – Viele Brokerarchitekturen sind dafür konzipiert worden, den Services als Infrastruktur zur Verfügung zu stehen und haben damit einen Servicefokus. In sehr heterogenen Umgebungen kann es jedoch sinnvoll sein, dem Broker ein gewisses Eigeninteresse zu geben, ein solcher Broker wird dann andere Broker und andere Services für seine Aufgaben nutzen.

2.6.3 Enterprise Service Bus

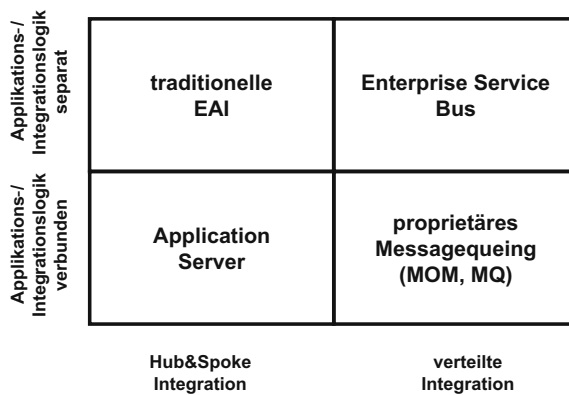


Abb. 2.17 Der ESB als Integrationsplattform

Informationen in einer Organisation sind oft auf verschiedenste Applikationen, Abteilungen und sogar Suborganisationen verteilt, wodurch es sehr schwierig ist, diese Informationen abzufragen oder zu konsolidieren. Der **Enterprise Service Bus (ESB)** ist ein Ansatz zur Integration von lose gekoppelten, verteilten Netzwerken. Der Begriff ESB wird mittlerweile von vielen Herstellern für ihre jeweiligen Produkte genutzt, ohne dass diese genau erklären, was ein ESB eigentlich leistet. Speziell die EAI⁷³-Hersteller geben ihren bestehenden Produkten gerne die Bezeichnung ESB, obwohl es sich bei diesen im Grunde um EAI-Systeme mit Webserviceprotokollinterfaces handelt.

Der ESB kann auf der einen Seite als Konzept verstanden werden, das es, als Infrastruktur für die Informationsweitergabe einer Organisation, aufzubauen gilt. Er kann aber auch als Produkt verstanden werden, da inzwischen zahlreiche Softwarehersteller eine solche Lösung anbieten. Ein wichtiges Kennzeichen des ESBs ist die Existenz eines allgemein zugänglichen homogenen Administrationsinterfaces,

⁷³ Enterprise Application Integration.

welches eine zentrale Konfiguration des gesamten Systems und damit aller angeschlossenen Subsysteme ermöglicht.

Tabelle 2.10 Lose und enge Koppelungen

Eigenschaft	Enge Koppelung	Lose Koppelung
Interaktion	Synchron	Asynchron
Messagestil	RPC	Dokument
Pfad	Fest kodiert	Gerouted
Technologiemix	Homogen	Heterogen
Datentypen	Abhängig	Unabhängig
Syntax	Konvention	Publiziertes Schema
Koppelung	Beim Design	Laufzeit
Semantische Adaption	Neu kodieren	Durch Transformation
Softwareziel	Effizienz	Breite Nutzbarkeit
Konsequenzen	Antizipiert	Unerwartet

Dieser Ansatz geht über das Konzept eines einfachen EAI-Brokers hinaus. Ein typischer EAI-Broker (meist MOM-basiert) bietet als Funktionalitäten:

- Konnektivität,
- Datentransformationssysteme,
- Adaptoren für Applikationen,
- Routing von Messages.

Ein ESB bietet dieselben Funktionalitäten, allerdings mit dem großen Unterschied, dass die Funktionen selbst Services sind. Somit können diese Funktionen unabhängig voneinander und verteilt implementiert werden. Im Zusammenhang mit einer SOA stellt der ESB das eigentliche Rückgrat der Laufzeitumgebung dar. Applikationen werden entkoppelt und durch den ESB über logische Endpunkte verbunden.

Die Charakteristika, die einen ESB auszeichnen, sind:

- Pervasive Computing⁷⁴ – Eines der Ziele hinter dem Einsatz eines ESBs ist es, innerhalb der Organisation und auch interorganisatorisch, Pervasive Computing zu erreichen. Dies hat zur Folge, dass der ESB eine globale Reichweite erlangt, in jeden Teil der Organisation wie auch in andere Organisationen vordringt. Services verbinden sich mit dem ESB, werden dadurch sichtbar für andere und können gleichzeitig andere „sehen“.
- Standardisierte Integration – Die standardisierte Integration ist eines der fundamentalen Konzepte des ESBs. Es sollten stets die vorhandenen Standards und Protokolle genutzt werden. Die fachlichen Interfaces werden zusammen mit ihren standardisierten Implementierungen in einer offenen „plug & play“-Architektur zusammengefügt. Das eigentliche Rückgrat bildet hierbei der ESB.

⁷⁴ Siehe Kap. 4.

Die immer stärkere Durchdringung der Organisationen mit Standards im technischen Umfeld ermöglicht erst die Entwicklung eines ESBs, denn ein ESB macht sich die Standardisierung zunutze.

- **Verteilte Umgebung** – Ein ESB muss in der Lage sein, für die Services einen Transportmechanismus zur Verfügung zu stellen, der bezüglich des Ausführungsortes transparent ist. Folglich muss der ESB eine verteilte Ausführungsumgebung unterstützen.
- **Selektives Deployment** – Bei der großen Anzahl von Services in einer Organisation sowie dem hohen Durchdringungsgrad des ESBs ist eine selektive und verteilte Einführung von Services notwendig.
- **Verteilte Datentransformation** – Ein wichtiger Teil der Integration ist die Fähigkeit Daten zu transformieren. Auf Grund der Verteilung des ESBs muss dies auch dezentral möglich sein.
- **Eventgetriebene SOA** – Innerhalb des ESBs werden Services als abstrakte Endpunkte von Verbindungen angesehen. Diese Endpunkte müssen auf asynchrone Ereignisse reagieren können. Die Unterstützung asynchroner Verarbeitung erhöht die Einsatzmöglichkeiten eines ESBs.
- **Workflowunterstützung** – Organisationen setzen immer stärker Workflowsysteme und Mechanismen zur Delegation ein, folglich muss der ESB auch diese unterstützen.
- **Verlässlichkeit** – Das Konzept einer verlässlichen Übertragung ermöglicht erst die Idee von verteilten Transaktionen im Umfeld des ESBs.
- **Autonomie** – Der ESB ist völlig losgelöst von der tatsächlichen Organisationsform, in der er eingebettet ist. Er ist autonom in dem Sinne, dass er Bestandteil einer alles durchdringenden Infrastruktur wird.
- **Dezentrale Steuerung des ESBs.**

Eine typische serviceorientierte Software muss die drei Rollen des Servicemodells unterstützen. Jeder Consumer muss ein Lookup durchführen, der es ihm ermöglicht, einen Service zu verwenden. Dafür müssen bestimmte Aufrufe in den Code des Consumers implementiert werden. Durch die Einbindung eines ESBs rufen Services sich nicht mehr gegenseitig auf, die Kommunikation und das Routing werden über den ESB erledigt. Es existieren zwar noch immer die Rollen, die Vermittlung dazwischen übernimmt aber der ESB. Dadurch wird es ermöglicht, die Kommunikation nicht mehr in den Code schreiben zu müssen, sondern die Abhängigkeiten über Konfigurationen und Deploymentinformationen angeben zu können. Eine SOA und eine ESB-Implementierung sind a priori nicht identisch (s. Abb. 2.18), insofern lässt sich ein ESB als eine spezielle Form einer SOA betrachten und wird oft als Enterprise-SOA bezeichnet.

Der ESB braucht ein Transportmittel, um in der Lage zu sein, Daten und Aufrufe an die entsprechenden Services weiterzugeben. Im Rahmen einer ESB-Implementierung wären durchaus unterschiedliche Möglichkeiten vorhanden, eine solche Transportschicht zu bauen, aber in der Praxis hat sich die Nutzung einer **Message Oriented Middleware (MOM)** als die heute favorisierte Lösung herausgestellt. Eine MOM übermittelt Daten zwischen verschiedenen Programmen durch

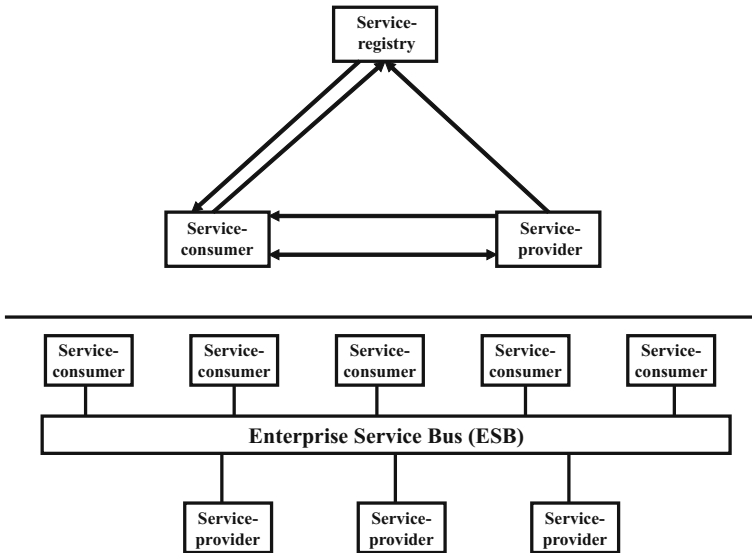


Abb. 2.18 Generisches SOA gegenüber einer ESB-Implementierung

Kommunikationskanäle, in denen feste Informationseinheiten (Messages) übertragen werden. Diese Messages enthalten neben der eigentlichen zu übertragenden Information (Payload) noch zusätzliche Informationen über Verteilweg, Struktur, Herkunft und so weiter.⁷⁵ Eine MOM-Umgebung ist üblicherweise asynchron aufgebaut, Sender und Empfänger wissen nur wenig voneinander. Das MOM-System übernimmt alle notwendigen Tätigkeiten, um die Message zu übermitteln. Die Nutzung eines MOM-Systems im ESB wird dadurch unterstützt, dass MOM-Systeme explizit eine Entkoppelung zwischen dem Sender und dem Empfänger vornehmen und damit auf natürliche Art und Weise eine lose Koppelung in den Services zwischen dem Consumer und dem Provider ermöglichen.

Das Übermittlungssystem der MOM garantiert, dass die gewählte Übertragungsoption eingehalten wird. Damit dies bei einer unsicheren Verbindung sichergestellt werden kann, wird intern ein Store and Forward System genutzt. Dabei wird die Message zunächst gespeichert, oft in einer Datenbank, es existieren aber auch einfache Dateisystemimplementierungen, und dann erst an einen Empfänger weitergeleitet. Ist der Empfänger nicht bereit, so kann der Prozess anhand der gespeicherten Message später noch einmal versucht werden. Neben der reinen Sicherstellung der Übertragung haben die meisten MOM-Systeme auch die Möglichkeit, die Reihenfolge von Messages sicherzustellen. Aus Performanzgründen dürfen Messages nur eine endliche Größe haben, von daher ist es üblich, große Messages in mehrere kleine aufzuteilen. Nach der Übertragung muss dann aber die Message wieder vollständig zusammengesetzt werden.

⁷⁵ Im Grunde ähneln MOM-Messages damit E-Mails, welche eine mögliche Erscheinungsform von Messages sind.

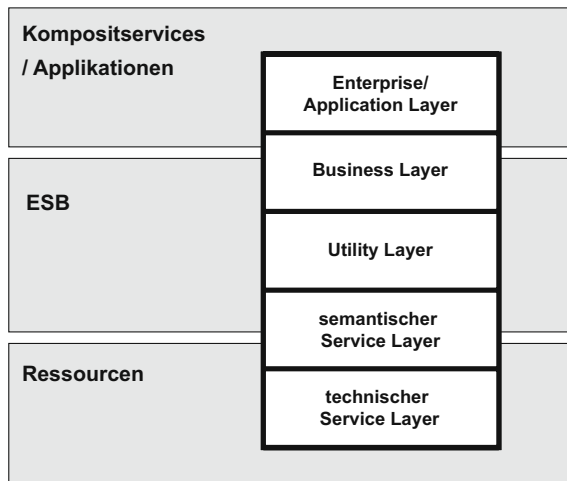


Abb. 2.19 Die einzelnen Layer einer SOA mit einem Enterprise Service Bus

Tabelle 2.11 Unterschiedliche Transporttechnologien im Vergleich

Kriterium	RPC	RMI	DCOM	Webservice	MOM	CORBA
Transaktionen	–	✓	✓	–	–	✓
Koppelung	Lose	Eng	Eng	Lose	Lose/eng	Eng
Standards	✓	✓	✓	✓	–	✓
Synchron	✓	✓	✓	✓	–	✓
Asynchron	–	–	–	✓	✓	–
Portierbar	✓	–	–	✓	✓	✓
Prozessunterstützung	–	–	–	✓	✓	–
Reliable	–	–	–	–	✓	–

2.6.4 Peer to Peer

In einer sehr dynamischen Umgebung oder in einem Ultra Large Scale System (s. Abschn. 6.9) funktioniert das Auffinden von Services nicht mehr so einfach, selbst dann, wenn eine Registry vorhanden ist. Das Fehlen einer zentralen Kontrolle macht dies sehr schwer. In diesem Fall ist es sinnvoller, bekannte P2P⁷⁶-Techniken einzusetzen. In einem solchen Netzwerk haben alle Beteiligten die gleiche Funktionalität, sie unterscheiden sich nur darin, ob sie eine bestimmte Ressource zur Verfügung stellen oder nicht. Damit man P2P zusammen mit Services nutzen kann, muss jeder Provider wie eine Ressource im P2P-Netz behandelt werden. In den P2P-Netzen sind allerdings Rollen wie Provider und Consumer sehr viel dynamischer als bei anderen Serviceprotokollen. Die P2P-Netzwerke kamen Ende der neunziger Jahre als direkte Folge der drastisch erhöhten öffentlichen Bandbreite auf, zunächst

⁷⁶ Peer-to(2)-Peer.

im Bereich des Instant Messaging⁷⁷. Die P2P-Protokolle wurden sehr schnell im privaten Umfeld für den Austausch von Dateien eingesetzt, mit allen daraus entstehenden lizenzrechtlichen Problemen, der einfache und anonym erscheinende Datenaustausch senkt die moralische Schwelle für illegales Raubkopieren. Bekannteste Vertreter von P2P-Netzwerken sind: *Napster*, *Gnutella*, *eDonkey*, *Overnet*, *Redswoosh*.

Das größte P2P-Netz jedoch ist heute immer noch das *SETI@home*⁷⁸ mit einer kombinierten Rechenkapazität von 10^{18} FLOPS⁷⁹. Für das Auffinden von Ressourcen (Services) existieren im P2P-Umfeld drei Basisstrategien:

- Zentralverzeichnismodell – In diesem Modell existiert ein Zentralverzeichnis aller Ressourcen (Services) und ihrer jeweiligen Adressen (URI). Die einzelnen Peers registrieren ihre Adressen und Ressourcen im zentralen Verzeichnis. Dieses Modell ist analog dem Registryansatz.
- Flooded Request Modell⁸⁰ – Hier werden viele redundante Messages im gesamten Netzwerk verschickt, um damit Adresse (URI) und Ressource (Service) anzukündigen. Oft wird versucht, die Gesamtzahl der Messages durch TTL⁸¹-Techniken, bei der die Messages eine endliche Lebensdauer haben und anschließend nicht mehr transportiert werden, zu reduzieren.
- DHT⁸²-Modell – Bei einem DHT-Netzwerk hat jeder Peer eine eindeutige ID und alle Peers bilden einen Ring. Die Position im Ring ist durch eine Hashfunktion und die ID gegeben. Jeder Peer kennt andere Peers und hat eigene Routingtabellen, welche durch die Distanz im Ring bestimmt werden. Neben den Peers wird auch jeder Ressource eine ID gegeben, so dass neben ihrer Existenz auch ihre Distanz innerhalb des DHT-Netzwerks bestimmt werden kann.

Eine solche Technik lässt sich auch einsetzen, um die Services als Ressourcen über ein P2P-Netzwerk zugänglich zu machen. Solche Netzwerke sind durchaus in der Lage große Systeme zu bilden. Speziell in öffentlichen Netzen können P2P-Systeme erfolgreich eingesetzt werden. Falls es mehrere Provider für denselben Service im P2P-Netz gibt, so lässt sich die Performanz eines Providers ermitteln:

$$T_{\text{predicted}}(i) = \frac{S}{1 - \lambda S} + \frac{1}{N} \sum_{n=1}^N D_n ,$$

hierbei ist $T_{\text{predicted}}(i)$ die vorhergesagte Responsezeit, λ die Ankunftsrate, S die mittlere Servicezeit und D_n die mittlere registrierte Zeitdifferenz für den n -ten Consumer und i -ten Provider. Auch schwierige Vorgänge wie Deployment sind relativ einfach in einer solchen Umgebung. Ein Nachteil existiert jedoch: Das Verschwinden von Services kann nicht aktiv publik gemacht werden, insofern müssen die

⁷⁷ Bekannteste Vertreter sind *ICQ* und *Skype*.

⁷⁸ Search for Extraterrestrial Intelligence.

⁷⁹ Sihe Abb. 1.4.

⁸⁰ *Gnutella* ist ein Vertreter dieser Art.

⁸¹ Time to Live.

⁸² Distributed Hash Table.

Consumer aktiv nach Ersatz suchen. Ein nicht mehr vorhandener Service würde $T_{\text{predicted}} \mapsto \infty$ bedeuten. P2P-Netze sind faktisch nicht administrierbar, innerhalb eines solchen „chaotischen“ Systems müssen Regelwerke für eine Steuerbarkeit sorgen, da keine zentrale oder dezentrale, beziehungsweise föderale Administrationsinstanz existent ist. In einem ULS-System (s. Abschn. 6.9) würde eine P2P-Implementierung durchaus angebracht sein, allerdings sind hier die Governance-mechanismen noch unklar. Neben der Frage der Sicherheit ist die Bandbreite im Netzwerk eine der limitierenden Faktoren hinter dem Einsatz von P2P-Systemen für Services. Der P2P-eigene Verwaltungsoverhead kann ein sehr großes Netzwerk stark belasten, mittlerweile existieren aber eigene P2P-Verwaltungsserver (zum Beispiel bei eDonkey), welche einen Teil dieser Administration übernehmen und so das Netzwerk deutlich entlasten, außerdem existieren schon die ersten Infrastrukturen, welche P2P-Servicenetze aufbauen, dazu zählen: JXTA⁸³, XtremWeb und BOINC⁸⁴.

2.6.5 Servicecontainer

Zwar ist es technisch möglich, dass ein ESB oder ein P2P-System eine Reihe von APIs zur Verfügung stellt und ein Service diese dann auch nutzt, aber eine solche Implementierung ist viel zu risikoreich:

- Der Service sollte von den Details des Transports nichts wissen.
- Damit die Infrastruktur die Services verwalten kann, müssen alle Services für die Infrastruktur identische Eigenschaften besitzen.

Daher muss der Service von der Infrastruktur isoliert werden, um die ex- und impliziten Abhängigkeiten zu minimieren. Ein mögliches Modell für eine Entkoppelung zwischen Service und Infrastruktur ist das Servicecontainermodell (s. Abb. 2.20). Ein solches Servicecontainermodell hat für den Betrieb immense Vorteile, da die Erfahrungen aus dem Bereich von EJB-Frameworks als auch die dort gemachten Erfahrungen bei der Policing und Steuerung solcher Container, inklusive Lastverteilung und Monitoring, recht gut übertragen werden können.

Die Managementinterfaces können nicht von außen aufgerufen werden, stellen folglich keine Services dar, sondern der Service reagiert auf Veränderungen seiner Umgebung, indem er aktiv nachfragt. Die Existenz solcher Interfaces, die keine Services implementieren, ist ein Bruch des Serviceorientierungsparadigmas, allerdings bildet das Gesamtsystem hiermit eine Art Metaservice, da das Verhalten im Sinne von Policies für das Gesamtsystem durch den Kontext der Umgebung gesteuert wird.

Eine der Aufgaben des Servicecontainers ist es, eine Ablaufumgebung für die Implementierung zu liefern und diese Implementierung der „externen Welt“ zur

⁸³ Akronym für **J**uxt**a**position, aus dem Lateinischen von *iuxta* (nebenan) und *positio* (Lage).

⁸⁴ **B**erkeley **O**pen **I**nfrastructure for **N**etwork **C**omputing.

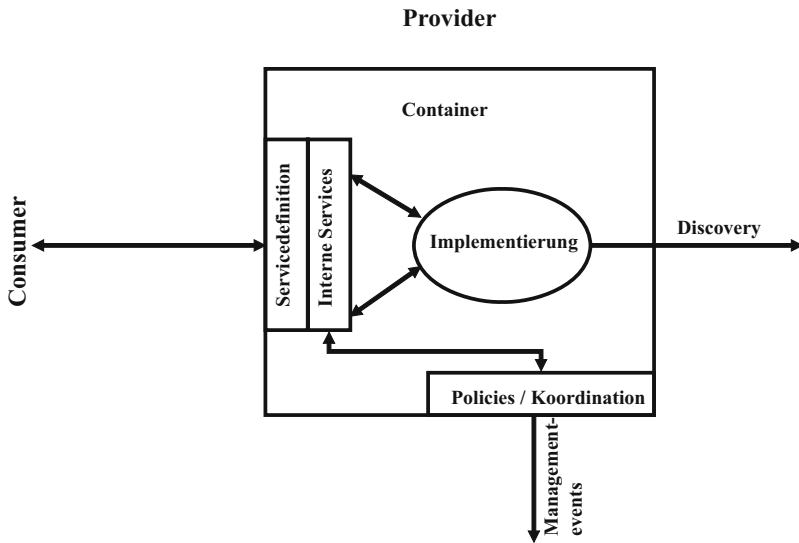


Abb. 2.20 Das Servicecontainermodell

Verfügung zu stellen. Die Kernfunktionalitäten eines solchen Servicecontainers sind:

- Discovery,
- Daten- und Protokolladapter,
- Konnektivität und Messageprocessing,
- Unterstützung für die dynamische Konfiguration (s. S. 43),
- Monitoring des internen Verhaltens und des Zustandes des Service,
- Mechanismen – Diese stellen Transaktionen, Sicherheit, Monitoring oder ähnliches zur Verfügung.

Der Servicecontainer sollte den gesamten Managementdatenfluss von und zu dem Service steuern. Dies betrifft speziell die Fähigkeiten in den Bereichen: Konfiguration, Monitoring und Audit, sowie Fehlerbehandlung. Damit der Service über den Servicecontainer mit der Infrastruktur überhaupt interagieren kann, enthält der Servicecontainer zwei separate Interfaces: Zum einen das Eingangsinterface und zum anderen das Ausgangsinterface. Die beiden Interfaces werden genutzt, um Messages an und vom eigentlichen Service zu senden und zu erhalten. Als Teil der Managementfähigkeiten muss der Servicecontainer die Möglichkeiten zum Audit und zur Protokollierung bieten. Dies verlangt jedoch, dass man in der Lage ist, den Service und sein Verhalten schon auf tiefer Ebene zu verfolgen.

Der Servicecontainer vermittelt zwischen der Infrastruktur und dem einzelnen Service, dafür bietet er eine Reihe von allgemeinen Funktionen (s. Abb. 2.21) an:

- Invocation,
- Deployment,
- Lifecyclemanagement,

- Transaktionsmanagement,
- Auditing, Tracking und Logging,
- Quality of Service (s. Abschn. 2.5.4).

Diese Trennung in die Eigenschaften des Servicecontainers und die des Service erlaubt es, einen Unterschied zwischen der Implementierung eines Service und entsprechenden Funktionen wie Auditing oder Transportfehlerbehandlung vorzunehmen, so dass die Serviceimplementierung sich auf ihre eigentliche fachliche Aufgabe konzentrieren kann. Bei anderen Funktionen wie den QoS führt diese Trennung zwischen Serviceimplementierung und dem Servicecontainer zu einer impliziten Standardisierung der QoS. Diese müssen jetzt für alle Services dieselben Kenngrößen erfassen. Da in diesem Modell der Servicecontainer fachlich neutral ist, sind es folglich die QoS auch. Mit den QoS können nur serviceübergreifende Größen erfasst und bewertet werden.

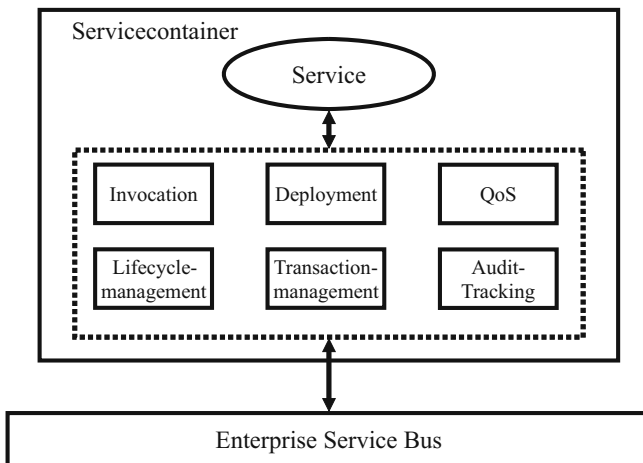


Abb. 2.21 Die Funktionalitäten eines Servicecontainers

Der eigentliche Aufruf eines Service innerhalb eines Servicecontainers ist die Hauptaufgabe eines Servicecontainers. Die übliche Abfolge (s. Abschn. 2.5) von Find, Bind und Execute ist für selten genutzte Fälle durchaus sinnvoll, aber im Rahmen eines großen Softwaresystems nicht gut einsetzbar. An dieser Stelle sorgt der Servicecontainer für eine praktikable Abkürzung. Im Rahmen der eingesetzten SOP muss die Servicedefinition von dem Mechanismus, Services zu finden und diese aufzurufen, getrennt werden. Fast alle Services nutzen über sehr lange Zeiten hinweg stets dieselben Services, warum sollte man sie daher andauernd suchen und wieder neu binden? Eine permanente Zuordnung ist gefragt. Und genau das leistet eine SOP.

Der Servicecontainer ermöglicht es, eine Entkoppelung zwischen dem Service und der Laufzeitumgebung sowie zwischen den einzelnen Services zu erreichen. Für den einzelnen Service innerhalb des Containers treten dabei eine Reihe von

Vereinfachungen auf. Durch den Servicecontainer und impliziten SOP wird der Service von der Verwendung konkreter Provideradressen (Endpunkte) befreit. Der Container übergibt diese Aufgabe an die SOP und fordert sie auf, anhand eines Regelwerks den geeigneten Provider zu finden und dynamisch zu binden. Diese Fähigkeit setzt allerdings voraus, dass die Servicekomposition dynamisch geschieht (s. Abschn. 2.7.4). Ein explizit genutzter Provider wird immer noch direkt angesprochen. Allerdings kann ein Servicecontainer diese Technik unterbinden, indem der Provider seine Services auch in einem Container zur Verfügung stellt und damit nur seine Interfaces dem Broker und sonst niemandem direkt gibt.

Die zweite Form der Entkoppelung ist die Entkoppelung von konkreten Interaktionsmustern. Der Container erlaubt es, synchrone und asynchrone, ja sogar parallele Aufrufe zu simulieren und damit den eigentlichen Service gegenüber den Komplexitäten des Interaktionsmusters zu isolieren. Die dritte Form der Entkoppelung durch den Servicecontainer ist die Neutralisierung der Messagestruktur. Die Mapping- und Transformationsregeln innerhalb der SOP können durch den Container genutzt werden, sodass der Service ein strukturell entkoppeltes Interface haben kann, welches die Chance auf langfristige Stabilität gibt (s. Abschn. 2.8.1).

2.7 Service Oriented Computing

Neben einer Architektur, welche die Nutzung von Services unterstützt, beziehungsweise einer Infrastruktur, welche den Aufruf von Services ermöglicht, muss man beim Einsatz von Services auch in der Lage sein Services zu konstruieren. Die entsprechende Disziplin hierfür ist das **Service Oriented Computing (SOC)**. Der Bau von Services ist nicht identisch mit dem Bau von Applikationen oder Komponenten. Zwar kommen auch bei der Implementierung von Services „klassische“ Design- und Programmiererkenntnisse zum tragen, aber das Design von Services verlangt noch mehr. Zum einen sollten Services möglichst zustandslos und zum anderen zu einem hohen Grad autonom sein. Des Weiteren zeigen Services ihre Mächtigkeit erst dann auf, wenn sie in einen fachlichen oder organisatorischen Kontext eingebettet werden. Das heutige SOC steht erst am Anfang seiner Entwicklung und wird, analog der Objektorientierung, vermutlich noch einige Jahre benötigen, um eine ähnliche Reife wie die Objektorientierung zu erlangen.

2.7.1 WSDL

Die **Web Services Definition Language (WSDL)** ist eine Spezifikation für netzwerk-basierte XML-Services. Die Sprache WSDL sieht auf den ersten Blick komplex aus, ist aber konzeptionell betrachtet einfach, insofern kann heute WSDL direkt⁸⁵ aus

⁸⁵ So verlockend diese Fähigkeit diverser Werkzeuge auch ist, Serviceinterfaces aus vorhandenem Code zu generieren, in aller Regel sind die daraus abgeleiteten „Services“ keine „echten“ Services im Sinne einer sinnvollen fachlich getriebenen Zerlegung.

vorhandenem Sourcecode generiert werden. WSDL ist ein notwendiger Bestandteil für die Infrastruktur von Webservices. Sie ermöglicht es, Providern ihre eigenen Services unabhängig vom darunterliegenden Protokoll zu beschreiben. De facto handelt es sich immer um SOAP als Protokoll. WSDL ist eines der Schlüsselemente für UDDI, da ohne eine klare Interfacesprache die Webservices nicht sinnvoll zu publizieren sind. WSDL benutzt XML, um seine Definitionen des Service an potentielle Consumer zu übertragen. Neben dem Namen der Datentypen wird auch der Portname, der logische Port des Servers, an den Consumer übertragen. Die einzelnen Definitionsmöglichkeiten sind:

- *types* – Eine Erweiterung der Standarddatentypen um eigene Typen ist hier möglich. Auf Grund der Erweiterungsfähigkeit von WSDL ist es auch durchaus möglich, ein anderes Typen-Definitions-System, wie zum Beispiel die CORBA Interface Definition Language (IDL) zu benutzen. Dazu wird dann einfach ein anderes Element in dem *types*-Element erzeugt.
- *message* – Die abstrakte Definition der zu übermittelnden Daten, welche auch komplexe oder hierarchische Strukturen enthalten kann. Innerhalb des *definitions*-Elements befinden sich mehrere *message*-Elemente. Jedes *message*-Element hat ein *name*-Attribut, das der Message einen eindeutigen Namen verleiht. Ein *message*-Element besteht aus einem oder mehreren *part*-Elementen, die wiederum mit einer bestimmten Typdefinition verknüpft sind. Für die Verknüpfung mit einer Typdefinition stehen die Attribute *element* oder *type* zur Verfügung.
- *portType* – Ist eine Liste der abstrakten Operationen getrennt nach Input- und Outputmessages. Die *portType*-Elemente, es können ein oder mehrere Elemente sein, bekommen durch ihr Attribut *name* jeweils einen eindeutigen Namen verliehen. Jedes *portType*-Element definiert eine Menge von benannten Operationen. Abhängig von der Art der Operation befindet sich innerhalb eines *operation*-Elements ein *input*- oder ein *output*-Element. Die Operationsarten können in folgende Kategorien eingeteilt werden:
 - one-way – Der asynchrone Aufruf des Service. One-way kann nur eine Eingabe empfangen und sendet keine Antwort.
 - notification – Asynchron mit Antwort.
 - request-response – Empfangen, dann Senden einer Message.
 - solicit-response – Senden, dann Empfangen einer Message.

Bei den beiden asynchronen Aufruftypen werden keine Exceptions produziert. Das *input*- und das *output*-Element legen das abstrakte Messageformat für ein- und ausgehende Messages fest. Dies geschieht, indem sie mit einem bereits definierten *message*-Element verknüpft werden.

- *binding* – Das *binding*-Element definiert Messageformat und Protokolldetails für Operationen und Messages, die durch ein *portType*-Element identifiziert werden. Es können beliebig viele dieser *binding*-Elemente für ein bestimmtes *portType*-Element existieren. Damit ist es möglich, ein und dieselbe Operation über verschiedene Protokolle anzubieten. Jedes der Elemente wird durch ein *name*-Attribut benannt. Das zu verbindende *portType*-Element wird durch das *type*-Attribut festgelegt.

- *port* – Die Adresse für das *binding*-Element.
- *service* – Beinhaltet die Menge der *port*-Elemente.
- *import* – Um XML-Definitionen zu importieren.

Eine der Grundlagen von WSDL ist der Einsatz von Namespacekonstrukten in XML. Da WSDL in der Lage sein muss, verschiedenste Webservices abbilden zu können, müssen Homonyme innerhalb des jeweiligen Kontextes sauber aufgelöst werden. Damit die WSDL-Dokumente innerhalb einer UDDI-Registry einfacher zu finden sind, werden diese Dokumente in zwei Klassen eingeteilt:

- Serviceinterfaces – Das Serviceinterface enthält die Elemente: *types*, *import*, *message*, *portType* und *binding*. Hierbei wird das *import*-Element genutzt, um andere Serviceinterfaces zu importieren.
- Service Implementations – Das Implementationsdokument enthält nur die Elemente: *import* und *service*, wobei *service* Informationen über die Ports und die Bindings enthält, inklusive der jeweiligen URL.

2.7.2 Semantische Services

Der Aufbau von Prozessen in „regulären“ Services ist ein Vorläufer für den Aufbau von Prozessen auf Grund von semantischen Services. Die semantischen Services gehen über eine rein syntaktische Beschreibung des Interface hinaus, da der Service für seine Auffindung mit zusätzlicher fachlicher Information über die Bedeutung des Service angereichert wird. Ein solcher semantischer Service ist ein Triplet aus Angaben:

$$\mathbb{S}_{\text{semantisch}} = \left\{ \left(\bigcup \mathbb{S}_{\text{SLA}} \right), \left(\bigcup \mathbb{O}_{\text{semantisch}} \right), \left(\bigcup \mathbb{S}_{\text{Policies}} \right) \right\},$$

hierbei wird der semantische Service $\mathbb{S}_{\text{semantisch}}$ durch die Summe der SLAs, $\bigcup \mathbb{S}_{\text{SLA}}$, die Summe der semantischen Operationen $\bigcup \mathbb{O}_{\text{semantisch}}$ und die Menge an unterstützten Policies $\bigcup \mathbb{S}_{\text{Policies}}$ beschrieben. Eine solche Beschreibung geht über die einfachen WSDL-Interfaces hinaus und ermöglicht es, den Service aus semantischer Sicht heraus zu finden und dadurch einsetzen zu können. Zur tatsächlichen Einordnung von semantischen Services ist es in der Regel notwendig, die semantisch beschriebenen Operationen durch Verweise auf eine Ontologie oder Taxonomie (s. Abschn. 2.8.4) näher zu erläutern. Ohne diese Einordnung in domänen-spezifische Informationshaushalte degradiert die Idee des semantischen Service zu einer Art „abstrakten“ WSDL-Sprache. Eine besondere Form der semantischen Services sind die semantischen Webservices (s. Abb. 2.22). Auf Grund der Historie sind die semantischen Webservices eine Mixtur aus Webservices und dem semantischen Web, ein Ansatz, der versucht, das WWW durch Bedeutungen zu ergänzen und damit mächtiger zu gestalten.

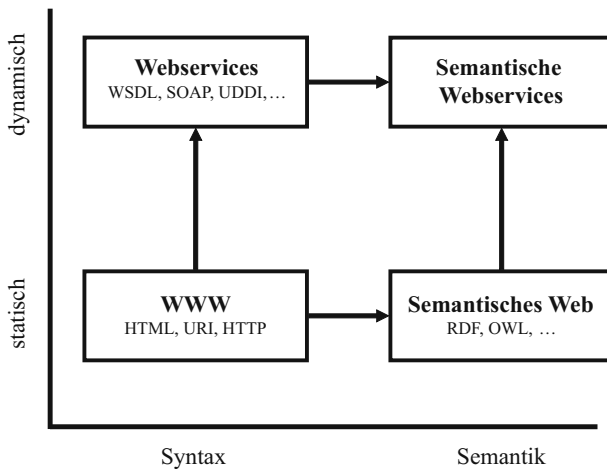


Abb. 2.22 Die Entwicklungen bei den semantischen Webservices

Die semantischen Operationen werden üblicherweise in zwei Klassen eingeteilt:

- Funktionssemantik – Es wird die Bedeutung der ausgeführten Funktion und anderer Funktionen dokumentiert.
- Datensemantik – Die Daten werden nicht nur durch ihre Typen beschrieben, sondern durch die Bedeutung der Daten und ihrer jeweiligen Zusammenhänge.

Bei der Beschreibung der SLAs nimmt man in der Regel eine Beschreibung der Bedeutung der QoS-Eigenschaften vor. Die Auswirkungen der Policies beziehen sich bei den semantischen Webservices auf die Darstellung der Ausführungssemantik. Die semantischen Services lösen sich komplett vom Prozessgedanken und ihren jeweiligen Kontexten heraus. Da man trotzdem aus den semantischen Services Prozesse zur Abwicklung von Aufgaben aufbauen muss, ist es notwendig, einen abstrakten Prozess zu definieren. Ein abstrakter Prozess stellt einen Prozess dar, welcher schon die Kontroll- und Datenelemente enthält, aber die konkrete Ausprägung der genutzten Services findet erst zum Ausführungszeitpunkt statt. Der Vorteil dieses Vorgehens ist, dass die Komplexität des Kontroll- und Datenflusses manuell kontrolliert werden kann und sich die Auswahl des entsprechenden semantischen Service automatisieren lässt. Diese Strategie ist dann sinnvoll, wenn sich Ausführungskosten und Zwangsbedingungen oder Kontexte ändern, aber der Kontrollfluss selbst relativ stabil ist. Um einen abstrakten Prozess im Rahmen von semantischen Services zu erzeugen, können alle Konstrukte der Prozessmodellierung genutzt werden. Der einzige Unterschied besteht darin, dass semantische Templates anstelle von Services genutzt werden. Die Templates ermöglichen es, diese später (zur Laufzeit) durch „echte“ semantische Services zu ersetzen. Diese Templates dienen als „Vorstufe“ zu den eigentlichen Services. Das Ziel der semantischen Services ist es, die Automatisierung im Umgang mit Services in den Bereichen Servicedis-

covery, Servicenutzung und Servicekomposition deutlich zu erhöhen und damit billiger Software bauen zu können.

2.7.3 Servicemodellierung

Die Modellierung von Services durchläuft, wie die Modellierung jeder Form von Software, die Phasen Analyse, Design und Implementierung und diese in aller Regel mehrmals. Bei der Serviceorientierung sind Zyklen notwendig, um überhaupt sinnvoll zu modellieren, mit dem Unterschied, dass die Zykluszeiten in der Serviceentwicklung deutlich kürzer sind als in der „klassischen“ Software. Am besten eignen sich agile Verfahren wie eXtreme Programing oder Scrum für den Bau von Services. Die dort gewonnenen Erfahrungen mit dem Bau von Komponentensoftware können nahtlos auf den Bau von Services übertragen werden.

Neben den Prozessen und Services, beziehungsweise deren Komposition, müssen auch die Messages und die Events sowie die ausgetauschten Informationen modelliert und als Typen implementiert werden. An dieser Stelle spielen Ontologien eine große Rolle, da sie die funktionale Information der Interfaces mit den statischen Informationen der Messages, Events und Informationsstrukturen verknüpfen können. Das Design von Services wird durch unterschiedliche Anforderungen getrieben, dabei kann ein Service nur dann als „gut designed“ gelten, wenn er wiederverwendbar ist.

Die Basisfragen im Servicedesign sind:

- Welcher Service wird für den Geschäftsprozess gebraucht?
- Wie wird dieser Service gebaut?

Die Antwort auf die erste Frage liefert die Serviceanalyse und die Antwort auf die zweite das Servicedesign. Die Serviceanalyse muss die vollständige Domäne betrachten, in der der entsprechende Geschäftsprozess angesiedelt ist. Folgende Schlüsseigenschaften unterscheiden die Serviceanalyse von traditionellen Verfahren:

- Services werden in multiplen Systemen wiederverwendet. Ziel der Analyse ist es nicht, eine einzelne Applikation zu bauen, sondern den Service so zu konzipieren, dass er in diversen potentiellen Systemen Wiederverwendung findet.
- Die Domäne muss in ihre organisatorischen und funktionalen Aspekte zerlegt werden, um die entsprechenden Prozesse zu identifizieren. Die so entstehenden Subdomänen werden modelliert und durch Services repräsentiert. Simultan ist aber auch eine Repräsentation in Form von Objekten notwendig, da sonst eine Implementierung der Services nicht möglich ist.
- Der Service muss so strukturiert werden, dass er agil einsetzbar ist.
- Nichtfunktionale Eigenschaften der Services bilden einen wichtigen Aspekt der Serviceorientierung, diese müssen schon während der Analyse formuliert und berücksichtigt werden.

Eine der Basisfragen hinter der Serviceanalyse ist: Wie wird eine guter Abstraktionsgrad für Services erreicht, wenn nur das Geschäftsprozessmodell und die abstrakten Businessrequirements bekannt sind? Auf die Services angewandt bedeutet dies, das richtige Maß an Granularität für die entsprechenden Kontexte zu finden.

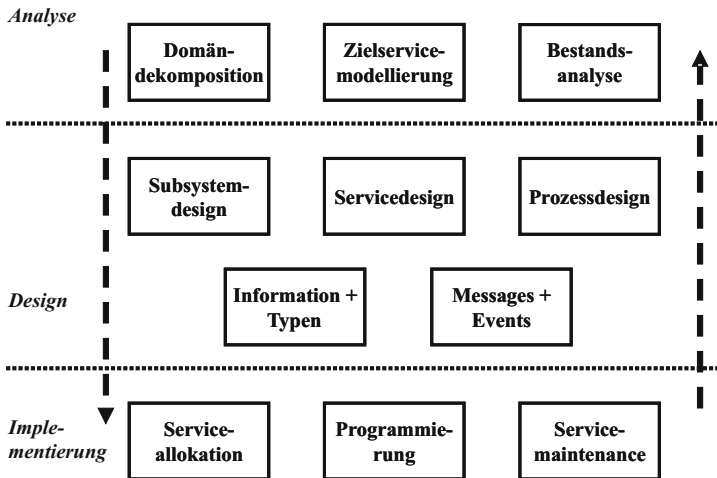


Abb. 2.23 Die Servicemodellierung verläuft zyklisch in den Phasen Analyse, Design und Implementierung

Das auffälligste Merkmal des serviceorientierten Designs ist die Trennung zwischen dem Verhalten von Komponenten und persistenten Objekten. Ein gutes Servicedesign hat drei Charakteristika:

- Die im Interface aufgeführten Methoden gehören zu den fachlichen Methoden der Domäne, nicht zu einfachen Datenzugriffen.
- Das Interface wird durch Elemente des Domänenmodells beschrieben.
- Die Services sind zustandslos und unabhängig vom jeweiligen Kontext.

Um Services zu modellieren, ist es am einfachsten in Komponenten zu denken. Diese stellen jeweils abgeschlossene Blöcke aus Funktionalitäten dar, welche dem Consumer zur Verfügung gestellt werden. An dieser Stelle wird schon zwischen Interface und zukünftiger Implementierung differenziert. Für einen solchen Service sind drei Aspekte entscheidend:

- Der Kontext, in dem der Service existiert. Ein Service existiert nie in vollständiger Isolation, er erfüllt eine spezifische Rolle in einem Kontext und kommuniziert auch aktiv mit diesem. Services können zusammenwirken und neue Services erzeugen und umgekehrt können Services auch in kleinere Services zerlegt werden. Hierfür ist eine Koordination zwischen den Services mit Hilfe des Kontexts wichtig.

- Der Servicevertrag, der von dem Service zur Verfügung gestellt werden soll. Dieser repräsentiert die Summe an Informationen, die für einen Consumer wichtig sind, ohne dass er die Implementierung kennen muss.
- Die Implementierung des Vertrages in gegebenen Kontexten. Die Implementierung eines Service kann auf beliebige Art und Weise erfolgen.

Ein zunehmend wichtiger Teil des Kontextes ist die Spezifikation und das Design für Events und Kommunikationsmuster. Services werden zwar direkt aufgerufen, die übergreifende Steuerung von Abläufen geschieht jedoch über Events und Kommunikationsmuster. Als steuernde Elemente müssen sie auch spezifiziert werden, um die „einfachen“ Services überhaupt einsetzbar zu machen.

Die fachliche Granularität eines Service ist ein Maß dafür, wie viel Fachlogik ein Service beinhaltet, je gröber die Granularität, desto mehr Fachlogik wird im Service gekapselt. Im Gegensatz dazu gibt die Interfacegranularität nur an, wie groß der Ausschnitt des Objektmodells ist, welches durch den Service bearbeitet wird. Üblicherweise ist eine grobe Granularität besser, da weniger Services entstehen und die Fachlichkeit konzentrierter auftaucht. Allerdings haben grobgranulare Services den Nachteil, dass bei ihnen die Änderungswahrscheinlichkeit ansteigt und dass sie im Allgemeinen zu einer hochgradigen Spezialisierung tendieren, was die Wiederverwendung deutlich erschwert.

Tabelle 2.12 Grobgranulare Interfaces

Vorteil	Probleme & Risiken
Alle Daten in einem einzigen Aufruf.	Komplexe Daten und -strukturen. Große Messages. Erhöhte Komplexität durch die Behandlung multipler Ausnahmen in einem Servicerequest.
Zustandsinformationen können explizit Bestandteil des Aufrufs werden.	Übermittlung der Zustände kann zum falschen Zustand führen. Daten, die beim letzten Aufruf gültig waren, sind beim nächsten ungültig. Bei jedem Aufruf müssen die Daten neu validiert werden.
Selbstbeschreibende Daten sind möglich. Der komplette Kontext kann übertragen werden.	Services werden nur für ganz spezifische Szenarien gebaut und sind nicht wiederverwendbar.

Die verschiedenen Sichten der Abstraktion und der Granularität unterscheiden sich darin, dass eine Granularität sich stets auf eine Ebene der Abstraktion bezieht (immer auf das Abstraktionsniveau des jeweiligen Servicelayers). Innerhalb eines objektorientierten Systems wird die einzelne Softwareeinheit durch Klassen von Objekten geprägt: modelliert, entworfen und implementiert durch die Anwendung der Prinzipien der Objektorientierung. Innerhalb einer komponentenbasierten Architektur bilden die Komponenten die Ebene der Granularität, obwohl eine Komponente auch eine einzelne Klasse (im Sinne der Objektorientierung) sein kann, bilden in aller Regel mehrere Klassen eine Komponente aus. Folglich sind komponentenbasierte Architekturen grobgranularer als objektorientierte. Die Serviceorientierung

Tabelle 2.13 Feingranulare Interfaces

Vorteil	Probleme & Risiken
Kleine Messages enthalten einfache Daten.	Möglicherweise entsteht ein Übertragungszustand, der zu zusätzlicher Komplexität führt. Wiederherstellung bei fehlerhafter Übertragung muss implementiert werden. (Checkpoint/Restart, beziehungsweise Kompensation)
Individuelle Services können zu größeren komponiert werden.	Der Consumer muss die exakte Aufrufsequenz der Services verstehen.
Erhöhte Laufzeitflexibilität. Einzelne Services werden nur bei Bedarf angesprochen.	Komplexe Beschreibung im Sinne von Prä- und Postkonditionen. Individuelle Services haben keinen eigenen Kontext. Performanzprobleme durch die vielen Aufrufe.

besitzt eine abweichende Granularität, da diese den Service als die elementare Einheit betrachtet. Da Services beliebig gewählt werden können, rangiert ihre Größe von sehr groß (vollständige Applikationen) bis hin zu sehr klein (einzelne Methode eines Objekts). Für den Fall einer sehr feinen Granularität degeneriert eine service-orientierte Entwicklung zu einem objektorientierten Design und im Rahmen einer sehr groben Granularität wird die Siloarchitektur der Legacysysteme erreicht.

2.7.4 Servicekomposition

Die Zusammensetzung neuer Services aus bestehenden Services kann auf zwei Arten geschehen, zum einen statisch und zum anderen dynamisch. Die statische Komposition entspricht in etwa der „klassischen“ Programmierfähigkeit nur mit dem Unterschied dass keine Unterprogramme oder Bibliotheken aufgerufen werden, sondern Services. Bei der dynamischen Komposition ist die Situation schwieriger, hier sind die Services zum Designzeitpunkt nicht bekannt, dafür existiert aber eine Vorstellung darüber, welche Anforderungen an den Service zu stellen sind. Die gefundenen Services müssen in diesem Ansatz mit den Vorstellungen über die gewünschten Eigenschaften verglichen werden, genannt Matching. Beim Matching werden die Eigenschaften des Providers mit den Anforderungen des Consumers verglichen und einer der angebotenen Services wird gewählt. Das Matching hängt von zwei Voraussetzungen ab:

- Die Services der Provider müssen bekannt gemacht worden sein.
- Es muss eine Beschreibung der Consumeranforderungen vorhanden sein.

Wenn beide Vorbedingungen gegeben sind, dann lässt sich ein Matching in zwei logischen Schritten vollziehen. Zuerst wird nach einem semantisch gewünschten Service gesucht und in der gefundenen Menge nach der geeigneten Implementierung. Bei einem einzusetzenden Service muss nicht unbedingt der gesamte Service exakt passen, es reicht auch aus, wenn nur Fragmente des Service in semantischer wie struktureller Hinsicht angemessen sind. Eine Komposition von Services kann

auf verschiedenen Ebenen ablaufen, von der technischen über die semantische Ebene bis hin zur Geschäftsprozessebene. Da eine Komposition von Services auf der Geschäftsprozessebene stets eine Komposition auf den anderen Ebenen voraussetzt, ist es sinnvoll, ein Kompositionsmodell durch mehrere Ebenen aufgebaut zu modellieren. Es existieren drei Basismechanismen, unabhängig von der betrachteten Ebene, um Services zu komponieren (s. Abschn. 5.13):

- Horizontal – Das horizontale Komponieren entspricht einer „Supply Chain“. Die Services werden nacheinander aufgerufen und jeder nachfolgende Service nutzt den Output des vorhergehenden Service.⁸⁶
- Vertikal – Das vertikale Komponieren entspricht dem „Subcontracting“, ein Service nutzt intern andere Services, um seine Aufgaben zu erledigen.⁸⁷
- Hybrid – Im hybriden Fall werden vertikale und horizontale Mechanismen gemeinsam eingesetzt.

Es lässt sich ein Grad D für Komponierbarkeit durch folgende Formel angeben:

$$\mathfrak{D}(S_i, S_j) = \sum_{p=0}^{p_{\max}} \alpha_p \left(\sum_{q=0}^{q_{\max}} w_{pq} \mathfrak{F}(S_i, S_j) \right). \quad (2.3)$$

Hierbei läuft p über die Ebenen und q über die Attribute. Die Gewichte w_{ij} und α_i können je nach Aufgabenstellung gewählt werden. Die Funktion $\mathfrak{F}$ gibt an, ob das gewählte Kriterium erfüllt ist oder nicht. An Hand der Gl. 2.3 lässt sich ein Grad für die Verknüpfbarkeit zweier Services automatisiert ermitteln. Falls eine automatische Komposition tatsächlich eingesetzt wird, empfiehlt es sich, Schwellenwerte für D festzulegen.

Eine spezielle Methode des Matchings ist das Conversationmatching. Dabei wird die Struktur des Prozesses für den die Services eingesetzt werden sollen, mit den möglichen Zuständen und Zustandsübergängen des nutzenden Serviceprozesses verglichen. Dies funktioniert natürlich nur bei zustandsbehafteten Services, die ihr Zustandsmodell publiziert haben! Für den Consumer stellt dann der Service einen endlichen Zustandsautomat dar. Eine Überprüfung der benötigten Zustände (Globaler Zustandsautomat) mit den zur Verfügung gestellten Zuständen durch den Service liefert ein Matchingmuster und damit eine Selektion von Services.

2.7.5 Serviceinteroperabilität

Interoperabilität ist eine Beziehung zwischen zwei Services, bei der die beiden Services in der Lage sind, spezifische Informationen gemeinsam zu nutzen und auf Basis dieser Informationen mit einer gemeinsamen operationellen Semantik zu agieren. Die Interoperabilität wird bei der Entwicklung der meisten Softwaresysteme schon zu einem frühen Zeitpunkt angestrebt. Die Interoperabilität spielt sich auf

⁸⁶ Unter Unix ist dies als Piping bekannt.

⁸⁷ Dies entspricht in traditionellen Programmiersprachen einem direkten Call.

verschiedenen Ebenen quasi aufsteigend ab (s. Abb. 2.24). Interoperabilität ist eine der zentralen Forderungen an einen Service, da nur durch diese die langfristige Verwendung des Service dauerhaft sichergestellt werden kann. Interoperabilität innerhalb einer Organisation ist schon lange bekannt und wird meist (fälschlicherweise) mit dem Begriff EAI belegt, mit dem Unterschied, dass im Rahmen von EAI die Interoperabilität ex post versucht wird, während bei Services diese ex ante notwendig ist. Insofern hat die Interoperabilität eines Service die gleiche Wichtigkeit wie dessen fachliche Funktionalität, während bei der EAI die Interoperabilität mehr schlecht als recht nachträglich versucht wird. Auf der Serviceebene kann bei der Interoperabilität zwischen vier Gebieten unterschieden werden:

- Signaturebene,
- Protokollebene,
- Semantikebene,
- Kontextebene.

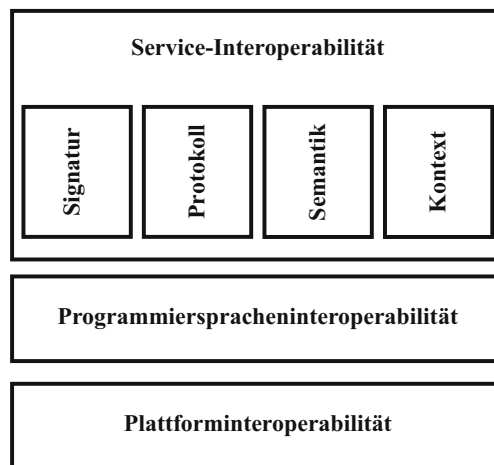


Abb. 2.24 Die Ebenen der Interoperabilität

Wann ist ein Service interoperabel? Es existieren zwei Merkmale, dies zu beurteilen: Zum einen die Kompatibilität und zum anderen die Ersetzbarkeit. Zwei Services gelten als kompatibel, wenn sie die Fähigkeit haben korrekt zusammenzuarbeiten, das heißt alle zwischen diesen beiden Services ausgetauschten Messages werden verstanden. Insofern handelt es sich bei der Kompatibilität um eine symmetrische Relation. Damit eine Kompatibilität zwischen zwei Services möglich ist, müssen beide Services auf allen Ebenen kompatibel sein, mit der Folge, dass beide Services innerhalb einer Ebene ein gemeinsames „Verständnis“ haben. Für dieses gemeinsame Verständnis sind notwendig:

- **Signatur** – Beide Services müssen eine korrespondierende Signatur besitzen. Dies stellt die Kompatibilität auf dem syntaktischen Gebiet sicher. Der Consumer muss alle von ihm genutzten Interfaces des Providers syntaktisch verstehen.
- **Protokoll** – Auf der Protokollebene bedeutet Kompatibilität, dass ein gemeinsames Verständnis über die Menge aller gültigen Zustandsübergänge vorhanden ist.
- **Semantik** – Beide Services müssen der in der Kommunikation verwendeten Syntax die gleiche Bedeutung beimessen, das heißt, deren Semantik kennen und identisch interpretieren.
- **Kontext** – Wenn beide Services ein identisches Verständnis über die internen und externen Einflussfaktoren besitzen, sind beide in Bezug auf den Kontext kompatibel.

Die Interoperabilität in Form der Kompatibilität auf Protokollebene ist dann gegeben, wenn alle Restriktionen bezüglich des Ablaufs der Serviceinteraktion beachtet werden. Auf der Semantikebene ist die Kompatibilität gegeben, wenn das Verhalten eines Service mit dem Verhalten übereinstimmt, welches die anderen Beteiligten von diesem Service erwarten. Dies betrifft primär die Erwartungshaltung des Consumers gegenüber der Serviceimplementierung. Meist überprüfen die Provider Präkonditionen und die Consumer die entsprechenden Postkonditionen, um die Kompatibilität auf Semantikebene sicherzustellen. Auf der Ebene des Kontexts sind zwei Services kompatibel, wenn sie das gleiche Verständnis bezüglich der ausgeführten Interaktion in ihrer Umgebung haben.

Die Ersetzbarkeit ist im Gegensatz zur Kompatibilität keine symmetrische Relation zweier Services. Wenn ein Service durch einen anderen ersetzt werden kann, dann muss der andere Service nicht a priori äquivalent sein. Genauer betrachtet ist ein Service $\mathcal{A}$ ersetzbar durch einen Service $\mathcal{B}$, wenn er folgende Kriterien erfüllt:

- **Signatur** – Es muss eine vollständige und eindeutige Abbildung der Signatur von $\mathcal{A}$ auf $\mathcal{B}$ möglich sein, $\mathcal{B}$ kann mächtiger als $\mathcal{A}$ sein.
- **Protokoll** – Es muss eine vollständige und eindeutige Abbildung der möglichen Zustände von $\mathcal{A}$ auf $\mathcal{B}$ möglich sein, $\mathcal{B}$ kann größer als $\mathcal{A}$ sein.
- **Semantik** – Wenn die Semantik von $\mathcal{A}$ in $\mathcal{B}$ enthalten ist und alle Prä- und Postkonditionen vollständig und eindeutig abbildbar sind, dann ist der Service $\mathcal{A}$ durch $\mathcal{B}$ semantisch ersetzbar.
- **Kontext** – Es muss eine vollständige und eindeutige Abbildung der externen Einflüsse des Services $\mathcal{A}$ auf die externen Einflüsse des Services $\mathcal{B}$ existieren.

Die Interoperabilität auf Signaturebene ist durch den Bezug der Interfaces beider Services gegeben, da diese Definitionen das gemeinsame syntaktische Verständnis zwischen Consumer und Provider darstellen. Eine direkte Ersetzbarkeit ist dann gegeben, wenn der Ersatzservice mindestens die Menge an Methoden und die darin verwendeten In- und Outputparameter besitzt, welche der zu ersetzende Service anbietet.⁸⁸ Die Ersetzbarkeit auf Protokollebene erfordert die Überprüfung, ob die

⁸⁸ Für einen Service mit WSDL-Beschreibung müssten für die Ersetzbarkeit die Elemente *types*, *message* und *portType* zusammen passen.

Reihenfolge der Ein- und Ausgabe sowie der Aufrufzeitpunkt beider Services konsistent zueinander sind. Theoretisch gesehen ist ein Service zustandslos, folglich kann er jederzeit aufgerufen werden, in der Praxis sind zustandslose Services eher selten anzutreffen, insofern spielt die Reihenfolge und der aktuelle Zustand eine wichtige Rolle.

Eine Möglichkeit, die Ersetzbarkeit auf semantischer Ebene zu beurteilen, ist das Behavioural Subtyping. Beim Behavioural Subtyping wird das Verhalten zweier Services auf eine Vererbungsbeziehung zwischen beiden zurückgeführt: Wenn das Verhalten einer Instanz einer abgeleiteten Klasse (Service $\mathfrak{B}$) konsistent mit dem Verhalten einer Instanz der Superklasse (Service $\mathfrak{A}$) ist, dann ist $\mathfrak{A}$ durch $\mathfrak{B}$ ersetzbar. Die kontextuelle Ersetzbarkeit eines Services $\mathfrak{A}$ durch einen Service $\mathfrak{B}$ bezüglich eines dritten Services $\mathfrak{C}$ ist gegeben, wenn $\mathfrak{B}$ in jedem relevanten Aspekt bezüglich der Interaktion zwischen $\mathfrak{C}$ und $\mathfrak{A}$ kontextuell konsistent ist. Konsistent werden zwei Interaktionen genannt, wenn ihre relevanten Aspekte aufeinander abbildbar sind.

Die semantische Interoperabilität ähnelt sehr stark dem Problem des Verstehens von Sprache zwischen einzelnen Menschen. Zwar gibt es jahrhundertalte Forschungen auf diesem Gebiet, jedoch ist man immer noch sehr skeptisch darüber, ob Menschen sich wirklich vollständig verstehen. Angewandt auf das Problem der semantischen Interoperabilität bedeutet dies, dass es keine eindeutige Interpretation einer Servicespezifikation gibt und folglich der Automatisierung Grenzen gesetzt sind, da eine gemeinsame „natürliche“ und sichere semantische Basis fehlt. Das Serviceinterface wird vom Consumer, dem Provider oder dem Softwareentwickler in Form der Implementierung unterschiedlich interpretiert und daher mit Semantik gefüllt. Eine solche gemeinsame Basis kann durch eine gemeinsame Ontologie geschaffen werden.

Neben der syntaktischen und semantischen Interoperabilität zweier Services müssen auch die QoS miteinander verglichen werden, da auch diese Information ein, zumindest aus Sicht des Consumers, wichtiges Merkmal sein kann. Speziell in großen heterogenen Umgebungen sind die QoS nicht immer a priori identisch für jeden Provider. Einer der Gründe für eine Differenz der QoS liegt in der Vorgehensweise der Softwareentwickler auf Providerseite begründet: Interfaces (syntaktisch und semantisch) werden direkt oder indirekt zum Designzeitpunkt fixiert, während die QoS im laufenden Betrieb an eine sehr spezifische Umgebung angepasst werden.

2.7.6 π -Kalkül

Die meisten heutigen Systeme haben als Basisidee einen zustandsbehafteten Workflow.⁸⁹ In einem solchen Umfeld spiegelt der softwaregestützte Ablauf einen Verwaltungsprozess wider, strukturelle Veränderungen sind sehr selten und werden in

⁸⁹ Verarbeitungsschritte erfolgen fast immer sequenziell und jede „passierte“ Station verändert ein Attribut; diese Veränderung wird vom Folgeschritt ausgewertet.

einer vorweg antizipierten Adaption berücksichtigt. Solche Applikationen sind zu starr und zu inflexibel, um auf rasche Veränderungen zu reagieren. Mittlerweile steht die Flexibilität und nicht mehr die rigide Ablaufkontrolle bei Software im Vordergrund. Eine rigide Ablaufkontrolle führt die Idee des SOEs (s. Abschn. 2.2) ad absurdum. In neuen Systemen dürfen Abläufe nicht durch Zustandsänderungen ausgelöst und verfolgt werden, sondern durch Events. Diese Events werden produziert und konsumiert durch Aktivitäten, welche keine statischen Verbindungen zu anderen Aktivitäten haben, sondern nur weitere Events absetzen und so das Gesamtsystem steuern. Events werden als Vorbedingungen genommen, um Aktivitäten auszulösen und ein Ergebnis zu produzieren. In einer solchen Architektur existiert keine zentrale Kontrollinstanz, welche alles steuert, vielmehr erlaubt die lose Kopplung mittels der Events auch In- und Outsourcing von Aktivitäten. Außerdem ermöglicht die Ereignissteuerung eine flexible Integration von Aktivitäten in andere Aktivitäten.

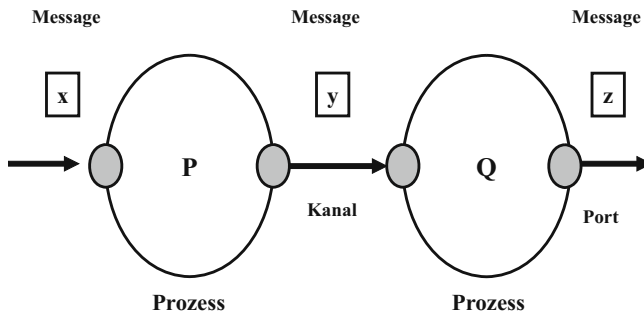


Abb. 2.25 Flussdiagramm des π -Kalküls

Traditionelle Integration braucht eine Kontrollinstanz, welche den Überblick über alle Teile besitzt. Messagebasierte Systeme hingegen haben keinen Bedarf an dieser künstlichen Kontrollschicht. Innerhalb von messagebasierten Systemen haben die einzelnen Teile nur noch Vor- und Nachbedingungen und führen eine Transformation des Inputs auf den Output durch. Neben dem In- und Output existiert eine einzelne Aktivität oder ein Service in einem definierten Kontext. Die Veränderung des Kontexts ist eine Eigenschaft, die alle offenen Systeme haben, da hier nicht vorausbestimmbar ist, welcher Kontext für die Ausführung einer Aktivität oder eines Service gerade der aktuelle ist.

Historisch gesehen waren zuerst sequenzielle Systeme mit vordefinierten Statusübergängen die Grundlage der meisten heutigen Applikationen und Workflowsysteme. Solche sequenziellen Systeme werden mathematisch durch das λ -Kalkül beschrieben. Systeme mit Parallelität und Verteilung oder mit mobilen Services benötigen eine gegenüber dem λ -Kalkül erweiterte mathematische Grundlage: Das π -Kalkül. Eine detaillierte mathematische Beschreibung der Grundlagen des π -Kalküls befindet sich in Anhang D.

Sequenzielle Systeme können formal durch das λ -Kalkül beschrieben werden. Dieses Kalkül kann Rekursionen und Aufrufe nach dem einfachen EVA-Prinzip⁹⁰ beschreiben. Eine typische Repräsentation für sequenzielle Systeme sind die Flussdiagramme. Auf tieferer Ebene lassen sich viele Programmiersprachen durch das λ -Kalkül beschreiben. Parallele Systeme lassen sich nicht mehr so einfach durch Flussdiagramme beschreiben, hier sind andere Mechanismen notwendig, so zum Beispiel Petrinetze. Speziell beim Auftreten multipler Instanzen des gleichen Service tun sich Petrinetze schwer. Um die Limitierungen der Petrinetze zu überwinden, wurden Theorien für mobile Systeme entwickelt. Mobile Systeme sind notwendigerweise immer auch parallel. Das für mobile Systeme entwickelte π -Kalkül repräsentiert Mobilität durch die Beschreibung der Bewegung von Verknüpfungen in einem abstrakten Raum von verknüpften Prozessen.⁹¹ Das π -Kalkül benutzt das Konzept von wertbeschränkten Namen (eine Sammelbezeichnung für Kanäle, Links und so weiter) für die Interaktion zwischen parallelen Prozessen. Diese Fähigkeit des π -Kalküls macht es zu einer idealen mathematischen Grundlage für das Service Oriented Computing.

2.8 Service Oriented System Engineering

Die Softwareentwicklung in den nächsten Jahren wird sich anderen Herausforderungen stellen müssen, zum einen Services mit loser Koppelung, Autonomie und Verteilung und zum anderen dem Aufbau immer komplexerer Systeme (s. Abschn. 6.9). Die Umgebung der Services ist stark heterogen und verändert sich permanent, Services müssen sich auf diverse Verhaltensmuster ihrer jeweiligen Umgebung einstellen und diese explizit nutzen können. Das heute dominante Verfahren der Kenntnis aller Hard- und Softwarekomponenten zum Designzeitpunkt wird mit zunehmender Vernetzung und Verteilung immer weniger möglich sein, folglich steht die Runtime- und nicht die Designtimeevolution des resultierenden Systems im Vordergrund. Aber nicht nur das Verhalten der anderen Services und Systeme, sondern auch die genutzten Daten werden immer heterogener und immer weniger verlässlich werden, die Vielfältigkeit steigt stark an.

Das **Service Oriented System Engineering**⁹² (SOSE) betrachtet die für ein Serviceorientierungsparadigma notwendigen Systeme, erstellt sie und zeigt Wege, wie eine **Service Oriented Software** (SOS) gebaut werden kann. SOSE beschäftigt sich nicht damit, wie ein einzelner Service gebaut wird, sondern damit, wie man aus Services Systeme und Funktionalität bauen kann. Daher liegt beim SOSE der Schwerpunkt der Betrachtung auf dem Prozess und der Funktionalität, nicht auf dem einzelnen Service. Der einzelne Service ist für das SOSE ein atomarer Teil. Speziell bei der Servicekomposition ist die Trennung zwischen SOC und SOSE willkürlich, da hier beide den Prozess betrachten müssen, aber SOC kommt dabei vom Service,

⁹⁰ Eingabe Verarbeitung Ausgabe (EVA).

⁹¹ Analog zu dynamisch erzeugten Hyperlinks in Hypertexten.

⁹² Manchmal auch als **Service Oriented Development of Applications** (SODA) bezeichnet.

mit dem Ziel neue Services zu schaffen, während SOSE ein System aus Services aufbauen möchte. SOSE ist eine disziplinübergreifende Tätigkeit, die ein System aus Services aus dem Blickwinkel des Lebenszyklusses, der Kybernetik und der Kundensicht betrachtet.

Die heute vorliegenden Implementierungen zeigen noch nicht das volle Potenzial, welches im Serviceorientierungsparadigma steckt, dies hat zwei Ursachen:

- Viele der heutigen SOA-Implementierungen sind durch einfaches Wrapping bestehender Applikationen entstanden. Diese Applikationen waren aber nie für Architekturen in Form einer SOA ausgelegt, mit der Folge, dass sie nur bedingt serviceorientiert sind.
- Heute werden neue Services fast immer von der IT initiiert. Das implizite Entwicklungsmodell dahinter ist providerzentriert. Solche Services reflektieren das, was der Provider glaubt, was genutzt werden kann. Wirklich gute Services können aber erst dann entstehen, wenn eine kundenzentrierte Sicht eingenommen wird.

Eine aus Services erstellte Software besitzt eine inhärente Unsicherheit, da sie verteilt und dynamisch zusammengestellt ist, beides liefert abweichendes Verhalten und mögliche Fehlerquellen. Die Vermischung von unterschiedlichen Granularitätsebenen verschärft diese Situation noch zusätzlich.

2.8.1 Evolution und Komplexität

Die Services und die aus den Services entstandenen Systeme entwickeln sich stets weiter. Dabei unterliegen Software und Services durchaus unterschiedlichen Gesetzmäßigkeiten. Eine Software befindet sich immer in einem definierten Zustand; das heißt implizit, dass jede Software einen Lebenszyklus durchlebt. Dieser Lebenszyklus ist die definierte zeitliche Abfolge von Zuständen. Das Lebenszyklusmodell als Ganzes zeigt die möglichen Zustände auf, welche die Summe aller Softwaresysteme annehmen können; insofern ist der Lebenszyklus ein generisches, idealtypisches Modell von Software.

Grundlage des Lebenszyklusmodells der Software ist die Beobachtung, dass jede Veränderung der Software einen Übergang von einem Zustand der Software in einen anderen Zustand darstellt:

$$\hat{O}(t, t_0)|\psi(t_0)\rangle \rightarrow |\psi(t)\rangle, \quad (2.4)$$

wobei der zeitliche Unterschied $\delta t = |t - t_0|$ hinreichend klein sein sollte. Wenn die beiden Zustände $|\psi(t_0)\rangle$ und $|\psi(t)\rangle$ sehr nahe beieinander liegen, so gehören beide zum gleichen Zustand des Lebenszyklusmodells. Ein solches Vorgehen lässt aber die Vergleichbarkeit unterschiedlicher Systeme als sehr fragwürdig erscheinen. Aus diesem Grund werden die möglichen Zustände nicht nach internen, quantifizierbaren Eigenschaften der Software eingeteilt, sondern nach äußeren, qualitativen Eigenschaften. Die Zustände werden üblicherweise nach der jeweiligen Form der

Tabelle 2.14 Die unterschiedlichen Softwareparadigmen

Eigenschaft	Serviceorientierung	Objektorientierung	Strukturierte Programmierung
Methodik	Entwicklung durch Identifizierung lose gekoppelter Services, welche zu einer SOS zusammengefasst werden. (Lego-Prinzip)	Entwicklung durch Identifizierung eng gekoppelter Klassen mit einer hierarchischen Architektur, die auf der Vererbung beruht.	Entwicklung durch Identifizierung von Gruppen von fachlichen Funktionen, die Zustände im System verändern oder Funktionen aufrufen.
Abstraktion und Kooperation	Entwicklung verteilt: Applicationbuilder, Servicebroker, Serviceprovider. Applicationbuilder verstehen Softwarelogik und Domäne, Provider verstehen Programmierung.	Entwicklung durch ein einzelnes Team. Entwickler müssen programmieren und Domänenkenntnisse besitzen.	Entwicklung durch ein einzelnes Team. Entwickler müssen programmieren und Domänenkenntnisse besitzen.
Wiederverwendung	Zweistufig: Wiederverwendung von Services und Code in der Implementierung	Wiederverwendung von Code durch Vererbung und Klassenbibliotheken, seltener durch Laufzeitbibliotheken.	Wiederverwendung von Code mit Copy & Paste, seltener durch Laufzeitbibliotheken.
Dynamic Binding, Rekombosition	Serviceaufruf wird zur Laufzeit dynamisch gebunden. Services können auch erst zur Laufzeit entdeckt werden.	Assoziation zwischen Name und Methode zur Laufzeit, die Methode muss in ausführbarem Code gelinkt sein.	Statische Koppelung zwischen Name und statischem Code.
Maintenance	Service und SOS werden unabhängig voneinander verändert.	Softwareveränderung in Form von Releases und Patches durch Verteilung von Upgrades.	Softwareveränderung in Form von Releases und Patches durch Verteilung von Upgrades.

Entwicklung klassifiziert; dies zu einem gewissen Grad willkürlich, es hat sich aber in der Praxis ein hohes Maß an Übereinstimmung über alle Softwaresysteme hinweg gezeigt. Diese Form der Einteilung ermöglicht es, eine system- und organisationsübergreifende Vergleichbarkeit sicherzustellen. Relativ klar wird das Lebenszyklusmodell (s. Abb. 2.26), wenn es als ein Stufenmodell betrachtet wird, das sich zunächst nur entlang der Zeitachse bewegen kann. Die so entstehenden Stufen sind:

- **Initialer Software-Zustand** – Die initiale Software bezeichnet den Anfangszustand, den die Software hat, wenn sie zum ersten Mal in Betrieb genommen wird. Die initiale Software besitzt schon die Architektur der Software, welche während des gesamten Lebenszyklus erhalten bleiben wird.
- **Entwicklung** – Das Ziel der Entwicklungsstufe ist es, sich an die stetig verändernden Benutzeranforderungen sowie die Veränderungen der Umwelt anzupassen und Defekte aus der initialen Stufe zu beheben. In der Entwicklungsstufe werden das zunehmend gesammelte Wissen der Endanwender über die Software und

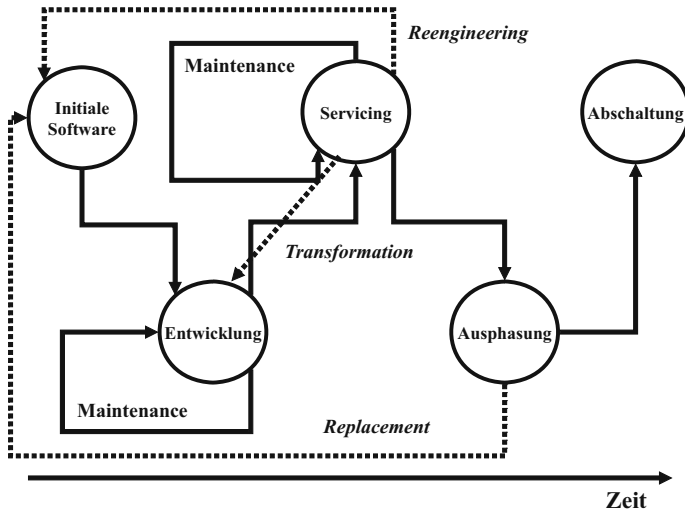


Abb. 2.26 Der Lebenszyklus von Software

das Wissen der Softwareentwickler über die Domäne zu treibenden Kräften, um die Software selbst zu verändern.

- Servicingzustand – Ist das Ende der Entwicklungsstufe erreicht, so tritt die Stufe des Servicing ein. In der Servicingstufe finden zwar auch noch Veränderungen statt, diese lassen aber die Software im Kern unverändert. Typischerweise spricht man in dieser Phase bei Veränderungen von „Patches“ und „Bugfixes“.
- Ausphasungszustand – Während der Ausphasung werden keine „Patches“ mehr geliefert, wie noch im Zustand des Servicing; damit unterliegt die Software keinerlei Maintenance mehr. Eine längerfristige Benutzung der Software in dieser Phase durch die Endbenutzer führt zu einer Anhäufung von „Work-Arounds“. Auf Grund der Tatsache, dass die Software sich nicht mehr verändert, werden Teile in ihr semantisch reinterpretiert und anders benutzt, als ursprünglich intendiert. Mit der Einführung eines „Work-Arounds“ verändert der Endanwender unbewusst das System, dadurch wird der „Work-Around“ zu einer Quasifunktionalität⁹³ des Systems.
- Abschaltungszustand – Die Abschaltung der Software ist der finale, meist irreversible, Schritt. Die Software wird entfernt und steht nicht mehr zur Verfügung.

Dieses Lebenszyklusmodell für Services führt dazu, dass die ersten beiden Phasen stark komprimiert werden und die dritte (Servicingphase) die längste sein sollte, da das Ziel der Services eine hohe Stabilität ist. Aus Sicht eines autopoietischen Systems (s. Anhang A.8) durchläuft ein Service nur zwei Phasen, Bootstrapping (die Erstinstallation) und Evolution, durch permanente Veränderung seiner Struktur mit dem Versuch ihn zu optimieren.

⁹³ Dies kann so weit gehen, dass offensichtliche Fehler zu Features werden.

Neben den einzelnen Phasen ist es wichtig, sich die Evolution von Software und speziell von Services vor Augen zu führen. Die sogenannten „Softwareevolutionsgesetze“ gehen auf *Lehman* zurück. Die sieben Gesetze der Softwareevolution zeigen auf, welche Größen der Software (Services und Applikationen, eventuell mit einer unterschiedlichen Skala) sich wie, mit zunehmendem Lebensalter, ändern:

- **Kontinuierliche Veränderung –**

I. Software, die genutzt wird, muss sich kontinuierlich anpassen, ansonsten wird sie sehr schnell nicht mehr nutzbar sein.

Bei der Software wird der Veränderungsdruck durch die Differenz zwischen den Anforderungen der Domäne und damit implizit den Anforderungen der Organisation und den implementierten Eigenschaften der Software ausgelöst. Je größer diese Differenz, desto höher ist der Druck, die Software zu verändern. Ein Teil des Veränderungsdrucks wird durch die Software selbst produziert: Jede neue Installation oder Version der Software ändert durch ihren Einsatz die Domäne ab und erzeugt damit implizit eine Differenz zwischen der Anforderung der Domäne und der Software selbst. Dieser Druck ist wichtig, damit sich Software überhaupt entwickelt. Fehlt dieser Druck, indem beispielsweise die Software „eingefroren“ wird, so wird die Software rapide obsolet. Dieses Gesetz resultiert daraus, dass Veränderungen der Domäne die einmal zu einem früheren Zeitpunkt getroffenen Annahmen ungültig machen.

- **Wachsende Komplexität –**

*II. Die Komplexität einer Software wächst stetig an, außer es wird Arbeit investiert, um diese Komplexität zu reduzieren.*⁹⁴

Die wachsende Komplexität resultiert aus der stetig steigenden Zahl von Änderungen an der Software. Durch die Gesetze *I* und *VII* bedingt, müssen permanent Veränderungen an der Software vorgenommen werden, mit der Konsequenz, dass jede einzelne Änderung zu einer Erhöhung der Komplexität führt, da es immer eine nicht verschwindende Wahrscheinlichkeit für inkompatible und unstrukturierte Änderungen gibt. Mit zunehmender Komplexität wird es immer schwerer, die einmal erreichte Komplexität auf dem Anfangsniveau zu halten. Die Änderung der Komplexität, sprich das Komplexitätswachstum eines Systems beim Hinzufügen eines neuen Services, ist infinitesimal gegeben durch:

$$d\phi = \mathcal{K}(\phi) , \quad (2.5)$$

wobei ϕ ein gegebenes Komplexitätsmaß ist. Hierbei ist $d\phi$ der Komplexitätszuwachs und $\mathcal{K}$ eine nichtverschwindende Funktion der Komplexität. Für kleine Systeme ergibt sich im Grenzfall:

$$\lim_{\phi \rightarrow 0} \mathcal{K}(\phi) = k_0 > 0 ,$$

sodass sich für kleine ϕ die Änderung des Komplexitätsmaßes zu

$$d\phi \approx k_0$$

⁹⁴ Diese Beobachtung entspricht dem zweiten Gesetz der Thermodynamik: Die Entropie eines geschlossenen Systems ist streng monoton wachsend.

ergibt. Auf der anderen Seite kann gezeigt werden, dass das Funktional $\mathcal{K}$ sich für große ϕ wie ein Potenzgesetz verhalten muss, das heißt $\mathcal{K}(\phi \gg 1) \sim \phi^v$, mit einer nichtnegativen Konstanten v . Das Lehman'sche Gesetz besagt nun, dass die Zahl der Quellmodule ϑ eines Softwarepakets einer einfachen Differentialgleichung genügt:

$$\frac{\partial \vartheta}{\partial t} = c_1 \vartheta + \frac{c_2}{\vartheta^2}. \quad (2.6)$$

Diese Differentialgleichung korreliert die Zahl der Quellmodule ϑ mit der Zeit. Näherungsweise lässt sich die Differentialgleichung durch

$$\lim_{t \rightarrow 0} \vartheta \approx \sqrt[3]{3c_2 t} \quad \text{und} \quad \lim_{t \rightarrow \infty} \vartheta \approx e^{c_1 t} \quad (2.7)$$

lösen, was auch als das Turski-Lehman'sche Wachstumsmodell bekannt ist. Dieses Wachstumsmodell lässt sich auch auf Services übertragen, da Services wie jede Software aus Quellmodulen ϑ aufgebaut sind. Alternativ zu Gl. 2.6 kann für Systeme das Wachstum durch die Services für große Zeiträume ($t \rightarrow \infty$) angenähert werden durch:

$$\frac{\partial N_{\text{Services}}}{\partial t} = \frac{\partial}{\partial t} \sum_{i \in \text{Services}} \vartheta_i = \sum_{i \in \text{Services}} \left(c_{1,i} \vartheta_i + \frac{c_{2,i}}{\vartheta_i^2} \right) \approx \alpha N_{\text{Services}}. \quad (2.8)$$

Hierbei wächst die Zahl der Services exponentiell an. Da jede Änderung eine Steigerung der Entropie zur Folge hat und eine Änderung sich in erster Näherung in einem proportionalen Wechsel der Entropie niederschlägt, lässt sich die Entropieänderung (s. Gl. 2.5) durch $\Delta S \sim S$ beschreiben. Die Folge ist, dass sich für die Entropie in erster Näherung eine Differentialgleichung der Form $\partial S / \partial t = \alpha S$ ergibt. Die Auswirkungen dieser einfachen Entropieentwicklung sind drastisch. Jeder noch so kleine Unterschied bei der Startentropie zum Zeitpunkt t_0 resultiert in einem noch größeren Unterschied in der Entropie zum Zeitpunkt $t_1 > t_0$.

- **Selbstregulierung –**

III. Die Evolution von Software ist selbstregulierend mit Produkt- und Prozessmetriken, welche nahe an einer Normalverteilung liegen.

Eine große Zahl dieser Mechanismen wirken auf die Software als Treiber oder als stabilisierende Kräfte. Da die Zahl dieser einzelnen Kräfte relativ hoch ist, kann die Entwicklung der Software als ein Subsystem betrachtet werden, welches in ein großes System eingebettet ist. Dieses Gesamtsystem übt nun durch eine große Zahl von quasi unabhängigen Entscheidungen einen Druck auf das System aus, welcher einer Normalverteilung ähnlich ist. Wenn dieser Druck eine Zeit lang auf die Entwicklung und damit implizit auf die Software angewandt wird, ist die Eigendynamik so stark, dass sich das Subsystem ein Equilibrium sucht.

- **Erhaltung der organisatorischen Stabilität –**

IV. Die mittlere effektive Aktivitätsrate, welche für eine evolvierende Software angewandt wird, bleibt während der Lebensdauer der Software konstant.

Die tatsächlichen Kostentreiber in Unternehmen sind meistens externer Natur: Verfügbarkeit von qualifiziertem Personal und Ähnliches. Auf der anderen Sei-

te ist ein großer Teil des Aufwands durch das Gesetz *III* vorgegeben: Um die Trägheit des Gleichgewichtszustands zu überwinden, ist sehr viel Aufwand notwendig. Die Folge dieser hohen Trägheit ist eine quasi konstante effektive Aufwandsrate.

- **Erhaltung der Ähnlichkeit –**

V. Die Inhalte von aufeinander folgenden Releases innerhalb einer Software sind statistisch konstant.

Einer der Faktoren bei der Veränderung von Software ist das Wissen aller Beteiligten über die Zielsetzung hinter der Software. Die Entwicklung tendiert dazu, in jeder der einzelnen Versionen besonders viel an neuer Funktionalität unterzubringen, denn es möchte einen „Mehrwert“ suggerieren. Die entgegengesetzte Kraft ist die Notwendigkeit, dass alle Beteiligten die große Menge auch verstehen und umsetzen müssen. Der Fortschritt und die Dauer sind auch beeinflusst durch die Menge an Information, welche gewonnen werden muss. Das Verhältnis zwischen Informationsmenge und Aufwand zur Gewinnung der Information ist nicht linear, sondern der Aufwand erhöht sich ab einer gewissen Größenordnung so drastisch, dass von diesem Moment an die Gesamtmenge an zu gewinnender Information unüberwindbar erscheint. Die Folge dieser beiden wirkenden Kräfte ist eine statistische Konstanz bezüglich der Inhaltsmenge pro Release.

- **Wachstum –**

VI. Die funktionalen Inhalte einer Software müssen stetig anwachsen, um der Endbenutzererwartung auf Dauer gerecht zu werden.

Wenn eine neue Software eingeführt wird, existiert immer eine Differenz zwischen der implementierten Funktionalität und der tatsächlich gewünschten. Mit zunehmender Dauer wird der Ruf nach der Implementierung dieser fehlenden Funktionalität stärker, was über die Feedbackmechanismen zur Erweiterung der vorhandenen Funktionalität führt. Mathematisch gesehen gilt: $dV/dt = \mathcal{M}(t)$, wobei V das Volumen in einer geeigneten Form misst und $\mathcal{M}(t)$ eine positive Funktion der Zeit ist. Das Volumen der Software ergibt sich dann als das Integral $V(t) = \int_0^t \mathcal{M}(\tau) d\tau$ mit der Eigenschaft: $V(t > t_0) \geq V(t_0)$.

- **Nachlassende Qualität –**

VII. Die Qualität der Software wird als nachlassend empfunden, solange keine massiven Anstrengungen zur Adaption vorgenommen werden.

Eine Software kann nie alle Anforderungen exakt treffen, dafür ist die Zahl der Anforderungen zu groß. Da die reale Welt sich permanent verändert, steigt die Unschärfe zwischen der möglichen Gesamtmenge von Anforderungen an, während die Zahl der implementierten Anforderungen auf Grund der Systemgrenzen praktisch konstant bleibt. Die Unschärfe wächst also! Der Endbenutzer empfindet die so anwachsende Unschärfe als einen Verlust an Qualität. Je größer diese Unschärfe wird, desto schwieriger ist es für den Endbenutzer, das Verhalten der Software erwartungskonform vorauszusagen.

Zwar hängen die Evolutionsgesetze nicht von der Granularität der Softwarebestandteile ab, aber die Geschwindigkeiten einzelner Gesetze können sehr wohl durch Services beeinflusst werden. Daher muss die Designmaxime für Services und Systeme lauten:

Das Design von Services muss so sein, dass diese langfristig stabil bleiben und das Design der Applikation und Funktionen muss so sein, dass es auf permanente Veränderungen eingestellt ist.

Diese Maxime ist sinnvoll, da in aller Regel der Prozess eine viel größere Flexibilität verlangt, als die Informationen, die durch ihn manipuliert werden. Prozesse sind unterschiedlichen Meinungen und Moden durch den menschlichen Consumer ausgesetzt, wie auch massiv durch Marktgegebenheiten beeinflusst. Daher müsste das Design so sein, dass zukünftige Veränderungen antizipiert werden. Dies ist aber nicht immer möglich. Das Beste, was man erreichen kann, ist es für die Erweiterbarkeit einer aktuellen Lösung um zukünftige Anforderungen zu sorgen. In einem großen System, bestehend aus vielen SOS-Applikationen, wird sich die Zahl der Services auf Dauer nach dem Turski-Lehman'schen Wachstumsmodell (Gl. 2.8) verhalten, das heißt am Anfang werden wenige Services entstehen, deren Anzahl dann aber stetig anwächst und im Grenzfall $t \mapsto \infty$ gegen

$$N_{\text{Services}} \sim e^{\alpha t} \quad (2.9)$$

geht. Das Komplexitätswachstum jedoch wird in einem großen System nicht durch die Zahl der Services bestimmt sein, sondern durch die mögliche Zahl der Verknüpfungen zwischen den einzelnen Services:

$$N_{\text{Interaktionen}} \sim \mathcal{O}(N_{\text{Services}}^2) . \quad (2.10)$$

Aus den beiden Gleichungen 2.9 und 2.10 folgt, dass das Gesamtsystem ein exponentielles Komplexitätswachstum verzeichnet:

$$\mathcal{K} \sim e^{\eta t} . \quad (2.11)$$

Serviceorientierte Systeme haben somit dieselben Probleme wie andere Softwaresysteme in Form eines exponentiellen Komplexitätswachstums. Vermutlich wird das Komplexitätsproblem bei einem auf Services basierenden System sogar drastischer sein als bei einem nichtvernetzten System. Bei nicht vernetzten Systemen erzwingt das siloartige Verhalten, dass sich die Komplexität faktisch additiv verhält⁹⁵:

$$\mathcal{K}_{\text{gesamt}} \approx \sum_{i \in \text{System}} \mathcal{K}_i ,$$

mit der Folge, dass die Entropien sich in etwa addieren:

$$S_{\text{gesamt}} \approx \sum_{i \in \text{System}} S_i .$$

Durch die Einführung der Vernetzung werden die Silos aufgebrochen und damit die vorher gekapselte Komplexität externalisiert, was dazu führt, dass die entsprechende Komplexität stärker als additiv wächst:

⁹⁵ Im Sinne der Vielfältigkeit (s. Anhang. C.3.1).

$$\mathcal{K}_{\text{gesamt}} \approx \sum_{i \in \text{Service}} \mathcal{K}_i + \sum_{i,j} \mathcal{K}(i,j) + \sum_{i,j,k} \mathcal{K}(i,j,k) + \dots,$$

und die Entropie stärker als linear skaliert:

$$S_{\text{gesamt}} = \sum_{i \in \text{Service}} S_i + S_{\text{Interaktion}} \gg \sum_{i \in \text{Service}} S_i. \quad (2.12)$$

Je kleiner die einzelnen Services sind und je größer ihre Anzahl, desto stärker dominiert die Interaktionsentropie, so dass im Grenzfall $n_{\text{Service}} \mapsto \infty$ der Zustand $S_{\text{gesamt}} \approx S_{\text{Interaktion}}$ erreicht wird.

Serviceorientierte Systeme gehen damit das Risiko ein, sehr viel schneller einen Komplexitätstod zu sterben als traditionelle Systeme. Ab einem gewissen Grad an Komplexität wird das Gesamtsystem nach dem Ashby-Conant-Theorem (s. Abschn. A.3) nicht mehr steuerbar und degradiert damit zu einem unbeherrschbaren System (s. Abschn. A.9), da die Vielfältigkeit nach Gl. C.5 und Gl. 2.12 durch

$$V \geq S_{\text{gesamt}} \gg \sum_{i \in \text{Service}} S_i$$

sehr groß wird.

2.8.2 Grenzen der traditionellen Verfahren

Traditionelle Methoden wie Wasserfallmodell oder der Rational Unified Process sind für eine Systementwicklung zum Teil sehr erfolgreich; als Beispiel kann hier die Mondlandung oder der Bau der ersten Atombomben dienen, trotzdem funktionieren sie wirklich gut nur unter bestimmten Randbedingungen. Diese sind typischerweise:

- Das gewünschte Ergebnis ist a priori bekannt, ist klar und unzweideutig beschreibbar.
- Es muss einen einzelnen zentralen Manager des ganzen Projekts geben.
- Veränderungen werden durch eine zentrale Instanz geregelt und freigesetzt.
- Es müssen austauschbare Ressourcen, welche auch wieder neu vergeben werden können, vorhanden sein.

Allerdings haben diese traditionellen Methoden auch ihre Grenzen. Sie sind nicht anwendbar, wenn eine der folgenden Bedingungen gegeben ist:

- Turbulente Umgebungen mit der Folge von permanenten und sehr schnellen Veränderungen der Anforderungen innerhalb der Organisation oder Domäne.
- Schlechtdefinierte Probleme –
- Kontextdominanz – Wenn technische Aspekte durch den Kontext überschattet sind, in den das Problemsystem eingebettet ist, dann ist der Erfolg des Systems nur dann möglich, wenn die fachlichen Probleme gleichzeitig gelöst worden sind.

- Unsicherheit – Wenn es keinen „optimalen“ Weg gibt oder dieser nicht bestimmt werden kann.
- Mehrdeutige oder widersprüchliche Erwartungen und Ziele – Da die Fähigkeit Erfolg oder Zielerreichung messbar zu machen eine der Voraussetzungen in allen traditionellen Methoden ist, können diese nur sehr schwer bei Widersprüchen und Mehrdeutigkeiten angewandt werden.⁹⁶
- Exzessive Komplexität – Traditionelle Methoden versuchen nicht die Komplexität als solche hinzunehmen, sondern wollen sie unter Kontrolle bringen oder reduzieren.

Die Folge dieser Defizite ist, dass man bei der Entwicklung großer Systeme (s. auch Abschn. 6.9) folgende Punkte stets berücksichtigen sollte:

- Die initiale Problemdefinition ist inkorrekt und sollte stets angezweifelt werden.
- Der Aufbau der Fähigkeit eines Systems sich zu verändern ist wichtiger als der Ersteinsatz, das heißt die Priorität liegt zu Beginn auf der Flexibilität eines Systems nicht auf der fachlichen Vollständigkeit.
- Zuerst muss der fachliche Kontext in den Vordergrund gerückt werden und die technischen Lösungen sollten in den Hintergrund treten.
- Die Effektivität eines Systems bestimmt sich hauptsächlich durch die „Weltsicht“ des Systems.

2.8.3 *Bricolage*

Ein besonderes Vorgehensmodell, welches beim Aufbau von Systemen aus Services gut funktioniert, ist die Bricolage⁹⁷. Unter der Bezeichnung Bricolage versteht man die Nutzung von vorhandenem Material, um daraus durch neue Verknüpfungen ein neues System zu schaffen. Der Bricoleur schafft kontinuierlich Neues aus dem Vorhandenen. Diese Vorgehensweise ist ideal in einer Umgebung, in der schon eine gewisse Menge an Services vorhanden ist, aus diesen werden neue gewonnen, ohne dass die bestehenden von Grund auf neu implementiert werden. Eines der Kernelemente der Bricolage ist, neben der Zusammensetzung von Neuem aus Bestehendem, die nichtvorhergesehene Verwendung des vorhandenen Materials in einer neuen Art und Weise. Insofern löst der Bricoleur bestehende Services aus ihrem Nutzungskontext und fügt sie in neue Kontexte ein. Eine solche Neuverwendung ist jedoch nur dann möglich, wenn der Service sich auf neue Kontexte einstellen kann und in ihnen weiter funktioniert. Bricolage ist eine explizite Bottom-Up-Vorgehensweise.

Die übliche Tätigkeit des Bricoleurs ist es, zunächst praktische Experimente mit den Services durchzuführen und das Ergebnis dann durch Überlegung zusätzlich zu

⁹⁶ In ihrem täglichen Leben haben die meisten Menschen gelernt, ohne größere Schwierigkeiten mit einigen Dichotomien zu leben.

⁹⁷ Aus dem Französischen, bedeutet in etwa Bastelei und wurde vom Soziologen *Lévi-Strauss* für das Vorgehen nach dem Motto: „Nehmen und verknüpfen, was da ist...“ als Bezeichnung eingeführt.

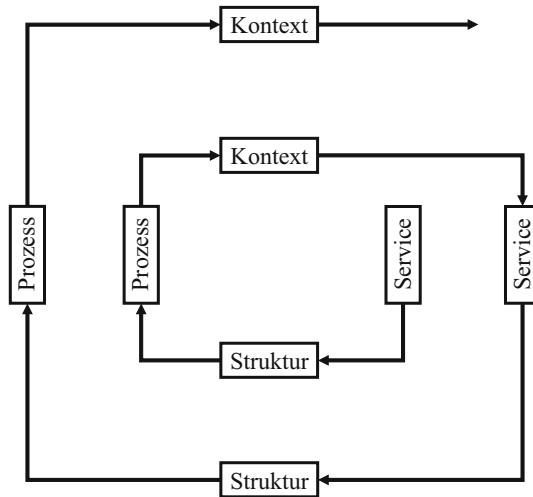


Abb. 2.27 Der Iterationsprozess der Bricolage

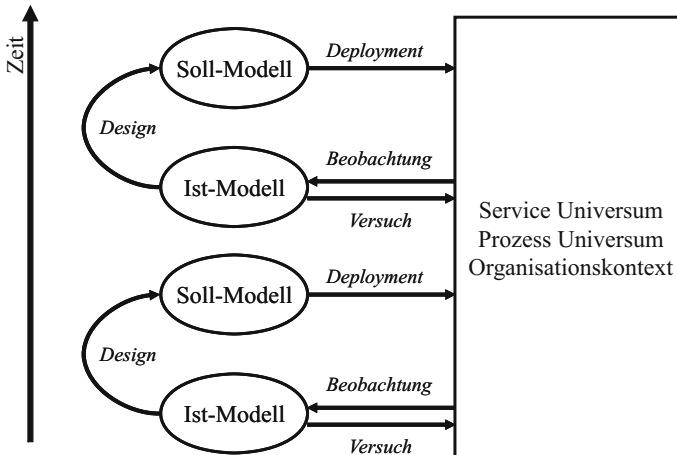


Abb. 2.28 Die Wechselwirkung der Bricolage mit der Umwelt

verfeinern und an neue Gegebenheiten anzupassen. Dabei wird dem Ergebnis eine gewisse Wahrscheinlichkeit gestattet, das heißt es steht nicht ab initio fest, ob überhaupt etwas Sinnvolles aus der Bricolage entsteht. Umgekehrt betrachtet gibt es aber auch eine Wahrscheinlichkeit, dass etwas völlig Neues durch die Bricolage hervorgerufen wird. Der Bricoleur benutzt nur das, was ihm zur Verfügung steht, da er nichts anderes hat. Insofern ist Bricolage eine Neuverknüpfung bekannter Services, um daraus neue Nutzung oder neue Services zu kreieren.

Aus systemtheoretischer Sicht ist Bricolage der Versuch, durch Beobachtung von Veränderungen im System (hervorgerufen durch den Bricoleur) die spezifische

Emergenz des Systems zu erkennen, zu verstehen und zu beschleunigen. Typisch für einen Bricolageprozess ist ein iteratives Vorgehen (s. Abb. 2.27), so dass aus Services neue Strukturen und Prozesse aufgebaut werden, welche wiederum neue Kontexte und Services schaffen. Genau wie der Bricoleur ist auch der Anwender einem endlichen Universum aus Services ausgesetzt, aber im Gegensatz zum Designer, welcher seine Umgebung aktiv abändern kann, fehlen dem Anwender die Mittel und Wege, die ihm zur Verfügung stehenden Services zu verändern, besonders in pervasiven Systemen. Der Anwender kann durch neue Arrangements und durch semantische Reinterpretation bestehender Services auch eine Bricolage an Systemen vornehmen und damit neue Nutzungen erschließen. Empirische Studien zeigen, dass in großen Systemen die Mehrheit der vorgenommenen Anpassungen auf Anwenderinitiativen zurückgeht. Bricolage ist oft das Ergebnis menschlicher Kreativität, Menschen nutzen Systeme anders, als ursprünglich gedacht oder antizipiert wurde.

2.8.4 Taxonomien und Ontologien

Taxonomien sind Klassifizierungsschemata, um Services einzuteilen. Diese werden durch die Provider oder Broker erstellt und mit Informationen über die Services gefüllt. Die so entstehenden Taxonomien dienen als Bindeglied zwischen Menschen und den beteiligten IT-Systemen. Die Taxonomien sind hierarchisch organisiert, da sie die großen Komplexitäten des Geschäftslebens abbilden, dabei bilden sich meist mehrere Taxonomien parallel aus. Idealerweise sind diese orthogonal zueinander. Die unterschiedlichen Taxonomien können nach den verschiedenen Aspekten des Verhaltens klassifiziert sein, mit der Folge, dass die Services sich mehrfach an unterschiedlichen Stellen registrieren lassen müssen, um ein hohes Maß an Aufrufbarkeit zu erreichen. Zwar wird auf Dauer eine gewisse Konvergenz der Taxonomien entstehen, aber da sich die Services selbst relativ rasch ändern dürften, existiert in der Taxonomie und damit in der Registry eine permanente Fluktuation. Taxonomien sind nur dann sinnvoll verwendbar, wenn sie von Menschen verwaltet und aufgesetzt werden, um den semantischen Kontext der Services reflektieren zu können. Jede entstehende Kategorie muss die Semantik ihrer enthaltenen Services definieren und jeder Service der Kategorie muss dieselbe Semantik implementieren. Eine solche semantische Beschreibung muss für einen Menschen verständlich und gleichzeitig für einen Parser syntaktisch interpretierbar sein. Die einzelnen Kategorien müssen ein einfaches oder multiples Vererbungsschema besitzen, welches wiederum ein Spiegelbild der Geschäftswelt ist. Die darin enthaltenen Services gehören in der Regel zu mehreren Kategorien. Bei der Ausführung von Services kann eine Reihe von Problemen auftreten, oft ist der „Aufrufer“ selbst ein Service, welcher seinerseits eine Menge an syntaktischen und semantischen Annahmen über den aufzurufenden Service macht. Umgekehrt machen selbstverständlich die aufgerufenen Services jede Menge von Annahmen über den Aufrufer. Zwischen dem Aufrufer und den Aufgerufenen kann es im Rahmen der Kommunikation zu Diffe-

renzen kommen.⁹⁸ Die entstehenden Differenzen lassen sich als Konflikte klassifizieren:

- Technischer Konflikt – Heterogene Hardware und Betriebssysteme.
- Syntaktischer Konflikt – Unterschiedliche Repräsentation von Daten oder Funktionen, abweichende Protokolle.
- Struktureller Konflikt – Heterogene Modellrepräsentationen.
- Semantischer Konflikt – Unterschiedliche Bedeutungen derselben Ausdrücke.

Für die Lösung von technischen und syntaktischen Konflikten existiert eine lange Tradition in der IT, speziell mit dem Einsatz von hardware- und applikationsneutralen Protokollen lassen sich diese Konflikte in den Griff bekommen. Im Gegensatz dazu sind die strukturellen und semantischen Konflikte viel schwieriger. Durch die Mechanismen der öffentlichen und standardisierten Übertragungs- und Ausführungsprotokolle, welche die Konflikte auf technischer und syntaktischer Ebene gut unter Kontrolle bringen, treten die Probleme auf struktureller und semantischer Ebene stärker in den Vordergrund. Ist beides gelöst, so spricht man von semantischer Interoperabilität. Damit eine semantische Interoperabilität in einer heterogenen Umgebung⁹⁹ erreicht werden kann, muss die Bedeutung systemübergreifend ähnlich sein. Die möglichen semantischen Konflikte lassen sich näher klassifizieren:

- Namensraumkonflikte entstehen entweder in Form von Synonymen oder Homonymen.
- Domänenkonflikte entstehen, wenn verschiedene Referenzsysteme für die Werte benutzt werden.
- Strukturkonflikte werden erzeugt, wenn unterschiedliche Darstellungsformen unterschiedliche Daten nutzen, um das gleiche Konzept zu beschreiben.
- Metadatenkonflikte entstehen, wenn die Konzepte in dem einen System als ein bestimmter Typ repräsentiert werden und in dem anderen System völlig unterschiedlich erscheinen.

Ontologien werden genutzt, um die Semantik einer Informationsquelle zu beschreiben und damit den Inhalt explizit zu machen, mit der Folge, dass semantische Äquivalenzen zwischen unterschiedlichen Konzepten entdeckt werden können. Eine Ontologie¹⁰⁰ definiert sich zu:

⁹⁸ Diese Differenzen sind teilweise die Folge der losen Koppelung. Bei einer starren Koppelung werden viele der technischen und syntaktischen Konflikte schon durch den Compiler oder Linker entdeckt und können während der Designphase behoben werden.

⁹⁹ Heterogen im Sinne der Semantik, nicht in technischem Sinne, so zum Beispiel unterschiedliche Domänen, Sprachen oder Kulturen.

¹⁰⁰ Aus dem Bestreben der Philosophen, die Welt zu ordnen, entstand der Begriff der Ontologie. Aus dem Griechischen *το ον* (das Seiende) und *λογία* (Lehre). Das allgemeinste Seiende wird als Entität bezeichnet. In der Philosophie unterscheidet man, historisch bedingt, eine Reihe von Ontologien:

- Materialismus – Materie ist das Fundament alles Seienden. Es gibt überhaupt kein Ideelles, nichts Unkörperliches (Karl Marx).

Eine Ontologie ist eine explizite Spezifikation einer Konzeptionalisierung einer abstrakten und vereinfachten Weltsicht, welche die Elemente eines bestimmten Bereichs und deren Relationen beschreibt. Das Vokabular einer Ontologie ist die Menge aller Bezeichner, die den Objekten und Relationen zugeordnet sind, die sich auf Basis der Ontologie beschreiben lassen.

Alle Objekte und die Relationen zwischen ihnen, die sich auf Basis einer Ontologie beschreiben lassen, finden ihre Entsprechung im Vokabular in Form von Bezeichnern. Dies ermöglicht die Beschreibung einer Ontologie durch die Definition einer Menge von Termen. Die Definitionen ordnen den Bezeichnern der Elemente umgangssprachliche Texte zu, welche die Bedeutung dieser Namen beschreiben, und sie bilden Axiome, welche die Interpretation solcher Terme eingrenzen. Dazu können zum Beispiel Definitions- und Wertebereiche, Einordnung in Hierarchien, Vererbung von Klasseneigenschaften an Instanzen und so weiter gehören. Die Beschreibung einer Ontologie erfolgt unabhängig von der internen Wissensrepräsentation und Struktur konkreter Systeme. Für Systeme, Programme und auch menschliche Benutzer sind Ontologien ein Mittel, um sich über die Bedeutungen von Objekten und Relationen zu einigen, auf deren Basis sie miteinander interagieren. Dabei ist es weder notwendig, dass die beteiligten Consumer ihre eigene Weltsicht auf dieselbe Weise repräsentieren, noch dass sich diese in Umfang und Mächtigkeit entsprechen. Je nach Art und Umfang der Interaktion kann eine Verpflichtung der Beteiligten zu einer begrenzten gemeinsamen Ontologie genügen.

Eine Ontologie beginnt immer mit einer speziellen Sicht auf eine Domäne des betreffenden Wissensgebiets.¹⁰¹ Diese Sichten werden typischerweise von vielen Anwendern in der jeweiligen Domäne geteilt. Eine Ontologie enthält neben den grundlegenden Theorien und Axiomen der Domäne auch eine Angabe über den Sprachgebrauch und „wie Dinge funktionieren“. Eine Ontologie ist eine Formalisierung dieser domänenspezifischen Sichten. Jede Ontologie identifiziert die interessanten Objekte einer Domäne und ordnet ihnen Symbole¹⁰², die diese deklarativ repräsentieren, zu. Diese Menge an Objekten wird innerhalb der Ontologie zum Universum des Diskurses, in dem sie die Begrifflichkeit der Kommunikation bestimmen.

Die Ontologien dienen aber auch dazu, bereits bestehende Wissensbestände zusammenzufügen. Der Unterschied einer Ontologie gegenüber einer Taxonomie ist,

- Idealismus – Unkörperliche und gestaltlose, außerhalb von Raum und Zeit bestehende, Elemente bilden die Grundlage alles Existierenden (Platon).
- Mischformen – Diese gehen davon aus, dass es faktisch unmöglich ist, ideelle und materielle Seinskomponenten zu trennen.

¹⁰¹ Typische Beispiele für solche Ontologien sind:

- Naturwissenschaftliche Ontologien: Elementarteilchen als Teil der Physik und Motorik des menschlichen Körpers als Teil der Biomechanik.
- Informatik: Definition und Manipulation von Daten als Teil der Informatik, Softwarearchitekturen, Algorithmen und Protokolle.
- Wirtschaft: Geschäftsstrategien, Finanzen, Buchhaltung, Marketing und Konzernbilanzierung.

¹⁰² In der Physik der Elementarteilchen e^- für Elektronen, e^+ für Positronen und γ für Photonen.

dass die Ontologie ein Netzwerk von Informationen mit logischen Relationen darstellt, während die Taxonomie eine einfache Hierarchie bildet. Die Ontologien sind in einem System, welches aus einer Vielzahl von Services besteht, unumgänglich, da eine Auswahl ohne semantisches Wissen und das Wissen über eventuelle Relationen der Begriffe und Services zueinander ohne eine Ontologie faktisch nicht möglich ist. Ontologien sind nicht einfach statisch oder entstehen spontan, sondern sind das Ergebnis einer lang anhaltenden Wissensakquisition und verändern sich dabei permanent.

2.8.5 OWL-S und WSMO

Die **Web Ontology Language for Web Services (OWL-S)** ist hervorgegangen aus DAML-S^{103,104} und hat als Zielstellung die semantische Beschreibung von Webservices, um die Lücke zwischen der herkömmlichen Infrastruktur der Webservices und dem semantischen Web zu schließen. OWL-S ist der bekannteste Vertreter für die Umsetzung semantischer Webservices, Anwendungsbereiche für OWL-S sind automatisierte Such- und Auswahlmöglichkeiten, Aufruf, Zusammenarbeit, Komposition und Laufzeitüberwachung von Webservices. OWL-S ist keine WSDL-Erweiterung, sondern eine komplementäre Entwicklung (s. Abb. 2.29). OWL-S ist durch eine Metaontologie strukturiert in:

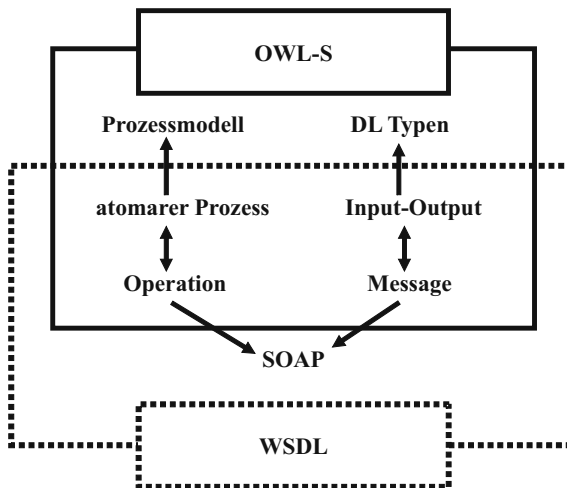


Abb. 2.29 Das WSDL-Grounding des OWL-S

¹⁰³ DARPA Agent Markup Language for Services.

¹⁰⁴ Defense Advanced Research Projects Agency (DARPA).

Tabelle 2.15 Einige OWL-Konstrukte und ihre mathematischen Entsprechungen

OWL-S	Mathematik	Bedeutung
IntersectionOf	$A \cap B$	Schnittmenge
UnionOf	$A \cup B$	Vereinigungsmenge
ComplementOf	$\neg A$	Negation
OneOf	$x \in \{a, b, c\}$	Element von
AllValuesFrom	$\forall R. \mathcal{C}$	Hat alle Werte aus $\mathcal{C}$
SomeValueFrom	$\exists R. \mathcal{C}$	Hat Wert aus Menge $\mathcal{C}$
MinCardinality	$\geq_n$	Mindestkardinalität
MaxCardinality	$\leq_n$	Maximalkardinalität
Cardinality	$\equiv_n$	Exakte Kardinalität
SubClassOf	$A \sqsubseteq B$	Konzept A subsumiert Konzept B
EquivalentClass	$A \equiv B$	Konzept A äquivalent Konzept B
SubPropertyOf	$a \sqsubseteq b$	Eigenschaft a subsumiert b
SameAs	$\{a\} \equiv \{b\}$	a identisch zu b
InverseOf	$a \equiv b^{-1}$	Eigenschaft a invers zu b
TransitiveProperty	$(x, y) \in A \wedge (y, z) \in A \rightarrow (x, z) \in A$	Transitivität
SymmetricProperty	$(x, y) \in A \rightarrow (y, x) \in A$	Symmetrie

- *ServiceProfile*,
- *ServiceModel*,
- *ServiceGrounding*.

Webservices werden in WSDL als eine Menge von *ports* definiert, die im Wesentlichen aus Netzwerkadressen mit zugehörigen Protokollen und Datentypspezifikationen bestehen. Zu jedem *port* gehört ein *porttype*, der den Messageaustausch beschreibt (*operations*), an dem der *port* beteiligt ist. OWL-S hingegen unterstützt eine erweiterte semantische Beschreibung von Webservices über Kommunikationsinterfaces hinaus. Die Grounding-Ontologie von OWL-S unterstützt die gemeinsame Nutzung von OWL-S und WSDL. Unter Grounding wird die Abbildung zwischen der Ontologie als semantische Beschreibung und der WSDL-Definition eines Service als syntaktische Beschreibung verstanden. Prozesse werden dabei als abstrakte Servicespezifikationen betrachtet und enthalten keine Aussagen zu den Details der Kommunikation. Diese werden nach wie vor durch WSDL beschrieben. Außerdem wird die Grounding-Ontologie genutzt, um die Abbildung von bestimmten OWL-S-Ontologie-Elementen auf gewählte Elemente der WSDL-Spezifikation vorzunehmen. Durch OWL-S lässt sich ein Webservice wesentlich detaillierter beschreiben als mit WSDL. An Hand der zusätzlichen Abbildung auf bestimmte Begriffe aus den Ontologien ist es möglich, nicht nur funktionale Eigenschaften, sondern auch Fähigkeiten von Services semantisch eindeutig zu beschreiben.

Die **Web Service Modeling Ontology (WSMO)** ist neben OWL-S ein anderer Versuch, Ontologien zugänglich zu machen. Das Ziel von WSMO ist es, Verfahren bereitzustellen, mit denen die semantische Servicenutzung automatisiert werden kann. Dazu wird eine geeignete Beschreibung von Services in allen Aspekten angestrebt, welche auch die Integrationsproblematik berücksichtigt. Das konzeptionelle Modell von WSMO ist ein Metamodell, in welchem die wichtigsten Komponenten zur Automatisierung der Servicenutzung festgehalten sind:

- Alle WSMO-Beschreibungen sind ontologiebasiert. Sie stützen sich auf das gemeinsame Verständnis der Anwender und die formale Semantik der zugrunde liegenden Sprache.
- WSMO-Beschreibungen werden strikt voneinander entkoppelt und werden unabhängig von ihrer Verwendung erstellt.
- Trotz der Heterogenität bei der Beschreibung von Ressourcen soll eine Kommunikation zwischen allen Teilnehmern möglich sein.
- Angebotene und benötigte Services unterscheiden sich grundlegend, was durch getrennte ontologische Rollen zum Ausdruck gebracht wird.
- Vorrangiges Ziel von WSMO ist die Versorgung von Teilnehmern mit Services, welche anschließend verwendet werden, um bestimmte Operationen durchführen zu können.

Das hieraus entwickelte konzeptionelle Modell von WSMO ist als Metamodell spezifiziert. Es basiert auf vier Hauptelementen:

- WSMO-Element – Stellt die abstrakte Oberklasse aller vier Hauptelemente dar. Jedes Hauptelement kann durch einen Satz nichtfunktionaler Attribute beschrieben sein. Jedes Hauptelement kann zu seiner Beschreibung Ontologien importieren.
- Ontologien – Ontologien liefern ein formal spezifiziertes Vokabular, das von allen anderen Komponenten verwendet wird. Die Ontologien der einzelnen Teilnehmer können unabhängig voneinander entwickelt und durch Import gegenseitig verwendet werden. Ontologien in WSMO bestehen aus Konzepten, Attributen, Funktionen, Relationen, Instanzen sowie Axiomen in Form logischer Ausdrücke.
- Ziele – WSMO sieht anders als OWL-S eine explizite Trennung der Beschreibung von angebotenen und benötigten Services vor. Für benötigte Services wird ein zielorientierter Ansatz verfolgt, bei dem der Wunsch des Consumers als Ziel formuliert wird, unabhängig davon, wie dieses durch einen konkreten Service ausgeführt werden könnte. Aufgabe der Ausführungsumgebung ist es dann, dieses Ziel durch Nutzung eines oder mehrerer Services zu erfüllen.
- Mediatoren – Bei der Nutzung von Webservices kann es zwischen Komponenten, die interagieren müssen, zu Heterogenität auf Daten-, Protokoll- und Prozessebene kommen, insbesondere dann, wenn die Interaktionen in einer offenen Umgebung stattfinden.

2.9 Sicherheit

Informationen haben für jeden einen Wert, unabhängig davon, wie diese Informationen zur Verfügung stehen. Es gilt diesen Wert zu schützen und zu erhalten. Mit dieser Erhaltung beschäftigt sich die Informationssicherheit, deren Ziel es ist, die Informationen und Prozesse in folgenden Kontexten zu schützen:

- Vertraulichkeit – Informationen sollten nur den dafür autorisierten Personen zugänglich sein.
- Integrität – Die Korrektheit und Vollständigkeit der Informationen und der entsprechenden Verarbeitungsmethoden muss geschützt werden.
- Verfügbarkeit – Die Informationen müssen, wenn sie benötigt werden, auch verfügbar sein.

Der Verlust von Vertraulichkeit, Integrität, Verfügbarkeit, Nachvollziehbarkeit, Authentizität und Verlässlichkeit von Informationen und Services hat große Auswirkungen auf jede Organisation, gleichgültig ob in der Wirtschaft oder Verwaltung. Im privaten Bereich kann ein Bruch der Vertraulichkeit peinliche Momente zur Folge haben. Besonders kritisch wird der Bruch der Vertraulichkeit im Bereich des Gesundheitswesens, der Finanzbehörden, der Justiz, beziehungsweise der Anwälte und der Polizei gesehen. Die Schaffung einer Umgebung für Informationssicherheit in einer finanzierbaren und „unmerklichen“ Weise war schon immer eine Herausforderung für Softwaresysteme. Die Verfügbarkeit angemessener Sicherheitssysteme ist heute wichtiger denn je, da mit der Zugänglichkeit und Mächtigkeit von Software auch deren Sicherheitsrisiko ansteigt. Leider wird Sicherheit immer noch sekundär zur Funktionalität in der Software gesehen, mit dem Ergebnis, dass bei vielen Organisationen Kernprozesse einem hohen Risiko ausgesetzt sind.

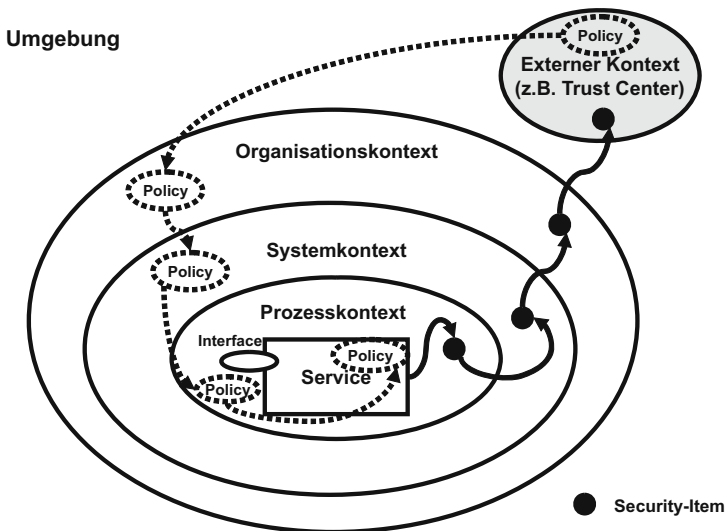


Abb. 2.30 Die Kontexte und Policies für Security aus systemtheoretischer Sicht

Sichere Applikationen entstehen nicht einfach zufällig, sie sind stets das Ergebnis davon, dass der Hersteller der Software sich entschieden hat eine sichere Applikation zu bauen. Insofern entsteht in aller Regel in einer Organisation, die kein Sicherheitsbewusstsein hat, auch keine sichere Software. Traditionell wurde die Sicherheit bei Einzelplatzcomputern oder kleinen Netzwerken durch den physischen

Zugang geregelt. Dies ist innerhalb des Internets oder eines großen vernetzten Systems nicht mehr möglich. Hier sind aufgrund der Verteilung andere Mechanismen gefragt. Selbst in größeren Systemen geht man bezüglich des Zugangs von einer „geschlossenen Welt“ aus. Eine solche Limitierung ist aber in pervasiven oder mobilen Umgebungen oder digitalen Businessökosystemen, kurz gesagt bei allen Formen von digitalen Ökosystemen, nicht mehr aufrechtzuerhalten. Sicherheit ist etwas, was in der Software nicht oder nur sehr schwer nachträglich eingeführt werden kann. Speziell beim Einsatz von Services stellt sich für die Sicherheit eine Reihe von neuen Herausforderungen:

- internetskalierend – Neue Services müssen das Potenzial haben, auf einer Vernetzung betrieben zu werden. Die Verwaltungs- und Sicherheitsmechanismen müssen dies explizit berücksichtigen inklusive der Nutzung öffentlicher Netze und Services als funktionale Erweiterung eigener Services.¹⁰⁵
- organisationsübergreifend – Auf Grund der Internetskala können sich Sicherheitsmechanismen nicht mehr auf die eigene Organisation beschränken. Services werden auch organisationsübergreifend genutzt und dies muss unterstützt werden.
- autonom – Da es organisationsübergreifend keine zentrale Autorität gibt, muss die Sicherheit auch autonom gewährleistet werden. Man muss in der Lage sein, disjunkte Sicherheitszonen miteinander zu verbinden.
- Offenheit – Jeder Service muss in diversen Kontexten agieren können, mit der Folge, dass er sich auf unterschiedliche Policies und Sicherheitsmechanismen einstellen können muss. Dies setzt eine adaptive Architektur (s. Anhang B) voraus.
- evolutionär – Das System bestehend aus Services sowie jede einzelne Serviceimplementierung wird sich auch in Zukunft verändern.

Der Einfachheit halber lassen sich die Mechanismen für die Entwicklung von Sicherheit¹⁰⁶ als Pyramide darstellen (s. Abb. 2.31):

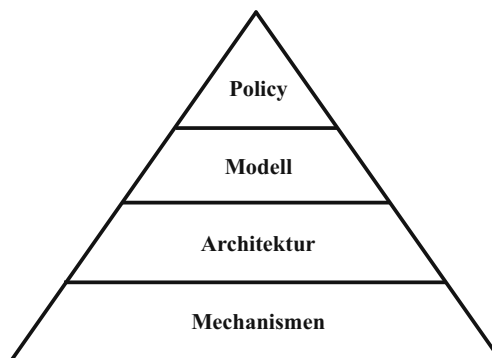


Abb. 2.31 Das Framework für das Securityengineering

¹⁰⁵ Die entstehenden Schwierigkeiten werden in Abschn. 6.9 angesprochen.

¹⁰⁶ Security Engineering.

- Policy – Die Policy formuliert auf abstrakter Ebene die Ziele und Anforderungen der Organisation an die Sicherheit. Sie wird durch die Antizipation von Bedrohungen und zukünftigen Zielen getrieben und sollte die Prinzipien des Risikomanagements¹⁰⁷ beachten.
- Modell – Das Modell zerlegt die Policy in abstrakte Ausdrücke, die analysiert und auf implementierbare Einheiten abgebildet werden können.
- Architektur – Diese beschreibt das Securitydesign in Bezug auf die Komponenten des Gesamtsystems und der Beziehungen untereinander. In einem verteilten System sind dies die Server, Middleware, Datenbanken und so weiter.
- Mechanismen – Hierunter wird eine Reihe von Wegen verstanden, das Securitydesign zu implementieren, dazu zählen Netzwerkprotokolle, Token, Credentials und so weiter.

Die Anforderungen an die Sicherheit machen die ohnehin schwierige Komposition von neuen Services aus einzelnen Services noch komplizierter, da Services von unterschiedlichen Providern zur Verfügung gestellt werden können und diese Provider sich oft nicht gegenseitig „trauen“. Auf der anderen Seite haben Provider sicherzustellen, dass der von ihnen gelieferte Service eine Summe an vorgegebenen Sicherheitsrichtlinien in jeder Form der Zusammenarbeit mit anderen Services, der darunterliegenden Infrastruktur, inklusive Betriebssysteme, erfüllt und zwar unabhängig von dem jeweiligen Consumer des Service. Eine zusätzliche Komplexität wird noch dadurch produziert, dass der Consumer ein Interesse daran hat seine eigenen Daten vor dem Provider oder anderen Consumern zu schützen.

Eine mögliche Lösung für dieses Problem ist es, beim Serviceaufruf die entsprechende Sicherheitsrichtlinie zu übermitteln, sodass sie wie eine Eigenschaft des Service wirkt.

2.9.1 Sichere Übertragung

Jede Form eines Service muss eine Reihe von Sicherheiten bieten. Im Fall von SOAP lässt sich eine solche Sicherheit oberhalb, beziehungsweise unterhalb, des SOAP-Layers einbringen, unterhalb wäre auf Transportebene, so etwa in Form von HTTPS. Allerdings sollten Sicherheitsvorkehrungen nicht alleine auf der Transportebene gültig sein, sondern unabhängig davon auch auf darüberliegenden Ebenen, von denen SOAP eine ist. Für sicheres SOAP spricht:

- End-2-End-Security – Die einzelnen SOAP-Messages können über diverse unsichere Kanäle ans Ziel gelangen. Oft ist es weder dem Absender noch dem Empfänger bekannt, ob die Message über einen unsicheren Kanal übertragen

¹⁰⁷ Ein Risiko besteht aus dem Produkt der Schadenshöhe und der Eintrittswahrscheinlichkeit $r = ps$. Häufig auftretende Risiken mit geringer Schadenshöhe (Erkältung der Mitarbeiter) sind zwar ärgerlich, werden aber nicht aktiv angegangen. Umgekehrt werden sehr seltene Ereignisse mit hohen Schäden (Ausbruch eines Vulkans in Deutschland) auch nicht angegangen.

wird oder nicht. Wenn nur eine der Zwischenstationen nicht vertrauenswürdig ist, ist der gesamte Übertragungsweg nicht vertrauenswürdig.

- Serviceunabhängigkeit – Der SOAP-Layer kann die Verschlüsselung als eigenständigen Service anbieten, daher wird dieser Aspekt der Sicherheit für den fachlichen Service transparent.
- Transportprotokollunabhängigkeit – Die SOAP-Verschlüsselung ist unabhängig vom genutzten Transportprotokoll.
- Persistente Messages – Da Messages nicht nur transportiert, sondern auch gespeichert werden können, ist es sinnvoll, diese persistenten Daten genauso zu schützen wie den Transport.

Durch die Verschlüsselung des XML lässt sich der XML-Teil einer Message mit Standardverschlüsselungstechniken chiffrieren und durch den Empfänger wieder dechiffrieren. Einziges Problem dieses Verfahrens ist der notwendige Schlüsselaustausch, dieser kann jedoch analog zum Verfahren des HTTPS über ein bekanntes Public/Private-Key-Verfahren implementiert werden.

Im Fall der XML-Encryption existiert ein spezieller *EncryptedData*-Tag, welcher direkt ein Teil der XML-Message ist:

```
<SecretMessageInfo xmlns="http://yourscheme.org/cryptv3">
  <Agent> James Bond </Agent>
  <SecretIdentity>
    <xenc:EncryptedData
      xmlns="http://www.w3.org/2000/10/xmlenc"
      Id="encData1"
      DecryptionInfoURL="#decInfo1"
      IV="afh834qf=">ansjhfkahsdj kf5623rJnWHUWDnjqkehwr fu=
    </xenc:EncryptedData>
  </SecretIdentity>
</SecretMessageInfo>
```

Durch die explizite Trennung der Struktur und des Inhalts in XML ist es sinnvoll den Inhalt zu verschlüsseln und die Struktur unangetastet zu lassen. Der verschlüsselte Teil kann wiederum ein XML-Dokument sein. Nachteil dieser Form der Verschlüsselung ist, dass hierbei kein XML-Parsing des verschlüsselten Teils mehr möglich ist. Aus reiner Sicherheitssicht sollten auch die „Tags“ mit verschlüsselt werden, da sonst aus der Struktur und den relativ kurzen Chiffretexten recht gut auf den Originaltext der Message geschlossen werden kann. Allerdings führt die Verschlüsselung der Struktur wieder zu dem XML-Parsing-Problem.

Das SOAP-DSIG beschreibt eine Möglichkeit, SOAP-Messages mit Hilfe von digitalen Signaturen zu unterzeichnen. Mit Hilfe von digitalen Signaturen können der Verfasser und die Integrität einer Message verifiziert werden. Nicht möglich ist jedoch die Verifikation des Senders.¹⁰⁸ Die XML Signature Specification be-

¹⁰⁸ Digitale Signaturen schützen daher nicht vor sogenannten Replay-Attacken, bei denen Messages abgefangen und zu einem späteren Zeitpunkt erneut versendet werden um sich unberechtigt Zugang zu einer Ressource zu verschaffen.

schreibt, wie SOAP-Messages digital signiert werden können. Zur Signierung einer Message berechnet der Sender aus den zu sendenden Informationen und einem geheimen Schlüssel eine Signatur, die er zusammen mit den zu sendenden Nutzdaten in der SOAP-Message an den Empfänger sendet. Anhand des öffentlichen Schlüssels des Senders, der empfangenen Signatur, und den empfangenen Nutzdaten kann der Empfänger Integrität und Verfasser der Message überprüfen. Zum Unterzeichnen von SOAP-Messages mit einer digitalen Signatur ist der SOAP-Header Eintrag *<SOAP-SEC:Signature>* definiert worden. Unterzeichnet werden können beliebige Teile einer SOAP-Message.

Auch die Non-Repudiation-Anforderung, der zweifelsfreie Nachweis, dass der Sender die Message wirklich verschickt hat und der Empfänger die gleiche Message erhalten hat, lässt sich mit Hilfe von digitalen Signaturen erreichen. Dazu ist es notwendig, dass die Message die Identität des Senders, des Empfängers und eine Kennzeichnung, wie zum Beispiel einen Zeitstempel oder eine Sequenznummer zur Vermeidung von Replay-Attacken enthält. Durch die digitale Signatur des Zeitstempels und des Inhaltes der Message kann der Empfänger nachweisen, dass er genau diese Message vom Sender erhalten hat.

Falls es erforderlich ist, dass der Sender den Versand der Message nachweisen muss, ist es erforderlich, dass der Empfang der Message bestätigt wird. Der digital signierte Zeitstempel verhindert neben den Replay-Attacken auch, dass der Empfänger den mehrmaligen Erhalt der Message vortäuschen kann. Wenn in der Message die Identität des Empfängers angegeben wird, lässt sich vermeiden, dass jemand anderes als der beabsichtigte Empfänger den Empfang der Message für sich reklamieren kann. Bei asymmetrischen Verschlüsselungsverfahren benötigt man den öffentlichen Schlüssel des Kommunikationspartners, an den eine verschlüsselte Message gesendet werden soll. Auch bei der Prüfung einer Signatur braucht man den öffentlichen Schlüssel des Kommunikationspartners, der die Message versendet hat.

Die Security Assertion Markup Language (SAML) handhabt den Austausch der Anfragen und Antworten zur Authentifizierung und Autorisierung durch definierte XML Datensätze. SAML selbst stellt keine Funktionalität zur Autorisierung oder Authentifizierung zur Verfügung. Durch SAML lässt sich lediglich zusichern, dass dies bereits an anderer Stelle erfolgt ist. Für die Unterstützung der Authentifizierung existiert ein Single Sign-On-Konzept, für die Vereinheitlichung der Autorisierung besteht ein Service.

Beim Single Sign-On authentifiziert sich der Benutzer einmal gegenüber einem Service und kann dann ohne erneute Authentifizierung auch Ressourcen anderer Services oder Websites nutzen, da die anfangs erfolgte Authentifizierung per SAML weitergereicht werden kann. Voraussetzung ist, dass alle Services SAML explizit unterstützen und der ursprünglichen Authentifizierungsstelle auch vertrauen.

Eine solche Authentifizierungszusicherung kann durch entsprechende Erweiterungen der SOAP-Spezifikation des Headers einer SOAP-Message dazu verwendet werden, SAML-Zusicherungen über die Identität des Absenders der SOAP Message zu liefern. Die digitale Signatur von Zusicherungen mit Hilfe der XML Signature Specification verhindert, dass Zusicherungen unberechtigt abgeändert werden können:

```

<saml:Assertion MajorVersion="1" MinorVersion="0"
AssertionID="127.19.7.7.7007"
Issuer="MI6" IssueInstant="2007-11-03T10:02:00Z">
<saml:Conditions
NotBefore="2007-11-03T10:00:00Z"
NotAfter="2007-11-03T10:05:00Z"/>
<saml:AuthenticationStatement
AuthenticationMethod="password"
AuthenticationInstant="2007-11-03T10:02:00Z">
<saml:Subject>
<saml:NameIdentifier
SecurityDomain="mi6.gov.uk"
Name="James Bond" />
</saml:Subject>
</saml:AuthenticationStatement>
</saml:Assertion>

```

Beim Autorisierungsservice geht man davon aus, dass ein Consumer auf eine Resource oder einen Service zuzugreifen will. Die Sicherheitskontrolle (Gatekeeper) für diesen Service überprüft bei einem Autorisierungsservice (Policy Decision Point), ob der Benutzer autorisiert ist, den entsprechenden Service auch nutzen zu dürfen.

Wie auch die SAML Authentifizierungszusicherung kann die SAML Autorisierungszusicherung nach erfolgter Prüfung dem Header einer SOAP Message hinzugefügt werden:

```

<saml:Assertion ...>
<saml:Conditions .../>
<saml:AuthorizationStatement
Decision="Permit"
Resource="http://cia.gov/rpt_12345.htm">
<saml:Subject>
<saml:NameIdentifier
SecurityDomain="cia.gov"
Name="James Bond" />
</saml:Subject>
</saml:AuthorizationStatement>
</saml:Assertion>

```

2.9.2 WS-Security Suite

Die ursprünglichen Webservices besaßen keinerlei Unterstützung in punkto Sicherheit, aber es wurde rasch erkannt, dass eine explizite Unterstützung von Sicherheitsaspekten wichtig für einen kommerziellen Einsatz dieser Form von Services ist. Aus

diesem Grunde wurde von *IBM* und *Microsoft* zusammen die **WS-Security-Suite** (WS³, s. Abb. 2.32) geschaffen.

Die Basisidee hinter WS³ ist es, keine neuen Sicherheitstechnologien einzuführen, sondern auf bestehende zurückzugreifen. Damit wird gleichzeitig ein Sicherheitsmodell spezifiziert, welches es erlaubt, diverse Implementierungen simultan zu betreiben. Das Webservicesicherheitsmodell definiert Anforderungen an die Webservicesicherheit. Möglichkeiten zur Implementierung der definierten Sicherheitsziele bietet die WS³ mit seinen Frameworks. Diese bieten Sprachkonstrukte mit denen unterschiedliche Sicherheitsfunktionen realisiert werden können. Die WS³ ist modular aufgebaut. Sie beinhaltet ein Messagessicherheitsmodell (WS-Security), das mit seinem Framework die Basis der Spezifikationen bildet. Dieses wird erweitert mit einem Webservicesicherheitsrichtlinien Framework (WS-Policy), einem Vertrauensmodell (WS-Trust) und einem Datenschutzmodell (WS-Privacy). Weitere Standards bieten Funktionalitäten für den Aufbau anderer Sicherheitsmodelle. Durch Kombination der unterschiedlichen Komponenten kann eine komplette Sicherheitslösung für Webservices erreicht werden.

Das Protokoll WS-Security definiert die Erweiterungen zum SOAP-Messageaustausch, um Integrität und Vertraulichkeit der Messages zu gewährleisten und Authentifikation zu ermöglichen. Dazu definiert WS-Security Sicherheitstoken, die in SOAP-Messages integriert werden und beschreibt, wie die XML-Sicherheit Spezifikation (XML-Verschlüsselung und XML-Signatur) zur Verschlüsselung und zum Signieren verschiedener Teile von SOAP-Messages verwendet werden kann. Sind Bedingungen und Sicherheitsanforderungen in der Interaktion zwischen einem Consumer und einem Webservice erforderlich, können diese durch Sicherheitsrichtlinien beschrieben werden.

Bedingungen und Anforderungen werden durch den Provider mittels Policies festgelegt. Dazu stellt das WS-Policy Protokoll das erforderliche Framework. Speziell zur Festlegung von Sicherheitsrichtlinien wird der WS-Policy Standard verwendet. Dieser definiert Prinzipien zur Spezifizierung von Sicherheitsanforderungen. Die Verteilung von Richtlinien und deren Bezug zur Anwendung ist nicht Bestandteil dieser Standards. In der Spezifikation Webservices Metadata Exchange wird beschrieben, wie eine Richtlinie über SOAP-Messages oder eingeschlossen in XML- und WSDL-Dokumenten zugänglich gemacht werden kann.

Der WS-Policy-Standard definiert ein Framework, mit dem Webservices Sicherheitsbedingungen und Anforderungen zur Nutzung ihrer Funktionalität durch Consumer definieren können. Diese Anforderungen und Bedingungen werden in Policy Assertions formuliert. Die Richtlinien können Eigenschaften aus WS-Security, WS-Trust oder WS-SecureConversation festlegen, die erfüllt sein müssen, um die angebotenen Services zu nutzen. Möglich sind auch Angaben von Alternativen zur optionalen Anwendung, um bestimmte Sicherheitsziele zu erreichen. Flexibilität in Bezug auf unterschiedliche Tokentypen, kryptographische Algorithmen und Mechanismen ist Ziel des Standards. Für die Richtlinienerstellung bietet der Standard Sprachkonstrukte, mit denen Teile eines Dokumentes bestimmt werden können, die bestimmten Richtlinien entsprechen müssen. Das Protokoll WS-Trust ist eine Erweiterung der WS-Security-Spezifikation. WS-Security bietet ein Framework für

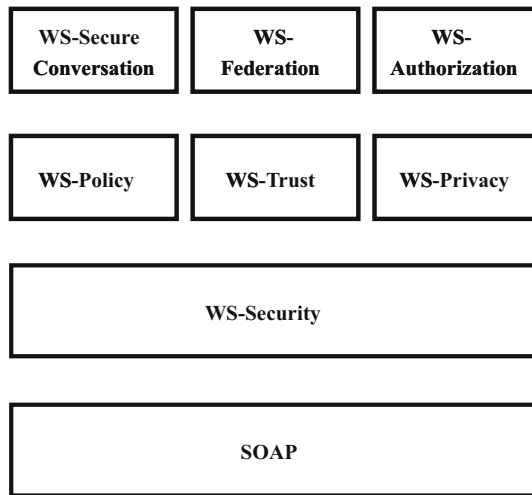


Abb. 2.32 Die WS-Security Suite

das Sichern von SOAP-Messages, basierend auf XML-Signatur, XML-Verschlüsselung und dem Sicherheitstokenkonzept. Mit Hilfe dieser Komponenten können verschiedene Teile eines SOAP-Envelope für unterschiedliche Vermittler oder Consumer gesichert werden mit dem Ziel, Authentizität, Vertraulichkeit und Integrität zu gewährleisten. Als Token kommen dabei X509-Zertifikate, Kerberos- und Benutzername-tickets zum Einsatz. WS-Security spezifiziert die Einbindung dieser Komponenten in SOAP-Messages und das Verweisen von signierten oder verschlüsselten Daten auf diese Token, legt aber nicht fest, wie diese Sicherheitstoken angefordert und ausgetauscht werden. Diese Aufgabe kann entweder programmtechnisch oder durch den Aufruf eines mit WS-Trust festgelegten Service erfolgen, der Token vergibt. WS-Trust bietet ein Framework für die Anfrage und Ausstellung von Sicherheitstoken und die Vermittlung von vertraulichen Verbindungen. Zentraler Service im WS-Trust Modell ist der Sicherheitstoken-Service (**Security Token Service, STS**), der für die Ausgabe, Prüfung und Erneuerung von Sicherheitstoken aller Art verantwortlich ist. Dafür definiert der STS Bindings, mit denen die entsprechenden Protokollnachrichten und die Eigenschaften, die ein Token und ein Consumer besitzen sollen, beschrieben werden. Alle Bindings verwenden ein Token Request Framework, das sich auf zwei Messagetypen beschränkt:

- *wst:RequestSecurityToken* für eine Anfrage an den STS,
- *wst:RequestSecurityTokenResponse* für die Antwort vom STS.

Entspricht ein Request nicht dem Binding, so sendet der STS einen entsprechenden Fehlercode (*wst:InvalidRequest*). Durch die Verwendung des WS-Adressing Framework ist es dem STS möglich, zwischen unterschiedlichen Bindings zu unterscheiden. Sicherheitstoken werden durch das Versenden einer *wst:RequestSecurityToken* Message bei dem STS angefordert. Entsprechend der Spezifikation sind drei Operationen möglich:

- Anfrage und Erteilung eines neuen Token,
- Erneuerung eines schon bestehenden Token,
- Überprüfung eines Token.

Nach der Überprüfung sendet der STS das angeforderte Token zusammen mit einem Proof-of-Possession-Token an den Anforderer zurück. Das Token kann nun verwendet und mittels WS-Security-Elementen referenziert werden. Mit Hilfe des Proof-of-Possession-Token kann der Empfänger beweisen, dass er das Sicherheitstoken verwenden kann. Es enthält einen, mit dem aus dem Zertifikat entnommenen öffentlichen Schlüssel chiffrierten, geheimen Wert. WS-Security und WS-Trust bieten Mechanismen für das Sichern einer einzelnen Message und für den Umgang mit Sicherheitstoken zur Authentifizierung. Webservices Secure Conversation definiert Erweiterungen, basierend auf WS-Security und WS-Trust, um eine sichere Kommunikation über mehrere Messages zu unterstützen. Dazu wird ein Sicherheitskontext zwischen den kommunizierenden Parteien gebildet, der mit Sicherheitskontexttoken aufgebaut wird. Diese enthalten ein von den Parteien gemeinsam genutztes Geheimnis, das die Basis für den Sicherheitskontext bildet. WS-SecureConversation definiert Mechanismen für die Bereitstellung und den Austausch von Sicherheitskontexttoken. Aus den bereitgestellten Sicherheitskontexttoken werden die Schlüssel zur sicheren Kommunikation abgeleitet. Der so entstandene Sicherheitskontext zwischen den kommunizierenden Parteien ist definiert als neuer WS-Security Tokentyp in Verbindung mit einem WS-Trust Binding.

Ein Sicherheitskontexttoken repräsentiert einen von zwei kommunizierenden Parteien gemeinsam genutzten Sicherheitskontext. Dieser wird über einen URI identifiziert und beinhaltet Schlüssel, die als Basis für die sichere Kommunikation der beiden Parteien genutzt werden. Der Sicherheitskontexttoken enthält keine Identitätsmerkmale. Eine eventuell erforderliche Authentifikation der Teilnehmer erfolgt vor der Bereitstellung des Token.

Das jeweilige Kontext-Token enthält außerdem Informationen zum Erstellungszeitpunkt und zur Gültigkeitsdauer. Zum Sichern der ausgetauschten Messages wird nicht das Kontext-Token selbst, sondern der Bezeichner verwendet, aus dem die Schlüssel abgeleitet werden. Nach Ablauf der Gültigkeitsdauer muss ein neues Kontext-Token ausgestellt werden.

2.9.3 Pervasive Umgebungen

Klassische Sicherheitsmechanismen gehen meist von der impliziten Annahme, dass es ausschließlich statische Komponenten im System gibt, aus. Zu solchen statischen Komponenten zählen unter anderem:

- Public Key Infrastruktur,
- Kommunikationsverbindungen zwischen den Principals,
- Sicherheitsmodule, meist signiert.

In mobilen pervasiven Umgebungen ist dies nicht mehr so einfach möglich, da hier die Einflussgrößen:

- Ad-hoc-Netzwerke und Ad-hoc-Applikationen,
- kurze Reichweiten der Geräte,
- Heterogenität,
- Selbstkonfiguration ohne Netzwerkmanagement,
- Serviceevolution,

den Einsatz solcher statischen Komponenten verbieten, beziehungsweise stark erschweren. Außerdem ist in einer solchen Umgebung die Sicherheit auf der reinen Kommunikationsebene nicht ausreichend, da dem Gegenüber nur bedingt vertraut werden kann. In einer solchen Umgebung muss auch die Sicherheit als ein Service verstanden werden, außerdem empfiehlt es sich, die anderen Services wie Principals zu betrachten.

2.9.4 Privacy

Eine Erweiterung der Security ist die Idee der Privacy, der Schaffung einer vom Benutzer kontrollierbaren privaten Sphäre. Historisch gesehen ist Privatleben erst in neuerer Zeit ein Wert, beziehungsweise ein Recht. Die Bewegung zur Individualität in den westlichen Nationen im 18.–20. Jahrhundert hatte die Idee des Privatlebens als Trennung vom öffentlichen Leben zur Folge. In dem Maße, wie die Macht der Nationalstaaten zur Kontrolle des Individuums anstieg, kam es zu Gegenmaßnahmen durch Einzelne. In diesem Sinne stellt Privacy in der IT den Versuch dar, der immer stärker werdenden Kontrolle des Menschen durch Organisationen, seien sie öffentlich oder privat, zu begegnen. Es gibt sieben Grundsätze für die Gestaltung von Privacy im IT-Bereich:

- Benutzerkontrolle und -zustimmung,
- minimale Funktionalität für eine eingeschränkte Nutzung,
- gerichtete Identität,
- Soft- und Hardwareheterogenität,
- Integration von Menschen,
- Konsistente Nutzung in diversen Kontexten.

Ein System zur Erhaltung der Privacy muss dem Benutzer Vertrauen gegenüber dem System geben. Dieses Vertrauen kann dadurch erreicht werden, dass der Benutzer die Kontrolle über die Rechte und Funktionen des Systems hat. Dieses Gefühl des Vertrauens und der Kontrolle muss unabhängig vom Kontext mit dem Benutzer in einem mobilen System wandern. Bezüglich des Zugriffs auf Daten sollte stets nur das Minimum, welches nötig ist, gespeichert werden und Benutzer mit der am ge-

ringsten identifizierenden Information gespeichert werden.¹⁰⁹ Umgekehrt formuliert bedeutet dies, dass je mehr Informationen aus verschiedenen Quellen aggregiert werden, desto höher das Risiko ist, dass die Privacy verletzt werden kann.¹¹⁰ Ein System muss den Benutzer auf Anfragen bezüglich seiner Identität aufmerksam machen und von diesem explizit die Zustimmung abfragen. In diesem Zusammenhang ist auch die Frage der digitalen Signatur problematisch, auf der einen Seite erscheint es sinnvoll, dass sich der Bürger in seinen Kontakten mit Behörden eindeutig identifizieren kann, auf der anderen Seite stellt eine Übertragung auf den privatwirtschaftlichen Sektor ein hohes Risiko in Bezug auf die Privacy dar.

¹⁰⁹ Dies widerspricht dem aktuellen politischen Trend. Legitimiert durch Maßnahmen gegen den Terrorismus werden immer mehr und immer stärker den einzelnen Menschen identifizierende Informationen von staatlichen Stellen gespeichert.

¹¹⁰ Die Finanzämter versuchen durch den Abgleich diverser Datenbanken, Bankkonteninformationen und Buchhaltungseinträgen Steuerhinterziehung zu entdecken.



<http://www.springer.com/978-3-540-79129-4>

Digitale Ökosysteme
Serviceorientierung bei dynamisch vernetzten
Unternehmen

Masak, D.

2009, IX, 326 S., Hardcover

ISBN: 978-3-540-79129-4