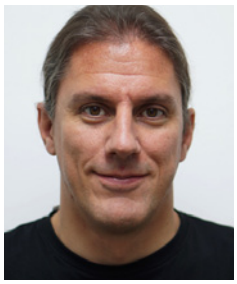

Dipl.-Inf. Eberhard Wolff
(eberhard.wolff@innoq.com)
arbeitet seit mehr als fünfzehn Jahren als Architekt und Berater oft an der Schnittstelle zwischen Business und Technologie. Er ist Fellow bei der innoQ. Als Autor hat er über hundert Artikel und Bücher geschrieben – u. a. ein Buch zu Continuous Delivery und eines über Microservices. Als Sprecher hat er auf zahlreichen internationalen Konferenzen, z. B. OOP, vorgetragen. Sein technologischer Schwerpunkt liegt auf modernen Architekturansätzen – Cloud, Continuous Delivery, DevOps, Microservices oder NoSQL spielen oft eine Rolle.

Services: SOA, Microservices und Self-contained Systems

Die Aufteilung komplexer Softwaresysteme in Services ist nicht neu: Service-orientierte Architekturen (SOA) nutzen diese Idee schon lange. In letzter Zeit sind Microservices als neuer Ansatz hinzugekommen. Microservice-Architekturen sind sehr flexibel und breit einsetzbar. Self-contained Systems (SCS) verwenden Microservices, um große und komplexe Systeme in unabhängige Module aufzuteilen. Das unterstützt die Zusammenarbeit der Teams. Dieser Artikel gibt einen Überblick über SOA, Microservices und Self-contained Systems.

Service-orientierte Architekturen (SOA)

Eine einheitliche Definition von SOA gibt es nicht. Aber zentral für SOA (siehe [Wik1]) sind Services. Sie sollen so grobgranular sein, dass sie eine eigenständige fachliche Funktionalität erfüllen. Die Services bieten ihre Schnittstelle im Netz an. So können sie von verschiedenen Programmiersprachen und Plattformen aus aufgerufen werden.

Typischerweise wird in einer SOA-Landschaft jede Anwendung in Services aufgeteilt. Das gilt auch für Anwendungen, die schon vor Einführung der SOA existierten. Beispielsweise könnte ein Customer-Relationship-Management (CRM) Dienste zum Anlegen von Kunden, zum Abfragen von Informationen über Kunden oder zum Anlegen neuer Interaktionen mit einem Kunden anbieten.

Diese Dienste sind alle gemeinsam im CRM implementiert (siehe Abbildung 1). Das CRM bildet eine Deployment-Einheit – also können alle Services nur gemeinsam in Produktion gebracht werden.

Neben den Services muss eine SOA noch weitere Elemente beinhalten:

- Die Kommunikation der Services muss eine einheitliche Technologie nutzen, über die alle Services erreichbar sind. Das kann ein ESBs (Enterprise Service Bus) für asynchrone Kommunikation sein. Eine Alternative ist SOAP, das unter anderem Kommunikation mit HTTP erlaubt.
- Die Integration und Orchestrierung wird üblicherweise in einer eigenen Schicht umgesetzt. Sie implementiert Geschäftsprozesse mithilfe der Services. Nützlich sind in diesem Kontext Orchestrierungslösungen. Beispielsweise kann der Bestellungsprozess die Informationen über die Bestellung im CRM ablegen und dann im Bestellsystem die Bearbeitung der Bestellung auslösen. Wenn ein neuer Service in den Prozess integriert werden soll, ist dazu nur eine Änderung des Prozesses notwendig. Die Services bleiben unverändert.
- Das Portal kann sich der Orchestrierung und der einzelnen Services bedienen. Es stellt eine Benutzeroberfläche dar. Also kann ein Benutzer sich über das Portal als Kunde registrieren oder eine neue Bestellung aufgeben. Ebenso kann ein Call-Center-Mitarbeiter alle Informationen zu einem Kunden heraussuchen – wahrscheinlich über ein anderes Portal speziell für Call-Center-Mitarbeiter.

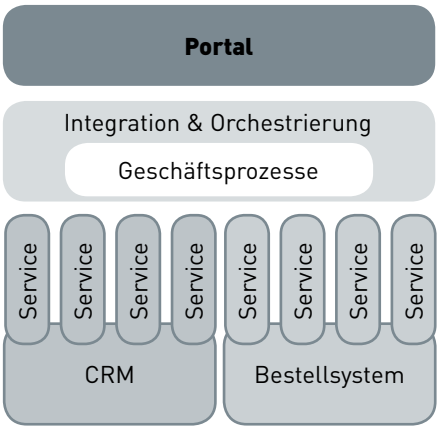

Abb. 1: Eine beispielhafte SOA-Architektur

SOA hat verschiedene Vorteile:

- Services können wiederverwendet werden. So ist es möglich, aus den verschiedenen Portalen dieselben Services zu nutzen oder unterschiedliche Prozesse aus denselben Services zu komponieren.
- Änderungen an den Geschäftsprozessen sind möglich, ohne dass dazu die Services geändert werden müssen. Es muss lediglich die Orchestrierung angepasst werden. Abhängig von der genutzten Orchestrierungstechnologie kann dazu eine Konfigurationsänderung ausreichend sein.

Eine SOA ist ein unternehmensweites Unterfangen. Alle Anwendungen in der gesamten IT-Landschaft müssen so umgestellt werden, dass sie Services anbieten. Außerdem sollten die Services idealerweise in eine Orchestrierung und ein Portal eingebracht werden.

Letztendlich ist das Ziel der SOA eine Flexibilisierung der IT. Änderungen an Prozessen können einfach durch die Modifikation der entsprechenden Geschäftsprozess-Modellierungen umgesetzt werden. Wenn Services anders zusammenspielen sollen, sind die Änderungen sehr schnell und einfach machbar.

Allerdings ist eine SOA mit erheblichen Investitionen verbunden. Alle IT-Systeme müssen in Services aufgeteilt werden, die im Netzwerk ansprechbar sind. Dazu muss jedes System erweitert werden. Ebenso müssen alle Geschäftsprozesse in der Orchestrierung-Schicht umgesetzt werden. Andernfalls sind sie nicht so einfach änderbar. Und die UI der Systeme muss in ein Portal eingebracht werden. Insgesamt ist ein recht hoher Aufwand notwendig, weil die gesamte Struktur der IT geändert wird. Dieser Aufwand ist ein zentraler Kritikpunkt an SOA (siehe [blo09]).

Den hohen Investitionen steht nur ein geringer Gewinn gegenüber: Lediglich die Implementierung von Geschäftsprozessen wird flexibler. Und das gilt auch nur, wenn die Änderung in den Services, der Orchestrierung oder dem Portal isoliert werden kann.

Sind Änderungen an mehreren Schichten erforderlich, so ist dazu ein größerer Aufwand notwendig. Schließlich müssen nicht nur alle Änderungen durchgeführt werden, sondern sie müssen auch gemeinsam in Produktion gebracht werden. Nur dann können die Features genutzt oder getestet werden. Das ist wesentlich aufwendiger, als wenn nur eine Änderung in einem einzigen System notwendig wäre.

Trotz der Herausforderungen von SOA bietet sich auf der Ebene einer unternehmensweiten IT aber eigentlich immer zumindest eine Aufteilung in Services an: Schließlich müssen oft verschiedene Systeme miteinander kommunizieren. Dazu ist eine einheitliche Kommunikationstechnologie notwendig. Es ist aber fraglich, ob eine vollständige SOA mit einer Aufteilung in eine getrennte Orchestrierung und Portal sinnvoll ist, denn das ist sehr aufwendig und bietet kaum einen Mehrwert.

Microservices

Ähnlich wie SOA sind auch Microservices (siehe [Wol15]) nicht genau definiert. Einige verstehen Microservices einfach als SOA. Aber eine solche Begriffsverwirrung hilft nicht weiter.

Auch bei Microservices kommunizieren Dienste miteinander. Dazu ist ebenfalls ein einheitliches Kommunikationsprotokoll notwendig. Jedoch ist die Ebene der Microservices anders: Der Einsatz von Microservices kann auf ein Projekt beschränkt sein. Es ist keine unternehmensweite Entscheidung. Microservices sind nur eine Möglichkeit, wie eine Anwendung modularisiert werden kann. Anwendungen können auch in Bibliotheken oder mit anderen Mechanismen in Module aufgeteilt werden. Microservices sind dazu eine Alternative.

Der entscheidende Unterschied zu SOA: Als Microservices können die Module unabhängig voneinander in Produktion gebracht werden. Sonst werden alle Module in der Anwendung zum Beispiel durch die Kompilierung zusammengefasst und dann gemeinsam in Produktion gebracht.

Jeder Microservice kann ein eigener Prozess sein, denn Prozesse können unabhängig voneinander in Produktion gebracht werden. Noch radikaler ist eine Aufteilung in getrennte virtuelle Maschinen oder Docker-Container. Dann hat jeder Microservice sogar seine eigene Betriebssystem-Installation. Das erzeugt eine noch stärkere Trennung der Microservices.

Die Microservices müssen zu einer Anwendung kombiniert werden. Konkret können Microservices beispielsweise mit RESTful HTTP integriert werden. Ebenso ist es denkbar, dass ein Microservice einen Teil einer Webschnittstelle implementiert.

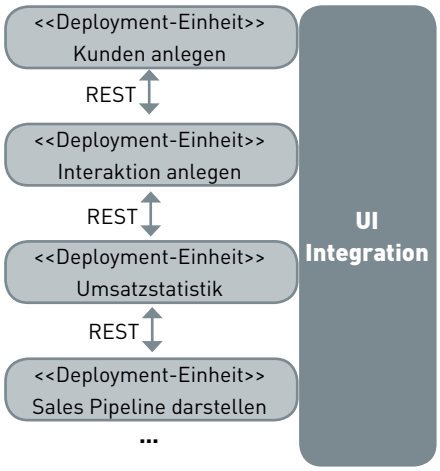

Abb. 2: Eine Microservices-Architektur

Die Anwendung ist dann die Kombination der Microservices, die sich gegenseitig aufrufen oder jeweils einen Teil der UI darstellen (vgl. Abbildung 2).

Also kann ein CRM aus verschiedenen Microservices aufgebaut werden. Ein Service kann zum Anlegen neuer Kunden dienen oder zum Anlegen einer neuen Interaktion mit einem Kunden. Jeder der Microservices kann unabhängig von den anderen in Produktion gebracht werden. Er bildet also eine Deployment-Einheit.

Auch getrennte Updates der einzelnen Services sind möglich. Bei einer SOA sind die Services nur eine Fassade über eine einzige Deployment-Einheit: Alle Services müssen gemeinsam in Produktion gebracht werden. Bei Microservices sind unabhängige Änderungen der einzelnen Bestandteile möglich (siehe Abb. 2). Das CRM als solches existiert eigentlich gar nicht. Es gibt nur noch Microservices, die miteinander integriert sind und zusammen die Funktionalitäten eines CRM anbieten.

Also setzen Microservices auf einer anderen Ebene an als SOA. Microservices dienen zur Strukturierung einer Anwendung, während SOA eine Strategie zur Strukturierung einer gesamten IT eines Unternehmens ist. Daher unterscheiden sich SOA und Microservices von der Ausrichtung her fundamental, auch wenn sie ähnliche Kommunikationsmechanismen nutzen können.

Ähnlich ist hingegen die Zielsetzung: Auch Microservices zielen darauf ab, die Entwicklung von Software zu flexibilisieren. Microservices erreichen das Ziel aber ganz anders. Nach Möglichkeit sollte eine Änderung an der Software nur die Modifikationen eines einzigen Microservice bedingen. Wenn das gelingt, muss nur eine relativ kleine Softwareeinheit geändert

und getestet werden und kann unabhängig von den anderen Modulen in Produktion gebracht werden. Das reduziert den Aufwand für Änderungen erheblich.

Microservices können auf organisatorischer Ebene unterstützt werden: Eine große Anwendung wird typischerweise von mehreren Teams entwickelt. Es liegt auf der Hand, die Teams anhand der technischen Skills zu organisieren: Zum Beispiel können alle Frontend-Entwickler zu einem Team zusammengefasst werden. Urlaubsvertretung oder fachlicher Austausch sind so sehr einfach. Weitere Teams können beispielsweise das Backend entwickeln oder sich um die Datenbank kümmern.

Die Aufteilung der Teams beeinflusst aber die Architektur. Das Gesetz von Conway (siehe [Con68]) besagt, dass eine Organisation nur eine Architektur hervorbringen kann, die ihren Kommunikationsbeziehungen entspricht. Im Gegensatz zu vielen Annahmen in der Softwareentwicklung gibt es in diesem Fall auch recht gute empirische Untersuchungen (siehe [Wik2]).

Ein konkretes Beispiel: Eine Aufteilung in drei Teams für Frontend, Backend und Datenbank führt zu Drei-Schichten-Architektur mit Backend, Frontend und Datenbank. Diese Beziehung ist nicht überraschend: Schließlich müssen Teams miteinander reden, wenn ihr Code eine Schnittstelle mit dem Code anderer Teams hat. Nur so kann die Schnittstelle sinnvoll umgesetzt werden.

Eine solche Organisation hat aber einige Herausforderungen. Bei einer Änderung muss ein Kunde mit drei Teams kommunizieren, wenn die Änderung nicht auf ein technisches Artefakt beschränkt ist. Die Arbeit der Teams muss eng koordiniert werden. Eine Verzögerung bei einem Team beeinflusst die anderen Teams, da sie Zulieferungen später erhalten. Diese Verzögerungen können sich aufsummieren und erhöhen das Projektrisiko.

Eine fachliche Architektur-Aufteilung ist bei Microservices praktisch vorgezeichnet: Schließlich soll eine Änderung möglichst nur einen Microservice betreffen. Also sollte jeder Microservice idealerweise eine bestimmte Fachlichkeit implementieren. Im Microservice-Umfeld wird meistens auch die Aufteilung der Teams anhand der Architektur vorgenommen.

Also bekommt jedes Team die Verantwortlichkeit für eine bestimmte Fachlichkeit, die in einem oder mehreren Microservices implementiert ist. Das Team muss daher technisch breit aufgestellt sein. Es muss schließlich Backend, Frontend und Datenbank für die Fachlichkeit verantworten. Das unterstützt die Idee der crossfunktionalen Teams aus der agilen Softwareentwicklung, die ebenfalls Teams mit vielen unterschiedlichen Rollen bevorzugen.

Der Vorteil dieses Vorgehens ist die Unabhängigkeit der Teams. Jedes Team kann an seiner eigenen Fachlichkeit arbeiten und neue Features umsetzten. Durch die Microservices kann es die Änderungen in Produktion nehmen, ohne dass dazu Absprachen mit anderen Teams notwendig sind – schließlich kann jeder Microservice unabhängig von den anderen produktiv gestellt werden. Und es ist keine umfangreiche technische Koordination notwendig: Jeder Microservice ist ein eigener Prozess oder sogar eine eigene virtuelle Maschine. Er kann daher in einer anderen Programmiersprache und mit einer ganz anderen Technologie implementiert werden. Jedes Team kann die Technologie nutzen, die für das jeweilige Problem angemessen ist.

Das ist allerdings nur eine Option: Natürlich können alle Microservices einer Anwendung auch einen einheitlichen Technologie-Stack nutzen. Aber selbst dann ist die technologische Freiheit von Vorteil: Beispielsweise kann in einem Java-Programm jede Bibliothek nur in einer bestimmten Version genutzt werden. Wenn alle Teams an einer großen Anwendung arbeiten, müssten also alle Teams sich genau auf eine Version einigen. Wenn ein Team eine neue Version mit einem Bug-Fix benötigt, müssen alle anderen Teams dieser Änderung zustimmen. Schließlich kann der Code aller dieser Teams durch die neue Version beeinflusst werden. Bei Microservices ist das nicht notwendig, denn jeder Microservice kann eine andere Version der Bibliothek nutzen.

Ein weiterer Vorteil von Microservices ist, dass sie andere Anwendungen ergänzen können. Das erleichtert den Einsatz bei der Migration von Software. Beispielsweise können Microservices einige Anfragen beantworten, während andere Anfragen weiterhin von der alten Software beantwortet werden. Also können die Vorteile der Microservices genutzt werden, ohne dass dazu gleich die gesamte Anwendung auf Microservices umgestellt werden muss.

Microservices setzten also Flexibilität mit ganz anderen Maßnahmen um, als SOA das tut: Das Vorgehen betrachtet einzelne Anwendungen, nicht die unternehmensweiten IT. SOA setzt auf eine Trennung der Software in einen Service-Baukasten mit einem Portal und einer Orchestrierungslösung. Dadurch wird die Software in drei Schichten geteilt. Microservices dagegen setzen darauf, alle Änderungen in einem Microservice umzusetzen und erfordern daher eine fachliche Aufteilung.

Self-contained Systems

Bei Microservices gibt es zahlreiche Spielarten: Die Größe der Microservices kann sich sehr stark unterscheiden. Ebenso besteht keine Einigkeit, ob ein Microservice eine UI enthalten soll oder nicht. Die Definition von Self-contained Systems (SCSs) ist hingegen relativ klar (siehe [scs1]):

- Jedes Self-contained System ist eine eigene Webanwendung. Sie enthält alle Daten, alle Logik und auch den Code zur Darstellung der Webschnittstelle.
- Für jedes Self-contained System ist ein Team verantwortlich.
- Die Kommunikation mit Fremdsystemen und anderen SCSs ist nach Möglichkeit asynchron. Das entkoppelt das SCS von anderen Systemen. Ein SCS funktioniert sogar, wenn ein abhängiges System für einige Zeit ausgefallen ist.
- Das SCS ist zwar eine Webanwendung, kann aber auch eine Service-API haben. So kann die Logik des SCS auch von anderen SCS oder mobile Clients genutzt werden.
- Jedes SCS soll seine eigene UI haben und sich auch keinen Geschäftscode mit anderen SCS teilen.

Ähnlich wie Microservices sollen auch SCSs zusammen eine Anwendung ergeben. Neben der schon erwähnten asynchronen Kommunikation kann dafür auch eine Integration auf der Ebene der Webanwendungen genutzt werden. Eine einfache Integrationsmöglichkeit sind Links: Beispielsweise kann mit einem Link zum CRM in einer Bestellung alle Informationen zu einem Kunden angezeigt werden. Allerdings muss der Nutzer auf den Link klicken.

Aber das kann vermieden werden: Die Informationen aus dem CRM können direkt angezeigt werden. Dazu muss der Link lediglich mit dem HTML-Code aus

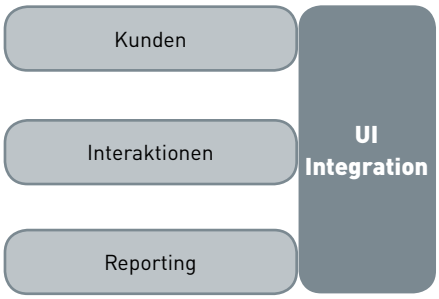

Abb. 3: Ein CRM mit Self-contained Systems

dem CRM ersetzt werden. Das kann beispielsweise JavaScript-Code erledigen: Er scannt eine Seite nach bestimmten Links und ersetzt die Links durch den Inhalt der referenzierten Seite.

Eine andere Möglichkeit sind Features von Web-Servern wie Server Side Includes. Dann ersetzt der Webserver Teile der Webseite durch die referenzierten Inhalte.

Eine solche Integration führt zu einer sehr losen Kopplung: Die Anwendungen müssen technologisch nur eine Webschnittstelle haben – die ist aber für die Darstellung der UI sowieso notwendig. Weitere Annahmen über die Technologien sind nur erforderlich, wenn die Anwendungen asynchron kommunizieren. Dann muss es noch eine passende Technologie für die Übermittlung der Nachrichten geben.

Auch die Robustheit des Systems profitiert von einer Frontend-Integration. Wenn ein System gerade nicht verfügbar ist, kann der JavaScript-Code den Link nicht durch HTML ersetzen. Das System bleibt aber benutzbar. Der Benutzer kann sogar auf den Link klicken und so das ausgefallene System nutzen, wenn es wieder verfügbar ist.

Natürlich gibt es auch bei der UI-Integration Herausforderungen: Es ist nicht so einfach, ein gemeinsames „Look and Feel“ zu erreichen. Auch sonst muss bei der Implementierung darauf geachtet werden, dass Teile der UI nachgeladen werden. Gerade Entwickler, die eigentlich im Backend zu Hause sind, müssen für solche Architekturen neue Frontend-Techniken und -Technologien kennenlernen.

Abbildung 3 zeigt wieder das CRM als Beispiel: Die SCSs sind im Vergleich zu den Microservices grobgranularer. Eine Integration finde nur über die UI statt.

SCS treffen auch eine klare Aussage zum organisatorischen Aspekt der Architektur: Ein SCS ist genau das, was ein Team entwickelt und auch in Produktion bringt. Das entspricht der Idee von der Unterstützung der Architektur durch die Organisation, der schon im Abschnitt über Microservices erläutert worden ist.

Im Vergleich zu Microservices können SCS größer sein: Sie sollen schließlich eine komplette Webanwendung darstellen. Tatsächlich kann ein SCS intern aus mehreren Microservices zusammengesetzt sein. Dadurch ergeben sich noch kleinere Deployment-Einheiten, aber die Komplexität steigt auch, weil diese Deployment-Einheiten auch alle in Produktion betrieben werden müssen. Letztendlich ist eine solche Aufteilung also ein Trade-Off.

Bei der Architektur eines SCS unterscheidet man die Mikro- und Makroarchitektur. Mikroarchitektur sind alle Entscheidungen, die jedes Team für sein SCS selber treffen kann. Makroarchitektur sind Entscheidungen, die global für alle Teams und SCSs festgelegt werden.

SCSs erlauben genauso wie Microservices mehr Entscheidungen in der Mikroarchitektur zu treffen. Das erhöht die Unabhängigkeit der Teams und erlaubt mit geringerem Koordinationsaufwand große Systeme zu entwickeln. Nur wenige Entscheidungen müssen zwingend in der Makroarchitektur getroffen werden – beispielsweise das Protokoll für die Kommunikation der SCSs untereinander oder die Technologien für die UI-Integration.

Alle anderen Entscheidungen – auch zu der Programmiersprache und Plattform – können Teil der Mikroarchitektur sein. Natürlich könnten alle Entscheidungen in der Makroarchitektur getroffen werden. Das widerspricht jedoch dem Ziel der weitgehenden Unabhängigkeit der Teams.

SCS sind von der Größenordnung her zwischen Microservices und SOA angesiedelt. Ein SCS kann aus mehreren Microservices bestehen. SCSs strukturieren aber im Gegensatz zu einer SOA nicht die ganze IT eines Unternehmens. SCSs können auch durchaus nützlich sein, um ein komplexes Portal in der Architektur zu unterstützen.

Fazit

Nur auf den ersten Blick sind SOA und Microservices ähnliche Konzepte: Beide basieren zwar auf Services. Bei SOA wird die Unternehmens-IT in Services aufgeteilt während Microservices die Entscheidung eines einzelnen Projekts sind.

Beide Ansätzen stellen die Flexibilität in den Vordergrund. Bei SOA soll die einfachere Wiederverwendung und Komposition von Services dazu führen. Bei Microservices ist es die Aufteilung großer Anwendungen in kleine Dienste, die einzeln in Produktion gebracht werden können.

Während eine SOA eine umfassende Änderung der kompletten Unternehmens-IT benötigt, können Microservices in einem einzigen Projekt eingeführt werden und auch zur Ergänzung von Legacy-Software genutzt werden. So sind Risiko und Aufwand für Microservices geringer. Aber am Ende sind die Ansätze nicht wirklich vergleichbar: Microservices sind im Gegensatz zu SOA eben kein Ansatz, um eine ganze IT-Landschaft zu strukturieren.

SCSs bauen ein System aus einzelnen Webanwendungen auf. Integration soll auf der UI-Ebene stattfinden, aber auch asynchrone Kommunikation mit anderen SCSs ist denkbar. So nehmen SCSs Ideen aus dem Microservices-Bereich wie die Aufteilung eines Systems in kleinere Einheiten auf. Die SCSs sind im Vergleich zu Microservices eher grobgranular. So ergibt sich ein Ansatz, den heute viele Projekte gerade für die Implementierung großer Webanwendungen nutzen (siehe [scs2]).

Durch die lose Kopplung ermöglichen SCS dank der UI-Integration eine noch unabhängigere Arbeit der Teams. Jedes Teams kann selbstständig Anforderungen implementieren, denn ein SCS implementiert jeweils die vollständige fachliche Funktionalität. SCSs nutzen also die Ideen der Microservices, um eine unabhängige Arbeit an Funktionalitäten weiter zu erleichtern – eine zentrale Aufgabenstellung jeder Modularisierung. ■

Literatur & Links

[Wik1] https://de.wikipedia.org/wiki/Serviceorientierte_Architektur

[blo09] http://apsblog.burtongroup.com/2009/01/soa-is-dead-long-live-services.html

[Wol15] Eberhard Wolff: Microservices: Grundlagen flexibler Software Architekturen, dpunkt, 2015, ISBN 978-3864903137.

[Con68] http://www.melconway.com/research/committees.html

[scs1] http://scs-architecture.org

[scs2] http://scs-architecture.org/links.html

[Wik2] https://de.wikipedia.org/wiki/Gesetz_von_Conway#Studien