Microservices und Makro-Architektur: drei zentrale Entwurfsfragen bei vertikalen Anwendungs-Architekturen

Stefan Zörner, embarc

Moderne Architektur-Stile wie Microservices oder Self-contained Systems lassen Teams, die einzelne Teile entwickeln, viel Freiheit bei Technologie-Entscheidungen. Drei Themen entpuppen sich jedoch regelmäßig als Kandidaten, um übergreifend adressiert zu werden, damit die Anwendung wie aus einem Guss wirkt oder andere Architekturziele nicht verfehlt. Dieser Artikel stellt die Fragestellungen vor und zeigt Antworten auf.

Die großen IT-Trends der letzten Jahrzehnte – seien es Objektorientierung, SOA, Model-Driven Architecture, Cloud oder zuletzt Microservices – versprechen im Grunde nur zwei Dinge: Kosteneffizienz und/oder Flexibilität. Kosteneffizienz heißt Kosten sparen in Entwicklung und Betrieb, erreicht beispielsweise durch effizientere Werkzeuge, Methoden oder (wie bei SOA und OO) Wiederverwendung. Flexibilität bedeutet, schnell auf Veränderungen reagieren zu können, etwa auf neue fachliche Anforderungen, technologische Trends oder Schwankungen in der Last. Microservices versprechen dabei in erster Linie Flexibilität. Wie genau erfüllen Microservices diesen Wunsch? Wie erreicht man Flexibilität durch Anwendung des Architektur-Stils?

Facette von Flexibilität	In Microservices unterstützt durch ...
Neue fachliche Anforderungen schnell umsetzen können	▪ Kleinteiligkeit („Micro") als Gegenmodell zum Monolithen ▪ Lose Kopplung der Services, leichte Austauschbarkeit
Technologische Trends schnell aufgreifen können	▪ Hoher Freiheitsgrad bei Technologieauswahl ▪ Services mit unterschiedlichem Technologie-Stack möglich
Schwankungen in der Last gut auffangen können	▪ Services einzeln skalierbar ▪ Services schnell start- und wegwerfbar

Tabelle 1: Flexibilität erreichen

Ein Spannungsfeld

Microservices verfolgen die Idee, eine einzelne Anwendung in kleine, lose gekoppelte Services zu zerlegen. Für diese Services gelten charakteristische Eigenschaften *[1]*, die positiv auf unterschiedliche Aspekte oder Fähigkeiten von Flexibilität wirken. *Tabelle 1* stellt diese gegenüber.

Wie bei Trends üblich, klingt der Ansatz verlockend. Doch schon in der zitierten Definition *[1]* schlummert ein Spannungsfeld. Denn der möglichen technologischen Vielfalt der Services (unterschiedliche Paradigmen, Programmiersprachen, Bibliotheken etc.) steht die Anforderung nach einer einzelnen Anwendung gegenüber *(siehe Abbildung 1)*. Ein gewisser Grad an Einheitlichkeit ist dafür unabdingbar.

Themen für die Makro-Architektur

Software-Architektur wird gerne als Summe wichtiger Entscheidungen verstanden, die im weiteren Verlauf schwer zurückzunehmen sind. Auch in anderen Kontexten als Microservices sehen wir unterschiedliche Ebenen von Entscheidungen. Eine Systemlandschaft und einzelne Anwendungen zum Beispiel oder eine Produktfamilie und einzelne Ausprägungen oder Varianten sowie eben wie hier eine Anwendung und einzelne Services. Die Frage lautet also: Welche Aspekte sind für alle Teile (Anwendungen, Ausprägungen, Services) des Ganzen (Systemlandschaft, Produktfamilie, Anwendung) gleich und wo haben einzelne Teile (oder besser: die dafür verantwortlichen Teams) Spielraum?

Die erste Ebene bezeichnen wir jeweils als Makro-Architektur. Dinge in der Makro-Architektur einer Microservices-Lösung zu vereinheitlichen, verspricht gewisse Vorteile; Freiheiten zu gewähren ebenso. Letzteres macht den Microservices-Architekturstil mit zu dem, was er ist. In *Tabelle 2* sind häufig genannte Argumente für Standardisierung und Individualisierung aufgezählt.

Die Vielfalt von Themen, bei denen Teams diskutieren können, ob sie diese in der Makro-Architektur für alle Services einheitlich adressieren wollen, ist groß. Sie zerfallen in diese (nicht 100% trennscharfen) Kategorien:

- Interaktion mit Benutzern und Anbindung von Fremdsystemen, also UI- und Integrationsthemen
- Technische Aspekte unter der Haube, wie zum Beispiel Programmiersprache oder Sicherheit

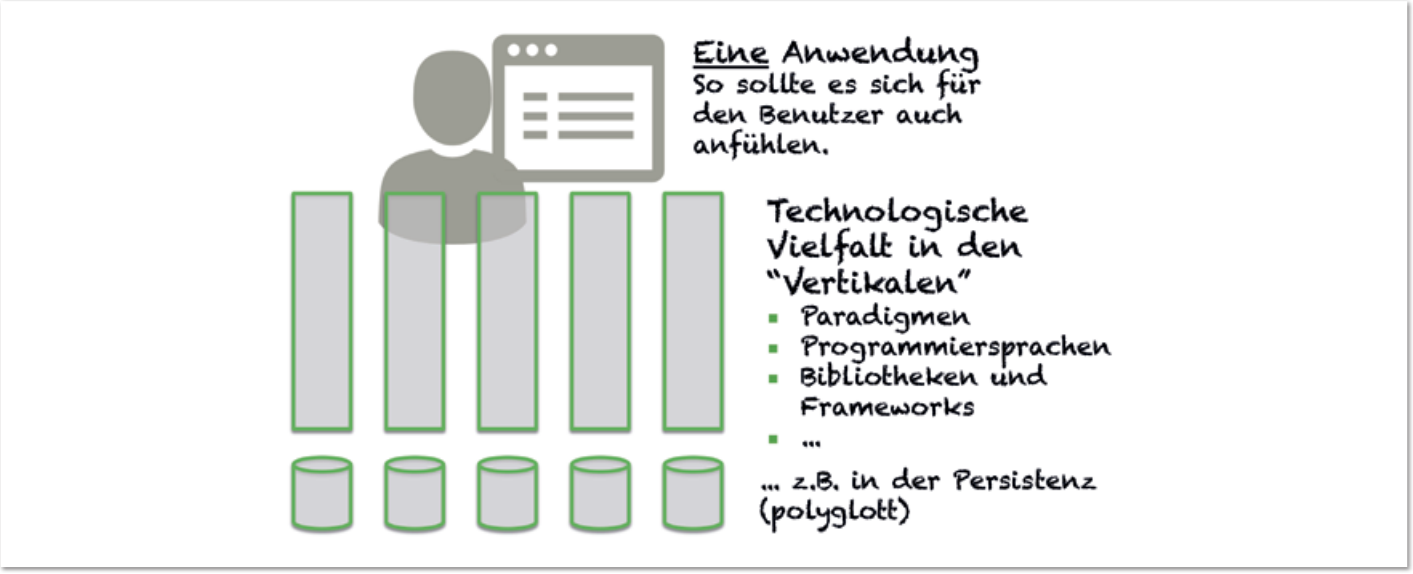

Abbildung 1: Technologische Freiheiten in den Vertikalen vs. Eine Anwendung

Pro Vereinheitlichung	Pro Individualisierung
▪ Entwickler wechseln leicht zwischen Teams und Projekten ▪ Konzentration auf Fachlichkeit leichter möglich ▪ Wiederverwendung technischer Lösungen ▪ Fehlervermeidung in kritischen Bereichen durch erprobte Konzepte	▪ Einsatz optimaler Lösungen für spezifische Probleme möglich ▪ Neue Trends lassen sich schneller aufgreifen ▪ Fehlentscheidungen haben geringere Relevanz ▪ Geringere Abhängigkeit von einzelnen Lieferanten

Tabelle 2: Häufig genannte Vorteile

Abbildung 2: Ein bunter Kandidatenzoo für übergreifende Themen

- Entwicklung und Weiterentwicklung, also alles rund um das Vorgehen und die Vorgehensweise (etwa Entwerfen, Quelltext verwalten, Testen, Bauen etc.)
- Zielumgebung und Betriebsaspekte, hierzu zählen Infrastruktur und Middleware, Monitoring, Disaster-Recovery etc.

Abbildung 2 zeigt eine Tag-Cloud mit vielen konkreten Themen zur Illustration. Sie sind nach den vier aufgezählten Kategorien farblich kodiert.

Bei welchen dieser zahllosen Themen sollten Teams vereinheitlichen? Wo sollten sie Spielraum haben, um spezifische Lösungen und Innovation zu ermöglichen? Vielleicht reichen für bestimmte Themen auch Vorschläge, die den Start erleichtern, sich mit der Zeit entwickeln und durch neue ersetzt werden.

Im Einzelnen hängt das vom Vorhaben und vom Kontext ab. Gleichzeitig gibt es zu bestimmten Themen nach Erfahrung des Autors sehr regelmäßig Diskussionen darüber, „ob und, wenn ja, wie wir das zentral machen …“. Er hat drei besonders häufige Fragestellungen herausgepickt, um sie im Folgenden zu diskutieren.

Thema 1: Die UI-Frage

Eine häufige Anforderung an Microservices-Lösungen: Trotz mehrerer Teile soll sich die Anwendung dem Benutzer „wie aus einem

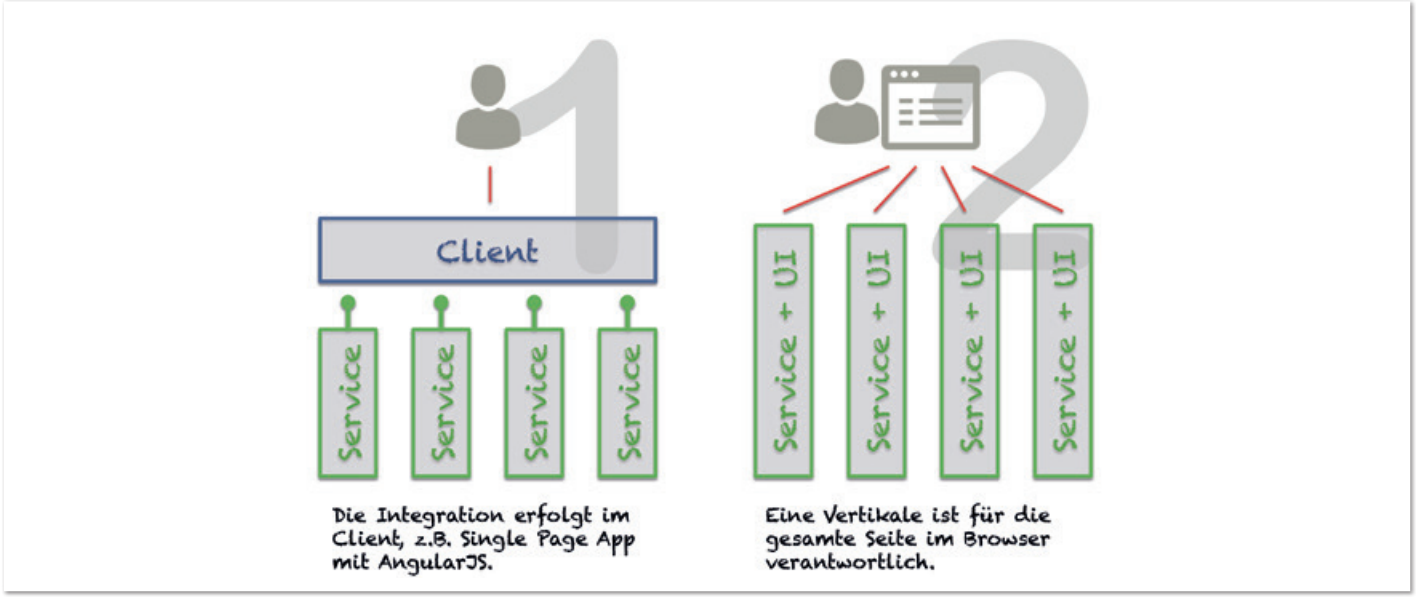

Abbildung 3: Zwei extreme Lösungsoptionen für die UI-Frage

Option 1: Gemeinsames UI für UI-lose Vertikalen	Option 2: Separate UIs für die Vertikalen
▪ Inhalte und Funktionen aus verschiedenen Themen lassen sich nahtlos integrieren – keine Brüche ▪ Einheitliches User Interface leicht erreichbar ▪ Spezialisiertes Team für optimale UX denkbar	▪ Teams vollumfänglich für Thema verantwortlich (Cross-funktionale Teams) ▪ Technologische Freiheiten ▪ Unabhängiges Arbeiten leicht möglich

Tabelle 3: Jeweilige Stärken der beiden UI-„Extrem"-Optionen

Guss" präsentieren. Die Frage: Wie realisieren wir mit mehreren Teilen (und Teams) ein UI? Darauf gibt es zwei extreme Antworten (und viele dazwischen).

Die erste Extremposition: Die einzelnen Microservices haben keinen UI-Anteil, sondern nur eine (REST-)Schnittstelle. Ein gemeinsamer Client greift auf alle Microservices zu. *Abbildung 3* zeigt links die Idee schematisch.

Die zweite Extremposition: Vertikalen (z.B. Microservices) bringen jeweils die komplette UI für „ihr Thema" mit. Die Integration erfolgt beispielsweise über Links im Browser. *Abbildung 3* zeigt diese Idee rechts.

.

Beide Positionen haben ihre Berechtigung und sind nicht nur akademischer Natur. Netflix hat beispielsweise die erste Option gewählt und entwickelt gleich mehrere übergreifende Clients in jeweils auf die entsprechende UI-Technologie spezialisierten Teams. Xing hingegen verfolgt bei seiner Webseite einen Ansatz nach Variante zwei.

Die unterschiedlichen Antworten auf die gleiche Frage verwundern nicht: Beide Optionen haben unbestreitbare Stärken, skizziert in *Tabelle 3*. Deswegen sind Lösungen wie so oft in der Software-Architektur ein Ausbalancieren und Kompromissefinden. Die perfekte User Experience lässt sich typischerweise mit der ersten Option (gemeinsames UI) erreichen. Der Wunsch, Teams unabhängig entwickeln und releasen zu lassen, führt zu Lösungsansätzen in Richtung der zweiten Option.

Thema 2: Kommunikation

Eine weitere häufige Anforderung an Microservices-Lösungen: Ein Service benötigt Funktionalität und/oder Daten eines anderen Service, um seine Aufgabe zu erledigen. Die Frage: Dürfen Services miteinander reden, und wenn ja, wie? Wenn nein, wie realisieren wir dann obige Anforderung?

Die Relevanz für Microservices-Architekturen liegt im Wunsch nach loser Kopplung, etwa um Teile leicht austauschen zu können, und auch, um Teams möglichst unabhängig voneinander arbeiten zu lassen. Technisch gesehen zerfällt die Frage streng genommen in zwei detailliertere Entscheidungen: direkte vs. indirekte Kommunikation sowie synchrone vs. asynchrone.

Bei synchroner Kommunikation wartet ein Client (etwa ein Service) auf die Antwort des genutzten Service und blockiert. Bei asynchroner Kommunikation wartet der Client nicht auf eine Antwort. Er erhält diese falls nötig später, gegebenenfalls über einen anderen Kanal.

Bei direkter Kommunikation kennt der Client den genutzten Service und spricht ihn direkt an. Im indirekten Fall kommuniziert der Client über eine Middleware, die ihn vom angesprochenen Service entkoppelt. Client und Server kennen sich also nicht. In der Makro-Architektur einer Microservices-Lösung ist zu klären, welche Kommunikationsmuster in welchen Situationen zulässig sind und wie sie umzusetzen sind.

Die engste Kopplung entsteht durch direkte, synchrone Kommunikation. Hier nutzt ein Service direkt Funktionalität von einem anderen Service und blockiert während des Aufrufs. Für die Implementierung ergeben sich zwei Herausforderungen. Zum einen ist das Auffinden („Service Discovery") zu klären. Die klassische Netflix-Architektur beispielsweise sieht hier Eureka als Service Registry vor. Auch moderne Plattformen zur Container-Orchestrierung (wie Kubernetes) bieten direkt Lösungen hierzu an.

Die zweite Herausforderung: Was, wenn ein Service fehlerhaft ist, gar nicht oder langsam antwortet? Typische Lösungen sind Resilience-Muster wie Circuit Breaker, bei Netflix etwa implementiert durch Hystrix. Eine gute Quelle für Lösungen rund um diese Herausforderungen bietet Chris Richardson auf seiner Webseite *[2]* und in seinem Buch „Microservices Patterns" an. Service Registry und Circuit Breaker sind Beispiele für Muster dort – die Diskussion erfolgt Technologie-neutral und nennt am Ende Beispiel-Implementierungen für verschiedene Plattformen wie Java.

Da die direkte, synchrone Kommunikation zu einer engen Kopplung führt, erfreuen sich andere Konstellationen großer Beliebtheit. Hier kommen dann Messaging und Events zum Einsatz. Die beteiligten Kommunikationspartner akzeptieren mitunter „eventual consistency" als Preis für geringere Abhängigkeiten.

Thema 3: Security

Kevin Hoffman führt in „Beyond the Twelve Factor App" *[3]* aus: „Security sollte niemals ein nachträglicher Einfall bei Deiner Anwendungsentwicklung sein." Die Frage ist: Welche Themen rund um Security adressieren wir in der Makro-Architektur? (und wie?) Tatsächlich ist Security ein weites Feld. *Abbildung 4* zeigt häufige Aufgaben, denen sich Teams im Microservices-Umfeld stellen.

Im Einzelnen geht es um folgende Themen:

- Die Benutzer der Anwendung müssen sich anmelden (Authentifizierung), deren Berechtigung muss überprüft werden (Autorisierung). Der Zugriff vom Browser erfolgt in der Regel verschlüsselt.
- Stellt die Anwendung ein API bereit, ergibt sich hier für die Clients die gleiche Aufgabenstellung wie für (menschliche) Benutzer. Die Lösung kann gleich aussehen, muss aber nicht.
- Falls Services mit anderen Services kommunizieren (siehe Thema 2), ergeben sich auch hier Fragen der Authentifizierung, Autorisierung, Verschlüsselung.
- Schließlich greifen Services mitunter auf Datenbanken oder generell Persistenz-Lösungen zu oder nutzen andere Fremdsysteme, in denen Berechtigungen überprüft werden. Wo und wie speichert man beispielsweise nötige Zugangsdaten (Zertifikate, Kennwörter etc.)?

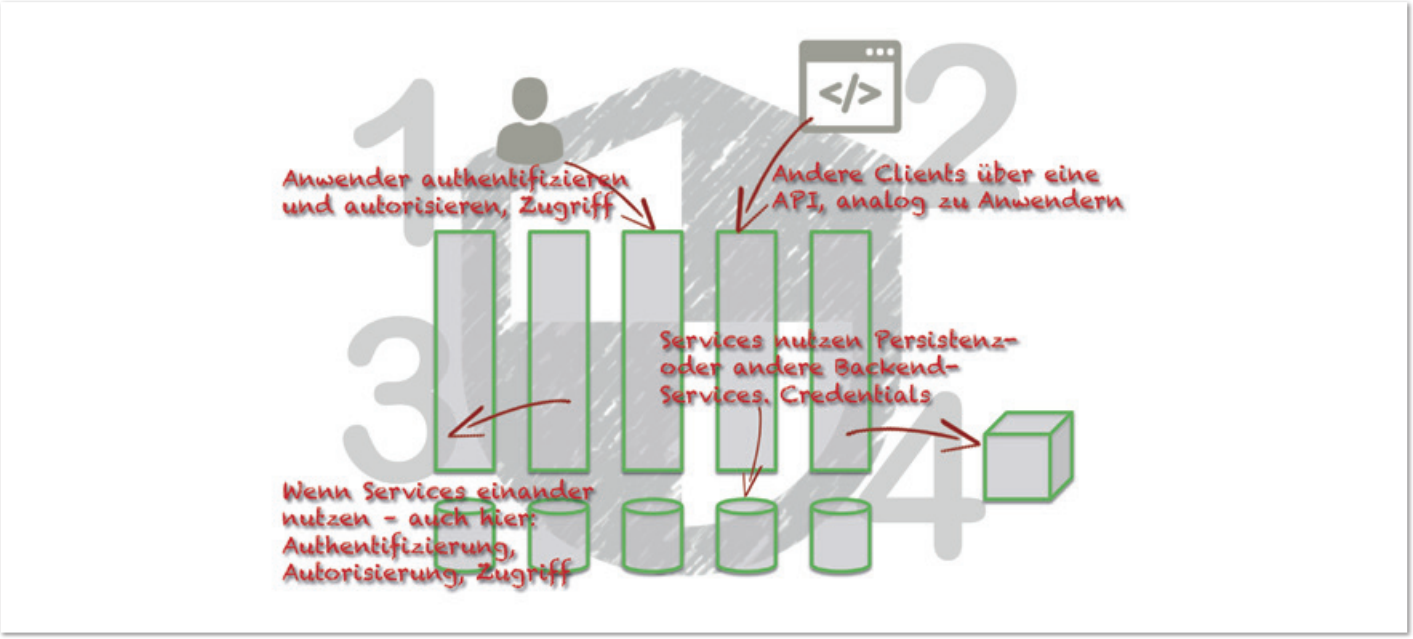

Abbildung 4: Zentrale Aufgaben rund um Security bei Microservices

Alle Punkte mit allen Optionen auszudiskutieren, sprengt hier den Rahmen. Zumindest eine häufige Antwort im ersten und zweiten Themenfeld soll allerdings kurz vorgestellt werden, da sie Verantwortlichkeiten zwischen Makro- und Mikro-Architektur interessant ausbalanciert.

Die Authentifizierung erfolgt dort zentral in der Makro-Architektur durch einen gemeinsamen Service. Single Sign-on ist erforderlich, damit die Anwendung wie aus einem Guss erscheint. Die einzelnen Services erhalten ein Authentifizierungs-Token, etwa via JSON Web Token (JWT). Die Autorisierung, also die Überprüfung der Berechtigungen, obliegt dann jedoch den einzelnen Services.

Ein Service erhält lediglich die Identität des Aufrufers und gegebenenfalls weitere Eigenschaften (wie globale Rollen). Hintergrund dieser Strategie ist zum einen der Wunsch, dass Teams neue Berechtigungen in ihren Services einführen und überprüfen können, ohne eine zentrale Komponente dafür anpassen zu müssen. Diese obliegen oftmals einem anderen Team und man könnte nicht mehr eigenständig liefern. Weiterhin sind Berechtigungen oftmals fachlich getrieben, die einzelnen Services (Vertikalen) fachlich geschnitten und somit der natürliche Ort dafür.

Weitere Informationen und Fazit

Ein Architektur-Stil, der klarere Präferenzen für die genannten Themen parat hat, sind die Self-contained Systems (SCS) *[4]*. Vertikalen (dort: Systeme) haben in SCS stets eine Web-UI. Kommunikation hat wo immer möglich asynchron zu erfolgen. Geteilte Infrastruktur sollte wo immer möglich vermieden werden. Ein zentraler Authentifizierungsdienst ist hier vielleicht eine der zulässigen Ausnahmen. Die ISA-Principles *[5]* bündeln Erfahrungswissen zu Microservices und SCS in insgesamt neun Prinzipien. Eines davon (das dritte) fordert die explizite Bearbeitung von Makro- und Mikro-Architektur.

Zusammenfassend ist ein gutes Verständnis von Makro- und Mikro-Architektur zentral für den Einsatz moderner Architektur-Stile. UI-, Kommunikations-, und Security-Themen sind häufige Brennpunkte für eine frühe, initiale Bearbeitung in der Makro-Architektur. Entscheidungen dort bergen oftmals Kompromisse, die entlang der Qualitätsziele ausbalanciert gehören. Dazu müssen diese natürlich für das Vorhaben bekannt sein.

Links und Literatur

[1] Martin Fowler, James Lewis, Microservices: a definition of this new architectural term: *https://martinfowler.com/articles/microservices.html*

[2] Chris Richardson: *http://microservices.io/patterns*

[3] Kevin Hoffman, Beyond the Twelve Factor App, O'Reilly Media 2016

[4] Self-contained Systems (SCS): *http://scs-architecture.org*

[5] Independent Systems Architecture: *https://isa-principles.org*

[6] Architektur-Spicker zu Microservices, Kurzreferenz: *https://www.embarc.de/architektur-spicker*

Stefan Zörner
stefan.zoerner@embarc.de

Stefan Zörner unterstützt Kunden von embarc in Architektur- und Umsetzungsfragen mit dem Ziel, gute Architektur-Ansätze wirksam in der Implementierung zu verankern. Sein Wissen und seine Erfahrung teilt er regelmäßig in Vorträgen, Artikeln und Workshops. Stefan ist Apache Committer, aktives Board-Mitglied im iSAQB und Autor des Buchs „Software-Architekturen dokumentieren und kommunizieren".