INNOQ

Self-contained Systems

Modulare Webanwendungen, die mit der Organisation skalieren

Johannes Seitz • Florian Fritzsche

Self-contained Systems

Modulare Webanwendungen, die mit der Organisation skalieren

Johannes Seitz
Florian Fritzsche

innoQ Deutschland GmbH
Krischerstraße 100 · 40789 Monheim am Rhein · Germany
Phone +49 2173 33660 · www.INNOQ.com

Layout: Tammo van Lessen with XƎLaTeX
Design: Murat Akgöz
Typesetting: André Deuerling

Self-contained Systems – Modulare Webanwendungen, die mit der Organisation skalieren
Published by innoQ Deutschland GmbH
2. Auflage · Oktober 2026

Copyright © 2026 Johannes Seitz, Florian Fritzsche

Inhaltsverzeichnis

1 Self-contained Systems in aller Kürze **1**

Was sind Self-contained Systems? ... 1

Was bedeutet es, „Self-contained" zu sein? ... 2

Unabhängig von der Idee bis in die Produktion ... 3

Unabhängig betreibbar ... 4

Architekturentscheidungen in Self-contained Systems ... 4

Die Eigenschaften eines SCS ... 5

Sind SCS nicht das Gleiche wie Microservices? ... 6

2 Domänenarchitektur **9**

Eine Domänenarchitektur entwerfen ... 9

Die Facharchitektur bewerten ... 10

Schnittstellen für unabhängige Systeme ... 12

Die Daten- und die Prozessfalle vermeiden ... 14

3 Self-contained Systems und die Organisation **17**

Softwareentwicklungsteams und SCS ... 17

Die Veränderung meistern ... 19

4 Mikro- und Makroarchitektur **21**

Betriebs- und Deploymentumgebung ... 22

Technologie- und Querschnittsstandards ... 22

Schnittstellen und Systemintegration ... 23

Nutzeroberflächen ... 24

Referenzanwendungen und Platform Engineering ... 25

5 Integrationsmuster **27**

Link-Integration ... 27

UI-Integration ... 28

Integration durch asynchrone Aufrufe ... 29

Integration durch synchrone Aufrufe ... 32

Alle Dimensionen der Kopplung betrachten 33

6 Integration durch asynchrones Messaging **35**

Pull vs. Push: Brauche ich eine Messaging-Middleware? 35

Nachrichtentypen: Events und Commands 37

Event-carried State Transfer 39

Reliable Messaging 40

7 UI-Integration **45**

Integration durch isolierte Frames 45

Transkludierter Content 46

Einbindung von „Self-contained Components“ 48

Module Federation 50

Konsistenter Look and Feel 50

Der gemeinsame Rahmen 51

8 Self-contained Systems and Agentic Engineering **53**

TL;DR 53

Die Case Study 53

Die Herausforderung: Bestehende Systeme weiterentwickeln 54

Ein Change: Lieferantenverfügbarkeit geografisch eingrenzen 54

Eine fachliche Änderung „End to End“ 55

Unabhängig bis in den Betrieb 56

Nicht mehr Kontext. Mehr Relevanz! 57

Schnellere Implementierung allein reicht nicht 57

9 Fazit **59**

Anhang **61**

Danksagung 61

Über INNOQ 61

Unsere Primer 62

Literaturhinweise 63

Die Autoren **65**

1 Self-contained Systems in aller Kürze

Was sind Self-contained Systems?

Self-contained Systems (im Folgenden: SCS) sind ein Architekturstil, der große IT-Systeme in mehrere eigenständige, zusammenarbeitende Webanwendungen zerlegt. Dabei werden fachlich komplexe Systeme so modularisiert, dass die entstehenden Bausteine sich auf fachliche Teilbereiche fokussieren, die unabhängig weiterentwickelt und betrieben werden können.

Der SCS-Architekturstil adressiert mehrere Probleme, unter denen insbesondere Systeme mit gewachsenen monolithischen Anwendungsarchitekturen leiden:

- eine langsame Entwicklungsgeschwindigkeit, verursacht durch zunehmende Abhängigkeiten zwischen einzelnen Teilen
- lange Testzyklen
- aufwändige, komplexe Deployments
- Engpässe bei der Kapazität der IT-Systeme und schwankende Verfügbarkeiten

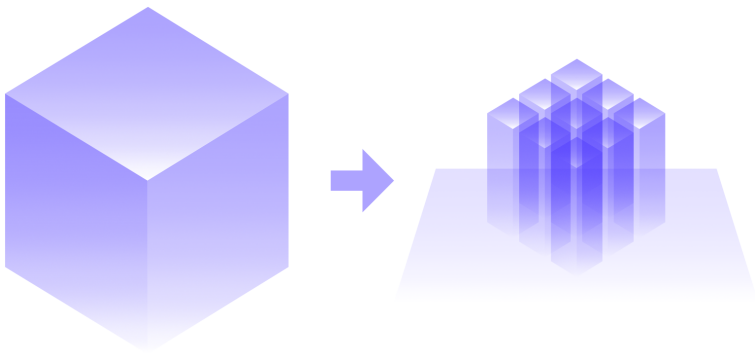

Abbildung 1.1: Ein monolithisches System wird in SCS zerlegt

Was bedeutet es, „Self-contained“ zu sein?

Self-contained bedeutet auf Deutsch etwa so viel wie „eigenständig“ oder „in sich abgeschlossen“.

> Ein Self-contained System ist im Kern eine Webanwendung, die einen geschlossenen, weitgehend unabhängigen Teil eines komplexen fachlichen Problems löst. Ein SCS besteht immer aus der jeweiligen Geschäftslogik, der zugehörigen Datenhaltung und der notwendigen Nutzeroberfläche.

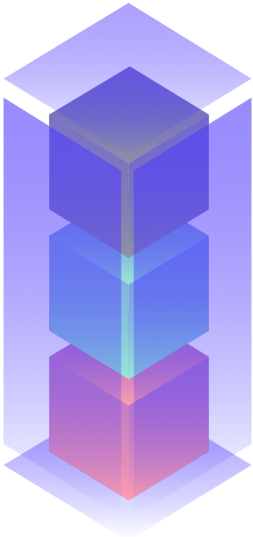

Abbildung 1.2: Eigene DB, Eigene Business-Logik, Eigene UI

Für Anwender:innen präsentieren sich verschiedene SCS dennoch als kohärentes Ganzes, dessen Modularisierung den Nutzer:innen kaum auffällt. Gemeinsam verwendete Styleguides und UI-Komponenten sorgen dafür, dass die Grenzen zwischen den SCS den Nutzer:innen kaum auffallen[1]. Auf Datenbankebene sind SCS in Form von eigenen Schemata[2] (mindestens logisch) voneinander getrennt und können über die Änderungen und Erweiterungen ihrer eigenen Datenbankschemata selbstständig bestimmen. Eine gemeinsame Nutzung von Infrastruktur sollte auf ein notwendiges Minimum reduziert werden.

Unabhängig von der Idee bis in die Produktion

Beim Entwurf von SCS wird dabei vor allem Wert auf die Unabhängigkeit der Systeme gelegt.

> Jedes SCS soll von einem kleinen Softwareentwicklungsteam (typisch sind ca. 5 bis 9 Personen) entwickelt und deployt werden können. Typische Erweiterungen können ohne Änderungen an anderen SCS durchgeführt werden. Statt sich mit anderen Teams zu koordinieren, arbeitet das Team eines SCS so weitgehend unabhängig.

Durch einen fachlich getriebenen „Schnitt“ der Systemgrenzen kann sich jedes Team auf einen Teil einer umfangreichen Fachdomäne fokussieren und spezialisieren. Wichtige Voraussetzung dafür ist, dass die SCS technisch sowie fachlich möglichst lose untereinander gekoppelt sind. Das bedeutet, dass eine Änderung in einem SCS nur in wenigen Fällen auch zwingend eine Änderung in anderen SCS erforderlich macht. Im Kapitel „Domänenarchitektur“ werden Ansätze vorgestellt, wie fachlich komplexe Systeme in ihre fachlichen Bestandteile zerlegt werden können.

Durch die lose Kopplung reduziert sich der Abstimmungsaufwand zwischen verschiedenen Softwareentwicklungsteams deutlich. Auch die Notwendigkeit, „Re-

[1] Weitere Details hierzu im Kapitel „UI-Integration“.

[2] Dabei ist es unerheblich, ob die Schemata explizit durch eine Data Definition Language (DDL) definiert sind wie bei vielen relationalen Datenbanken oder implizit wie bei den meisten NoSQL-Datenbanken.

lease Trains“ und einen „Gesamtintegrationstest“ zu verwenden, ist nicht mehr zwingend. Eine Koordination von Releases genügt bei inkompatiblen Änderungen an öffentlichen Schnittstellen und erfolgt auch dann nur zwischen beteiligten SCS-Teams. Hierdurch ermöglicht der Architekturstil eine hohe Releasefrequenz bis hin zu mehrmals täglichem Deployment, wie bei Continuous Delivery üblich.

Unabhängig betreibbar

Die Unabhängigkeit eines SCS endet aber nicht damit, dass es nur unabhängig entwickelt und deployt werden kann.

> Ein SCS kann weitgehend unabhängig von anderen SCS betrieben werden.

Der Architekturstil empfiehlt dafür eine Reihe von Integrationsmustern (genauer beschrieben im Kapitel Integrationsmuster), die konsequent angewandt dafür sorgen, dass Systeme ein Minimum an Laufzeitabhängigkeiten untereinander haben. Fallen einzelne oder mehrere SCS aus, können die verbleibenden SCS in vielen Fällen weiterhin ihren Zweck erfüllen.

Architekturentscheidungen in Self-contained Systems

Bei monolithischen Architekturen ist die gemeinsame technische Basis ein Zwang. Alle müssen die gleichen Versionen von Libraries, Programmiersprachen, Laufzeitumgebungen etc. verwenden. Das macht sie nicht nur sehr unflexibel, wenn es besser passende Lösungen für eine bestimmte fachliche Herausforderung gibt, es sorgt auch dafür, dass eine Modernisierung zum erheblichen Unterfangen wird, da oft die gesamte monolithische Anwendung als Ganzes angepasst werden muss.

> SCS lassen einzelnen Teams die Freiheit, die beste Technologie für ein fachliches Problem zu wählen und Modernisierungen in kleineren Einheiten durchzuführen.

Vielleicht eignet sich *Ruby on Rails* besonders gut für einen stark datenbankgetriebenen Teil der Anwendung, während für einen anderen Teil die Implementierung mit Java und Spring die bessere Wahl wäre.

Wie viel Entscheidungsfreiheit eine Organisation ihren Teams lässt, kann sich unterscheiden. Die Möglichkeit, Dinge (neue Technologien, Ansätze, etc.) erst mal in einem eingeschränkten Kontext auszuprobieren, und später gegebenenfalls auf weitere Systeme auszurollen, ist wertvoll.

Diese Freiheiten sollen aber nicht zu einem Wildwuchs an inkompatiblen Ansätzen und Technologien führen, der Betrieb und Entwicklung überfordern würde. Aus diesem Grund ist es wichtig, die Randbedingungen, die für alle SCS gelten sollen, klar abzustecken. Solche verbindlichen Vorgaben, die für alle SCS gelten, nennen wir auch die „Makroarchitektur“ eines Systems. Beispiele sind die Festlegung auf gemeinsame Integrationstechniken, Security-Protokolle, zentralisiertes Logging und Monitoring, etc. Aber auch Styleguides oder sogar fachliche Meldepflichten können ein Teil der Makroarchitektur einer SCS-Landschaft sein. Alle nicht dort verbindlich getroffenen Architekturentscheidungen sind „Mikroarchitektur“ und obliegen den einzelnen SCS-Teams. Die Einteilung in Mikro- und Makroarchitektur wird im Kapitel Mikro- und Makroarchitektur genauer beleuchtet.

Die Eigenschaften eines SCS

Auf der offiziellen Webseite scs-architecture.org werden Eigenschaften gesammelt, die ein Self-contained System haben muss. Zum Zeitpunkt der Veröffentlichung dieses Primers sind dort acht Eigenschaften gelistet. Ein SCS ist demnach:

- Eine autonome Webapplikation
- Durch ein Team verantwortet
- Überwiegend asynchron integriert
- Optional durch eine Service API integrierbar
- Mit eigener Businesslogik und Datenhaltung versehen
- Mit eigener UI versehen

- Ohne geteilte Geschäftslogik
- Mit minimaler geteilter Infrastruktur betrieben

Diese Eigenschaften sind jedoch nicht „in Stein gemeißelt“. Die Inhalte der Webseite werden von einer Community auf GitHub aktiv diskutiert und werden durch Diskussionen getrieben. Die sehr klare Definition unterscheidet in unseren Augen jedoch Self-contained Systems stark von anderen Architekturstilen.

Sind SCS nicht das Gleiche wie Microservices?

Was uns bewegt hat, den Begriff „Self-contained Systems“ zu prägen, ist die Beobachtung, dass es keine allgemein akzeptierte Definition eines „Microservices“ gibt. Vielmehr gibt es viele verschiedene Definitionen und Interpretationen des Begriffes, denen lediglich gemeinsam ist, dass sie ein unabhängiges Deployable beschreiben. Wenn wir von „Microservices“ sprechen, meinen wir deshalb den kleinsten gemeinsamen Nenner von allem, was sich als „Microservice“ bezeichnen lässt, auch wenn es Definitionen gibt die bestimmte Kriterien anders sehen würden.

Der Microservice-Architekturstil und Self-contained Systems sind sich auf den ersten Blick ähnlich. Beide Architekturstile verfolgen eine Modularisierung, bei der große, monolithische Systeme in kleinere Bausteine zerlegt werden, die unabhängig voneinander deployt werden können. Dennoch sehen wir einige Unterschiede zwischen dem SCS-Ansatz und Microservices.

Microservices sind in der Regel kleiner als ein SCS[3]. SCS sind so dimensioniert, dass sie ein Team von typischerweise bis zu 9 Personen ohne Überlastung entwickeln können. Es wird bei gleicher Fachdomäne deshalb in der Regel weniger SCS als Microservices geben. Ein E-Commerce-Shop hat je nach Komplexität vielleicht 5 bis 25 SCS, könnte aber aus hunderten Microservices bestehen.

[3] Das trifft insbesondere dann zu, wenn ein naives Verständnis des Begriffs eine Organisation dazu bewegt, besonders kleine Microservices (im Sinne von wenigen Lines of Code) zu bauen. Andere Definitionen, wie die von Sam Newman geprägte „Ein unabhängiges System, das um eine fachliche Domäne herum modelliert wurde“ münden eher in ähnlich großen Deployables.

SCS haben immer auch eine eigene Nutzeroberfläche, während eine API als optional angesehen wird. Microservices haben hier keine Präferenz, haben aber deutlich häufiger APIs, die dann von einem (zentralisierten) Frontend konsumiert werden.

Bei der Integration von SCS definiert der Architekturstil außerdem klarere Präferenzen (oder „Best Practices“) für bestimmte Integrationstechniken, während sich der Microservices-Ansatz hierzu nicht für eine Präferenz ausspricht.

	Microservices	Self-contained Systems
Lose Kopplung	Angestrebt	Zwingend
Unabhängig entwickelbar	Ja	Ja
Unabhängig deploybar	Ja	Ja
Fachliche Domänenarchitektur	Unterstützt, nicht obligatorisch	Obligatorisch
Eigene UI	Unterstützt (Microfrontends), nicht obligatorisch	Zwingend, unabhängig
Eigene Schemata	Keine Vorgabe	Obligatorisch
Integration	Keine Empfehlung	Best Practices
Anzahl und Größe	Typischerweise viele, kleinere	Typischerweise weniger, größere

Beide Ansätze schließen sich nicht grundsätzlich aus. So kann ein SCS intern bei spezifischen Anforderungen aus (häufig eher technisch geschnittenen) Microservices bestehen.

Der SCS-Ansatz konzentriert sich primär auf größere Projekte und eine fachliche Aufteilung von Software in Module, die durch mehrere Teams entwickelt werden können - Microservices können hingegen auch für andere Zwecke eingesetzt

werden: Häufig nutzen kleine Teams oder sogar einzelne Entwickler:innen sie, z. B. um Continuous Delivery zu erleichtern, robustere Systeme zu entwickeln oder jeden Microservice unabhängig zu skalieren. Microservices sind also vielseitiger, während SCS speziell Probleme im Zusammenhang mit der Architektur und Organisation großer Projekte lösen.

> Eine Modularisierung in Self-contained Systems ist insbesondere dort ratsam, wo die Probleme von begrenzter Skalierbarkeit und langsamen Test- und Entwicklungszyklen und/oder aufwändigen Deployments die Evolution eines stark wachsenden Softwaresystems einschränken.

Haben wir uns für den SCS-Ansatz entschieden, stellt sich unmittelbar die nächste Frage: Wo verlaufen die Grenzen zwischen den Systemen? Die Antwort darauf liefert keine technische Analyse, sondern der Blick in die Fachlichkeit. Diese behandelt das nachfolgende Kapitel 2: Domänenarchitektur.

2 Domänenarchitektur

Eine Domänenarchitektur entwerfen

Möchten wir fachlich unabhängige Systeme entwerfen, müssen wir zunächst einen tiefen Blick in die Fachlichkeit und deren Zusammenhänge werfen.

> Erst wenn wir die Komplexität des Problemraumes durchdrungen haben, können wir qualifizierte Entscheidungen treffen, welche Teile des IT-Systems fachlich zusammengehören und wie die einzelnen Systeme zusammenarbeiten. Wir sprechen hier auch von der Domänenarchitektur, also der Architektur, wie Systeme fachlich aufgebaut sind und wie sie zusammenarbeiten.

Zur Unterstützung dieses Lernprozesses hat die Domain-driven Design Community in den 2010er Jahren verschiedene Techniken wie beispielsweise das hier vorgestellte „Big Picture Event Storming“ entwickelt (siehe auch [1]). Fachbereich und IT entwickeln dabei kollaborativ ein visuelles Modell der Fachdomäne, an dem die Zusammenhänge studiert, analysiert und gegebenenfalls verbessert werden können.

Das folgende Bild zeigt das Ergebnis einer beispielhaften Big Picture Event Storming Session zu einer Baufinanzierungs-Domäne, wie sie vielleicht in einer Direktbank stattgefunden haben könnte. Dabei haben Fachbereich und IT von der Antragstellung über das Scoring des Antrags, Antragsprüfung und Kreditentscheidung verschiedene Schritte und alternative Abläufe des fachlichen Prozesses gemeinsam modelliert und diskutiert. Die quadratischen orangefarbenen Klebezettel entsprechen Ereignissen („Domain Events“), die an einer Zeitleiste angeordnet werden. Die rechteckigen violetten Klebezettel entsprechen einem externen System, das wir wie eine „Black Box“ verwenden. Auf Grundlage des Modells können wir nun gemeinsam mit dem Fachbereich beginnen, die Funktionen zu „clustern“, um fachliche geschnittene Bausteine zu bilden.

Eine Anleitung, die beschreibt, wie man auf Basis eines solchen Event Stormings fachliche Grenzen identifizieren kann, gibt Alberto Brandolini im Artikel „Discovering Bounded Contexts with EventStorming“[2]. An dieser Stelle wird, un-

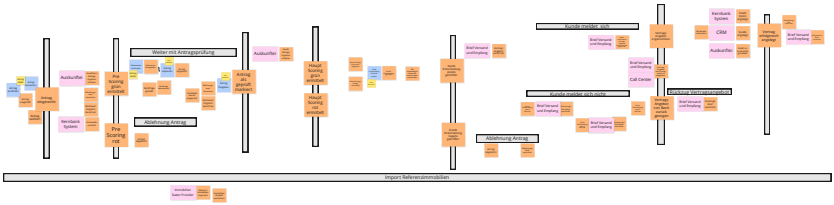

Abbildung 2.1: Event Storming Beispiel - mit freundlicher Genehmigung von Michael Plöd

abhängig von der Technik, mit der diese identifiziert wurden, auf wesentliche Eigenschaften fachlicher Bausteine eingegangen, die für Self-contained Systems unerlässlich sind.

Die Facharchitektur bewerten

Ein Resultat unserer gemeinsamen Arbeit ist eine mögliche Zerlegung unserer Fachdomäne in unabhängige Bausteine. Im oben genannten Beispiel ist ein solcher Baustein vielleicht das „Scoring“, das einen Antrag auf eine Baufinanzierung hinsichtlich der Genehmigungsfähigkeit bewertet. Das folgende Bild illustriert eine solche mögliche Aufteilung sowie die fachlichen Zusammenhänge zwischen den Bausteinen.

> Aber woran können wir erkennen, dass unsere Aufteilung wirklich gut ist und eine lose Kopplung zur Folge hat? Wie bereits erwähnt spielen die Schnittstellen zwischen den Systemen hier eine wesentliche Rolle. Nur wenn diese so gestaltet sind, dass sie sich selten inkompatibel ändern, kann ein SCS unabhängig vom anderen geändert werden.

Diese Schnittstellen wie das Event „Pre-Scoring war Grün“ aus Bild 2.2 enthalten Informationen darüber, welches Ergebnis erarbeitet wurde, aber verbergen, wie genau das Ergebnis zustande kam. Statt uns von der Reihenfolge, in der Dinge im fachlichen Prozess passieren könnten, ablenken zu lassen, haben wir das „Information Hiding“ als Entwurfsprinzip[3] angewendet, um kohäsive, fachliche Einheiten zu erhalten und ihnen einen Namen zu geben. An den Schnittstellen entstehen dadurch fachliche Abstraktionen, die es uns erlauben eine beliebige

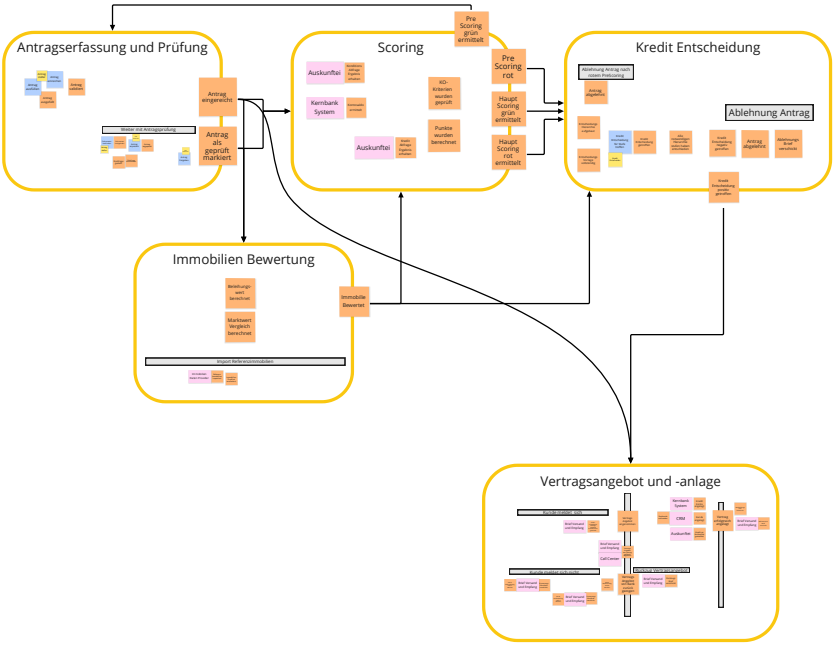

Abbildung 2.2: Mögliche Aufteilung der Fachdomäne in Bausteine

Änderung an der Implementierungsdetails des Scorings vorzunehmen, solange die Schnittstelle stabil bleibt.

Leider lassen sich gute, fachliche Abstraktionen nicht einfach „mechanisch“ aus den Modellen ableiten. Es gibt jedoch einige gute Heuristiken. Alberto Brandolini bezeichnet Ereignisse, die einen signifikanten Statusübergang markieren (zum Beispiel vom Interessenten zum Kunden) als „Pivotal Events“. Diese Ereignisse haben sich in vielen Fällen als gute Kandidaten für Schnittstellen herausgestellt. Manche fachliche Abstraktionen, wie beispielsweise eine Immobilienbewertung in Form einer Berechnung von Markt- und Beleihungswert auf Basis der importierten Referenzimmobilien, lassen sich aber nicht durch die reine Betrachtung der Pivotal Events identifizieren. Sie sind aber eventuell für unsere Domänenexpert:innen offensichtlich.

Die Erstellung einer lose gekoppelten Domänenarchitektur ist ein iterativer Prozess nach dem Prinzip „Trial and Error".

Eine umfangreiche Einführung in die Domänenmodellierung für lose gekoppelte Systeme bietet der Primer „Modernes Domain-driven Design" von Michael Plöd [4].

Schnittstellen für unabhängige Systeme

Um die Unabhängigkeit zu erreichen, die wir uns von SCS versprechen, ist kontinuierliche Analyse und aktives Management der Beziehungen zwischen den SCS nötig. Dabei ist ein Verständnis für die *fachliche* Kopplung zwischen den Systemen von oberster Priorität.

Eine fehlende Berücksichtigung von fachlicher Kopplung bei der Modularisierung äußert sich in der Praxis dadurch, dass kaum eine fachliche Erweiterung oder Änderung an *System A* möglich ist, ohne dass auch eine entsprechende Erweiterung in *System B* vorgenommen werden muss.

Gerade in historisch gewachsenen Systemlandschaften führt die starke fachliche Kopplung unabhängiger Belange zu komplexen Projektplänen, langen Releasezyklen und vielen Abstimmungsaufwänden zwischen Teams.

In IT-Systemen entsteht eine stark gekoppelte Systemlandschaft schnell, wenn fachliche Zusammenhänge wie Wertströme[1] nicht betrachtet und stattdessen technische (z. B. das ist eine GUI, das ist ein Backend) oder oberflächliche Gemeinsamkeiten (z. B. alle Datensätze, die über eine Postanschrift verfügen) als wichtigstes Kriterium für die gemeinsame Paketierung von Funktionen herangezogen werden.

Doch selbst wenn eine fachliche Paketierung das Ziel ist, können uns technische „Best Practices" wie die Reduktion von Datenredundanzen auf ein Minimum zu starker fachlicher Kopplung führen. Fachliche Änderungen können durch gemeinsam verwendete Strukturen wie ein globales Adressdatenmodell schnell zu

[1] Ein Wertstrom in der IT beschreibt den gesamten Weg, den eine Idee bis zur nutzbaren Software durchläuft – also alle Schritte und Änderungen an IT-Systemen vom Bedarf bis zum fertigen Ergebnis.

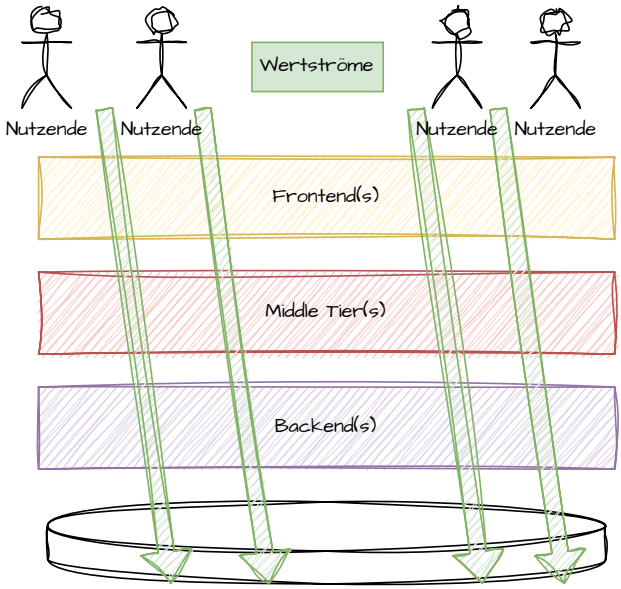

Abbildung 2.3: Wertströme durch eine nach technischen Kriterien paketierte Architektur

unerwarteten Nebenwirkungen führen. Durch die Einführung des Versands an Packstationen könnte es passieren, dass plötzlich Rechnungen auf die Adresse einer Packstation ausgestellt werden oder wegen fehlender Straße und Hausnummer nicht mehr korrekt gedruckt werden. Selbst wenn die Rechnungsstellung unabhängig von der Logistik entwickelt wurde, kann sich durch die gemeinsame Nutzung von Datenmodellen (in unserem Beispiel eine gemeinsam genutzte Adresse) eine starke Kopplung einschleichen.

Um diese Effekte zu verhindern (etwas an Fachlichkeit A wird geändert, das produziert in Fachlichkeit B unerwartete Seiteneffekte), kann bei der Releaseplanung eine genaue Analyse aller Abhängigkeiten zu den gemeinsam genutzten Artefakten durchgeführt werden. Bei stetig wachsenden Systemen wird diese Analyse jedoch immer aufwändiger und auch die Lösungen für mögliche Anforderungskonflikte

im gemeinsamen Modell immer komplexer. Wir befinden uns auf dem Weg, ein kanonisches Datenmodell[2] durchzusetzen, ohne es so zu nennen.

Eine bessere Alternative ist es, jedem SCS eigene, unabhängige Modelle zuzugestehen - eine Idee, die im Domain-driven Design durch die Aufteilung der Fachlichkeit in separate „Bounded Contexts“ mit eigenen Modellen erreicht wird. Dabei entsteht eventuell Duplikation von Daten und Funktionen, die wir bewusst hinnehmen. Dinge, die vorher nicht synchronisiert werden mussten, weil sie gemeinsam genutzt waren, müssen nun synchron gehalten werden. Diese Duplikation ist ein Preis, der für eine höhere Unabhängigkeit zu zahlen ist.

Wer unabhängige Systeme bauen möchte, sollte den Fokus deshalb unbedingt weg von Datenmodellierung und auf die Verwaltung der Schnittstellen legen, denn die Unabhängigkeit der Systeme entscheidet sich nicht mit der Paketierung von Code in Bausteine oder Services, sondern an deren Schnittstellen[3].

Die Daten- und die Prozessfalle vermeiden

Vielleicht hat es Sie überrascht, dass in Diagramm 2.2 keine SCS für Stammdaten oder zentrale Entitäten wie „Kunde“ oder „Vertrag“ auftauchen.

> In Systemen, die eine starke Unabhängigkeit anstreben, hat sich der Fokus auf gemeinsam genutzte Daten als Kriterium für den Modulschnitt eher als ein Anti-Pattern herausgestellt.

Daten sind für operative Systeme nur Hilfsmittel zur Erfüllung eines fachlichen Anwendungsfalls. Dabei gibt es viele Informationen, die einen fachlichen Kontext gar nicht verlassen müssen, da sie nur für eine bestimmte Fachlichkeit relevant sind.

---

[2] Stefan Tilkov - Why You Should Avoid a Canonical Data Model https://www.innoq.com/de/blog/2015/03/thoughts-on-a-canonical-data-model/

[3] Johannes Seitz - API gut, alles gut. https://www.innoq.com/de/articles/2024/06/api-schnittstellen-unabhaengige-systeme/

Wo Daten von mehr als einem SCS verwendet werden, sollten sie besser bei Bedarf und nur so weit wie nötig ausgetauscht werden. Häufig benötigte Entitäten können hierzu über eine gemeinsam genutzte ID „virtuell“ miteinander verknüpft und zwischen SCS ausgetauscht oder synchronisiert werden. Sie können z. B. für dispositive Auswertungen ihre Daten weiterhin zentral zusammen sammeln, etwa in einem klassischen Data Warehouse. Alternativ existiert mit dem Data Mesh Ansatz[5] eine moderne, dezentrale Alternative, die perfekt zur SCS-Architektur passt.

Wir möchten Systeme bauen, die um den fachlichen Nutzen zentriert sind und alles dafür Benötigte beinhalten. Ohne Kundendaten, Produktdaten etc. haben Sie kein vollständiges System. Zentrale, breit wiederverwendete Services haben marginale Vorteile, werden aber durch die starke Kopplung, die sie verursachen, schnell zum Flaschenhals für Änderungen.

Neben dieser „Datenfalle“ besteht auch ein starker Instinkt, einzelne „Prozessschritte“ naiv als Systemgrenzen zu gebrauchen. Bei der „Prozessfalle“ werden Systeme nach den Schritten in einem Prozess „geschnitten“. Parnas nennt diesen Effekt auch den „Flowcharting Instinct“, weil die Modulgrenzen in diesem Fall aus Boxen eines Flussdiagramms abgeleitet werden[6].

Dabei werden insbesondere Dinge die an vielen Stellen im Prozess passieren können, stark verteilt und/oder mehrfach implementiert. Und gerade bei stark iterativen oder kollaborativen Prozessen werden Schnittstellen unnötig breit. Beides macht die Systeme schwer zu ändern.

Er schlägt stattdessen die Frage vor: Wenn sich eines der Implementierungsdetails ändern würde, wie könnte man dieses Detail so verstecken, dass Systeme von der Änderung nichts mitbekommen würden. Den Schnitt auf Einhaltung des „Information Hiding“ Prinzips hin zu prüfen hilft hier, nicht in die „Prozessfalle“ zu tappen.

Haben wir unsere Domäne in fachlich eigenständige Bausteine zerlegt und deren Schnittstellen sorgfältig gestaltet, stellt sich die nächste Frage: Wie viel Freiheit bekommt jedes Team bei der Umsetzung — und wo brauchen wir verbindliche, systemübergreifende Regeln? Diese Frage beantwortet die Unterscheidung zwischen Mikro- und Makroarchitektur, die wir im nächsten Kapitel behandeln.

3 Self-contained Systems und die Organisation

Für viele Firmen bedeutet die Einführung der Self-contained-Systems-Architektur gleich an mehreren Stellen einen Bruch mit der Art und Weise, wie ihre IT bisher entwickelt wurde. Die Veränderung der Architektur muss daher von einer Veränderung in der Organisation begleitet werden.

Stefan Tilkov hat seine Beobachtungen dazu einmal so formuliert:

> Tilkov's Law: In einer digitalisierten Firma können sich Architektur, Organisation und Prozesse immer nur zusammen weiterentwickeln.[a]
>
> [a] Im Original: In a digital company, architecture, organization & processes can only evolve together [7].

Aus diesem Grund drängen sich mehrere Fragen auf: Wie organisieren sich Teams um ein SCS? Wie kann eine solche Veränderung gelingen? Und was passiert mit querschnittlichen Themen, die nicht einem einzelnen SCS zuzuordnen sind?

Softwareentwicklungsteams und SCS

Firmen, die in Self-contained Systems investieren, wünschen sich von ihrer IT eine schnelle Time-to-Market und dadurch einen Wettbewerbsvorteil. Statt eines Gesamtreleases zwei- bis viermal im Jahr soll jederzeit eine Erweiterung und Anpassung einzelner SCS möglich sein. Manche Firmen schaffen es sogar, morgens eine Idee auszuhecken und diese am Abend des gleichen Tages bereits mit richtigen Kunden in Produktion zu testen. Wie schaffen die das?

Ein Großteil der Antwort wurde bereits im Buch Continuous Delivery[8] gegeben. Grundvoraussetzung hierfür ist ein hoher Grad an Automatisierung in Test und Deployment. Ein solch hoher Grad an Automatisierung macht die schnelle Auslieferung in Produktion zu einem geringen Risiko. Eine weitere technische Voraussetzung wurde in der „State of DevOps"-Studie (auch bekannt geworden durch das Buch „Accelerate" [9]) gegeben: eine lose gekoppelte Architektur ist der wesentliche Faktor, der bestimmt, wie schnell und wie oft fachliche Änderungen

in Produktion gebracht werden können. Intuitiv ist das auch einleuchtend, denn starke Kopplung führt dazu, dass ein Team alleine keine wesentlichen Änderungen am System vollziehen kann, ohne andere Teams miteinzubeziehen. Wie wir diese lose Kopplung herstellen, haben wir im letzten Kapitel „Domänenarchitektur“ gesehen.

In einer zur SCS-Architektur passenden Organisation hat ein Softwareentwicklungsteam alles, was es benötigt, um den Softwarelebenszyklus von der Feature-Idee bis zu der Produktivsetzung des Features abzudecken. Dabei können je nach Firma verschiedene bestehende Silos in Frage gestellt werden. Hat Ihre Firma beispielsweise ein separates Datenbank-Team für die Definition von DB-Schemata, ist dies für die Funktionserweiterung eines SCS eine inakzeptable Hürde, die abgebaut werden muss. Ist ein separates Testteam für das Testing der Software zuständig, ist hier ebenfalls Handlungsbedarf. Ob die fehlende Kompetenz durch Personalverschiebungen, Matrixorganisation oder durch eine Schulung erreicht wird, hängt dabei von den Möglichkeiten und Randbedingungen der Organisation ab. Die Stoßrichtung ist jedoch immer die gleiche: Alle benötigten Fähigkeiten und Berechtigungen sollten im Idealfall Teil des Softwareentwicklungsteams eines SCS sein. Eine Abhängigkeit von anderen Teams sollte für die tägliche Entwicklung und Auslieferung des SCS die absolute Ausnahme sein. Coding Agents können hier eine beschleunigende Wirkung haben sowie fehlende Spezialisierungen zu einem Teil abdecken.

Sind die Silogrenzen in der IT erst einmal passé, ist die Auslieferung nur noch durch die Geschwindigkeit von der Idee zur Umsetzung beschränkt. Es bleibt aber noch eine große Chance für weitere Optimierung: die Grenzen zwischen Fachbereich und IT. Durch das tiefe Verständnis der Fachlichkeit und eine gemeinsame Sprache (im DDD nennen wir das „Ubiquitous Language“) wird es Entwicklungsteams ermöglicht, fachliche Funktionswünsche nicht nur schneller zu verstehen, sondern auch in der direkten Zusammenarbeit kreative Lösungen zu finden, die technisch und fachlich exzellent sind. Ein so um die Fachlichkeit einer Subdomäne organisiertes Team aus Technikern und Fachbereich kann sehr schnell iterieren. Wie in Kapitel 1 beschrieben, würde dieses kleine Team für ein einzelnes SCS zuständig sein, das diese Subdomäne komplett abbildet.

Die Veränderung meistern

Der Umstieg auf SCS beginnt selten mit Komplettumbau, sondern mit einem einzelnen, überschaubaren Experiment. Statt das bestehende System in einem großen Wurf zu zerlegen, schneidet ein Team einen fachlich klar abgegrenzten Bereich heraus – einen Bounded Context, der nur lose am Rest hängt – und baut daraus ein erstes eigenständiges System. Dieser „Carve-out“ ist vor allem eine Gelegenheit zu lernen: An ihm klären sich Fragen, die vorher nur auf dem Papier beantwortet waren. Wie verteilen wir geteilte Daten? Wie binden wir das neue System in die bestehende Oberfläche ein? Wie gehen wir mit den Abhängigkeiten um, die erst beim Herauslösen sichtbar werden? Die Wahl des ersten Kandidaten entscheidet viel: Er sollte wichtig genug sein, dass ein Erfolg auffällt, aber nicht so kritisch, dass ein Scheitern das Unternehmen gefährdet.

Ein Carve-out ist nie nur eine technische Übung, denn SCS gleichen den Schnitt des Systems an die fachlichen Grenzen der Organisation an. Jedes SCS gehört schließlich genau einem Team, das es unabhängig entwickelt und betreibt. Dadurch verschieben sich neben Systemgrenzen auch Verantwortungsgrenzen. Damit dieser Wandel gelingt, braucht er mittelfristig begleitende Mechanismen: beispielsweise eine kleine Gruppe, die den Umbau über Teamgrenzen hinweg steuert und auf der technischen Ebene zentrale Qualitätsmerkmale – etwa unabhängige Deploybarkeit oder den Verzicht auf geteilte Datenbanken – prüft.

Für den Übergang vom ersten Experiment zum flächendeckenden Umbau beschreibt Sam Newman in „Monolith to Microservices“[10] einen gut passenden Ansatz. Sein Leitgedanke ist eine inkrementelle Migration: nicht der große Rewrite, sondern viele kleine, jeweils in Produktion gebrachte Schritte, aus denen die Organisation lernt, bevor sie den nächsten geht. Das Herausschneiden selbst beschreibt er mit etablierten Mustern – die „Strangler Fig Application“, die Aufrufe nach und nach vom Altsystem auf das neue umlenkt, bis der alte Kern verkümmert; Branch by Abstraction für Fälle, in denen die abzulösende Fachlichkeit tief im Bestand steckt und sich nicht am Rand abgreifen lässt; und den Parallel Run, der altes und neues Verhalten eine Zeit lang nebeneinander betreibt, um Vertrauen aufzubauen, bevor umgeschaltet wird.

Besonders wichtig für SCS ist auch Newmans ausführliche Behandlung der Datenbank-Zerlegung, denn der Datenschnitt entscheidet über die Unabhängigkeit eines Systems mehr als jede andere Einzelentscheidung. Eine genauere Beschreibung dieser Migrationsstrategie mit Fokus auf SCS findet sich auch im Artikel „Von Legacy-Monolithen zu Self-contained Systems“[11].

4 Mikro- und Makroarchitektur

Für SCS empfehlen wir gemäß den ISA Principles[1] die Aufteilung der Architekturarbeit in zwei Ebenen: die Mikro- und die Makroarchitekturebene.

> Die Makroarchitektur besteht aus Entscheidungen und Prinzipien, die für alle SCS der Systemlandschaft gelten. Die Mikroarchitektur besteht aus Entscheidungen, die je SCS unterschiedlich getroffen werden können.

Bei der Frage, welche Entscheidungen zur Makro- und welche zur Mikroarchitektur gehören, betreten wir ein Spannungsfeld. Auf der einen Seite steht dabei der Wunsch nach den Vorteilen von Standardisierung und Harmonisierung und auf der anderen Seite die Freiheit, passende Tools für das im Team vorhandene Know-how sowie passende Lösungen für die fachlichen Herausforderungen einer bestimmten Domäne nutzen zu können.

Durch die gemeinsame Makroarchitektur werden alle SCS stark aneinander gekoppelt, denn eine Änderung an der gemeinsamen Makroarchitektur erfordert eine Änderung an allen SCS. Diese starke Kopplung ist nur dann akzeptabel, wenn sie mit einer hohen Stabilität bzw. sehr niedrigen Änderungsfrequenz einhergeht. Diese Entscheidungen sind also für eine lange Zeit getroffen. Zudem sollte so eine Aufnahme einer Entscheidung in die Makroarchitektur wohlüberlegt sein.

Häufig werden Makroarchitekturentscheidungen in einem Gremium getroffen, in das aus jedem betroffenen Team einzelne Stakeholder entsendet werden. Dieses Muster sorgt im Idealfall für eine breite Akzeptanz der getroffenen Entscheidungen. Je nach Firmenkultur kann eine Makroarchitektur hohe Freiheitsgrade für einzelne Teams lassen, oder einen hohen Grad an Vereinheitlichung auf der Mikroarchitekturebene vorschreiben. Im Zweifel würden wir uns für einen hohen Grad an Freiheit aussprechen, da dadurch passgenaue Architekturlösungen, Experimente mit eingeschränktem Risiko und die inkrementelle Migration bzw. Adoption neuer Technologien unterstützt werden.

In den folgenden Abschnitten werden gängige Aspekte einer gemeinsamen Makroarchitektur diskutiert.

[1] https://isa-principles.org/

Betriebs- und Deploymentumgebung

Self-contained Systems werden häufig in einer standardisierten Laufzeitumgebung betrieben, die eine entkoppelte Bereitstellung und Skalierung ermöglicht. Ob die Ausführung auf Containern oder virtuellen Maschinen erfolgt sowie ob die Infrastruktur als IaaS oder PaaS bereitgestellt wird, ist oftmals eine bewusste Makroarchitekturentscheidung und gilt für alle SCS. Der Betrieb kann sowohl in Cloud-Umgebungen als auch On-Premises erfolgen, sofern die gewählte Plattform die Anforderungen an Isolation, Automatisierung und unabhängige Deployments erfüllt.

> Eine Makroarchitektur definiert häufig die Betriebs- und Deploymentumgebung als verbindliche Rahmenbedingungen. Ziel ist es, einen stabilen, interoperablen und unabhängig betreibbaren Systemkontext für alle Self-contained Systems zu schaffen.

Ergänzend werden Observability-Standards vorgegeben, die sicherstellen, dass alle Systeme konsistent über Logs, Metriken und Traces beobachtbar sind. Dazu zählen einheitliche Konventionen für strukturierte Logs, Metrik-Exposition sowie die Verwendung von Correlation IDs, um End-to-End-Nachvollziehbarkeit über Systemgrenzen hinweg zu ermöglichen.

Damit stellt die Makroarchitektur sicher, dass Betrieb und Deployment vereinheitlicht, automatisierbar und resilient sind, während die technologische Entscheidungsfreiheit innerhalb der Systemgrenzen erhalten bleibt.

Technologie- und Querschnittsstandards

SCS erlauben es, viele technologische Entscheidungen wie die Programmiersprache und Laufzeitplattform (z. B. Java/JVM oder C#/.NET) auf die Mikroarchitekturebene zu legen. Solange alle SCS die definierten Schnittstellen-, Sicherheits- und Betriebsstandards einhalten, führt diese Heterogenität nicht zu Problemen der Kompatibilität oder Interoperabilität. Gleiches gilt für Persistenztechnologien: Self-contained Systems besitzen ihre eigene Datenhaltung und können je

nach fachlicher und nicht-funktionaler Anforderung frei relationale oder NoSQL-Datenbanken einsetzen, ohne dass andere SCS durch diese Wahl beeinträchtigt werden.

> Eine Makroarchitektur definiert einen Satz verbindlicher Technologie- und Querschnittsstandards, deren Zweck es ist, Interoperabilität, Sicherheit und Betriebsfähigkeit über alle SCS hinweg sicherzustellen.

Wir empfehlen hier, Vorgaben auf ein notwendiges Minimum zu beschränken, um gute Lösungen für einzelne Domänen nicht ohne Not auszuschließen. Beispielsweise könnte die Vorgabe, alle Daten in relationale Schemata zu legen, eine Verwendung viel passenderer Technologien wie einer Graphendatenbank für eine Teilanwendung, die sich mit Beziehungen unter Personen beschäftigt, verhindern.

In vielen Systemen gibt es Anforderungen, die sich querschnittlich durch alle SCS ziehen. Die Anforderungen an Betrieb, Auslieferung und Observability des Systems (Logging, Monitoring etc.) wurden im vorherigen Absatz bereits angeschnitten. Eine weitere Querschnittsanforderung ist in vielen Systemen das Thema Security - beispielsweise in Form von Anforderungen an Authentifizierung und Autorisierung (AuthN/AuthZ), den sicheren Umgang mit Secrets sowie den geschützten Transport von Daten (z. B. TLS, mTLS).

Häufig lassen sich für querschnittliche Anforderungen offene Standards finden, die zwar eine Schnittstelle definieren, aber keine bestimmte Library zur Implementierung vorschreiben. So könnte OAuth2 in der Makroarchitektur als Protokoll zur Authentifizierung vorgeschrieben werden, ohne dass dafür jede Anwendung die gleiche Library zur Authentication einsetzen muss. Diese Trennung von Schnittstellen und Implementierung ist der Schlüssel zur Technologieoffenheit der Systeme.

Schnittstellen und Systemintegration

Dies gilt insbesondere auch für die Schnittstellen zwischen SCS. Hier sind offene Formate proprietären Protokollen und Formaten vorzuziehen.

Zur Systemintegration legt die Makroarchitektur Format- und Protokollstandards fest, etwa die Nutzung von HTTP-basierten Schnittstellen und/oder Messaging für asynchrone Interaktionen.

Die Standards definieren ebenfalls, wie Schnittstellen gestaltet werden, aber nicht mit welcher Library, welchem Framework diese Schnittstellen implementiert werden.

Nutzeroberflächen

Die Makroarchitektur definiert verbindliche Prinzipien für die UI-Integration, um eine konsistente Nutzererfahrung über mehrere SCS hinweg zu ermöglichen.

Dazu gehören oft gemeinsame Styleguides, die grundlegende Gestaltungsregeln wie Layout, Typografie, Farben und Interaktionsmuster festlegen. Diese Styleguides dienen der visuellen Kohärenz und stellen sicher, dass unterschiedliche SCS aus Sicht der Nutzer als zusammengehörige Anwendung wahrgenommen werden.

Zur Wiederverwendung von UI-Funktionalität können gemeinsam genutzte Komponenten, etwa in Form von Web Components, bereitgestellt werden. Diese Komponenten sind bewusst auf Präsentations- und Interaktionsaspekte beschränkt und enthalten keine fachliche Logik, um eine enge Kopplung zwischen den SCS zu vermeiden. Die Nutzung solcher Komponenten ist standardisiert, ihre interne Implementierung bleibt jedoch unabhängig von den jeweiligen SCS.

Für die technische Zusammensetzung der Benutzeroberfläche legt die Makroarchitektur Standardwege der UI-Integration fest, beispielsweise über Edge Side Includes (ESI) oder vergleichbare Kompositionsmechanismen. Diese Integrationsformen erfolgen auf klar definierten Schnittstellenebenen und vermeiden geteilten UI-Zustand oder zentrale Frontend-Orchestrierung. Damit wird eine konsistente, integrierte Benutzeroberfläche erreicht, während die Autonomie und unabhängige Weiterentwicklung der einzelnen Self-contained Systems gewahrt bleibt. Verschiedene Ansätze zur UI-Integration behandelt das folgende Kapitel UI-Integration.

Referenzanwendungen und Platform Engineering

Ist die Anwendungslandschaft so weit gewachsen, dass die gemeinsame Wartung und Pflege der technologischen Basis einen Großteil der Makroarchitektur-Runden und/oder der SCS-Teams einnimmt, ist vielleicht der Zeitpunkt gekommen, um die gemeinsame technische Plattform in die Hände eines Plattform-Teams zu geben. Das Plattform-Team kümmert sich weitgehend um die technologische Basis, sodass sich einzelne SCS-Teams ganz auf ihre fachlichen Anforderungen konzentrieren können. Eine umfassende Behandlung des Themas findet sich bei Gregor Hohpe im Buch Platform Strategy[12].

Viele Unternehmen stellen ihren Teams ein „Service Template“ z. B. in Form einer Referenzanwendung zur Verfügung. Diese ist bereits mit sinnvollen Einstellungen und bewährten technischen Lösungen für Querschnittsthemen versehen, sodass ein schneller Einstieg in die Entwicklung gewährleistet werden kann, ohne dass lange über eine „beste“ Lösung diskutiert werden muss.

Mit der Unterscheidung zwischen Mikro- und Makroarchitektur haben wir geklärt, welche Entscheidungen systemübergreifend gelten und welche die Teams eigenständig treffen können. Was damit noch offen bleibt, ist die Frage, wie SCS miteinander kommunizieren — und welche Integrationstechniken dabei zur Verfügung stehen. Das folgende Kapitel bietet hierzu eine Abwägung verschiedener Integrationsmuster.

5 Integrationsmuster

In Kapitel 2 haben wir betrachtet, wie wir große Systeme in fachlich „geschnittene“ SCS zerlegen können. Obwohl diese SCS getrennt entwickelt und deployt werden, soll für Nutzende eine nahtlose User Experience ohne „Systembrüche“ entstehen. Die in Kapitel 3 beschriebene Makroarchitektur hat dafür wichtige Grundlagen gelegt, aber die Wahl einer passenden Integrationstechnik spielt für die User Experience sowie für die Erreichung von Qualitätszielen wie der Resilienz der Systeme eine wichtige Rolle. Dieses Kapitel beschäftigt sich mit verschiedenen Integrationstechniken und deren Vorteilen und Limitationen.

Link-Integration

> Häufig lässt sich eine starke Form der Kopplung von SCS vermeiden, indem lediglich Links von einem SCS auf das andere gesetzt werden.

Bei der Integration durch Links „kennen“ die Systeme voneinander nur die Web-Adresse. Dabei erfolgt die Verlinkung beispielsweise in Form eines „Deeplinks“ auf eine bestimmte Ansicht oder Funktion, die Nutzenden dadurch im fachlichen Kontext angeboten werden kann.

Wohlbekannte URLs werden so zur Schnittstelle zwischen SCS. Dabei können in der Schnittstelle auch Platzhalter vereinbart werden, um Parameter auszutauschen. Das ist praktisch, um gezielt bestimmte Funktionen für Fachobjekte anzusteuern. Beispielsweise könnte beim Absprung aus einer Bestellübersicht in ein PIM-System[1] die SKU[2] eines Produktes mitgegeben werden, um direkt zu den PIM-Daten des Produktes zu springen. Wollen wir den Weg zurück in die ursprüngliche Ansicht ermöglichen, können wir beim Absprung auch zusätzlich eine Rücksprungadresse als Parameter mitgeben.

---

[1] Englisch für *Product Information Management* System. Ein im E-Commerce häufig verwendetes System, das unter anderem Informationen über Produkte wie Produktbeschreibungen und Bilder bereitstellt.

[2] Englisch für *Stock Keeping Unit*. Ein gängiger Schlüssel, um Produkte im E-Commerce eindeutig zu identifizieren.

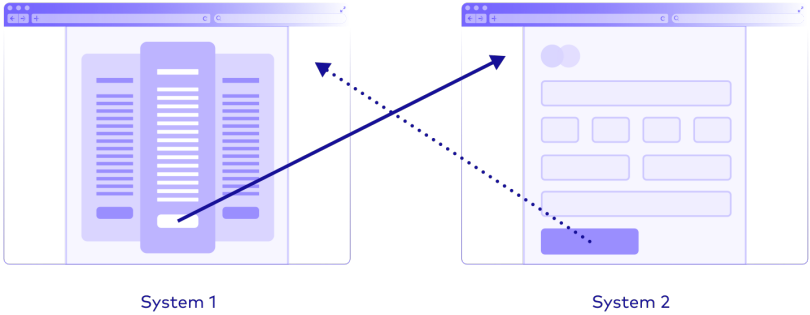

Abbildung 5.1: Zwei SCS werden durch Links integriert

Links sind vor allem dort als Integrationstechnik anzutreffen, wo nur wenig Daten von einem System zum anderen übertragen werden müssen (z. B. beim Übergang einer Artikelsuche in einen Checkout) oder wenn Informationen oder Funktionen erreichbar sein, aber nicht gleichzeitig angezeigt werden müssen. Wo das Verlassen einer Ansicht nicht nutzerfreundlich wäre oder Informationen aus vielen verschiedenen SCS benötigt werden, etwa um eine Entscheidung zu treffen, ist eventuell eine UI-Integration die bessere Wahl.

UI-Integration

Möchten wir beispielsweise alle Produkte einer umfangreichen Bestellung mit vielen Details aus dem PIM-System anzeigen, wäre der Absprung zu jeder der Produktseiten im PIM über einen Link eher unergonomisch. Statt aber die kompletten Produktdaten nun jedes Mal in das System mit der Bestellübersicht zu laden, könnte auch eine Integration der UIs ausreichen.

SCS laden bei der UI-Integration über eine wohlbekannte Web-Adresse „Schnipsel“ fremder Nutzeroberflächen, um Daten im fachlichen Kontext darzustellen. Diese URL wird dadurch wieder zu einer Schnittstelle, für die ebenfalls, wie oben angesprochen, Parameter vergeben werden können. Fremde UI-Komponenten werden in Form von HTML, CSS und ggf. JavaScript ohne die Hülle einer vollständigen HTML-Seite durch das fremde SCS bereitgestellt. Der Konsument dieser

Schnittstelle lädt die Schnipsel und platziert sie an geeigneten Stellen im eigenen HTML-Dokument.

Durch UI-Integration entsteht eine „Composite UI“, bei der verschiedene UI-Komponenten aus verschiedenen SCS stammen. Die Kopplung der SCS untereinander ist dabei sehr gering, denn das konsumierende SCS muss lediglich die URLs der Schnittstellen kennen. Keine weiteren Details über die angezeigten Daten wie Datenformate oder fachliche Bedeutung der Attribute müssen das Quellsystem verlassen. Erweiterungen, wie die Einführung von neuen Attributen, müssen deshalb auch nicht zwischen beiden Systemen koordiniert werden, damit diese im Kontext angezeigt werden. Damit durch diese Form der Integration kein „Flickenteppich“ aus Looks und Feels entsteht, gibt es bei dieser Form der Integration übliche Praktiken und technische Details zu beachten. Im Kapitel 6 wird die UI-Integration deshalb nochmals im Detail behandelt.

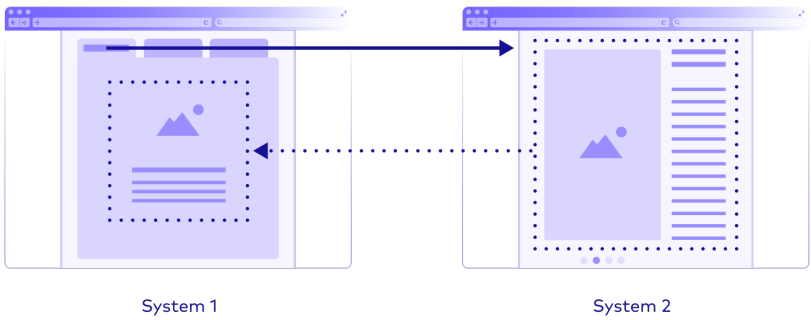

Abbildung 5.2: UI Elemente eines anderen SCS werden integriert

Integration durch asynchrone Aufrufe

In nahezu jedem verteilten System muss ein Teil der Daten auch in einer Form ausgetauscht werden, die zur programmatischen Verarbeitung geeignet ist. Wird der Preis eines Produktes beispielsweise nicht nur angezeigt, sondern zur Berechnung

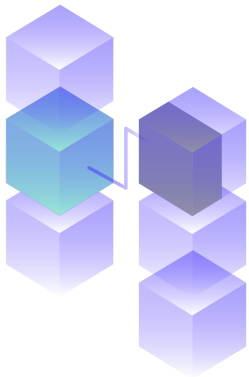

Abbildung 5.3: Ein SCS ruft die Schnittstelle eines anderen SCS asynchron auf

einer Provision benötigt, reicht die reine UI-Integration nicht mehr aus. In diesem Fall haben SCS eine starke Präferenz für asynchronen[3] Datentausch. Das kann zum Beispiel durch eine Message-oriented Middleware (MOM)[4] oder durch die Verwendung von HTTP Feeds[5] erfolgen. Dabei werden präferiert Nachrichten in herstellerunabhängigen Datenformaten wie z. B. JSON oder XML ausgetauscht.

Nachrichten werden dabei nicht nur verwendet, um nebenläufige Prozesse, wie den nebenläufigen Versand von E-Mails anzustoßen (das wäre ein sog. Com-

[3] Die Begriffe "asynchron" und "synchron" wurden leider mehrfach belegt, was zu Missverständnissen führen kann. Wenn in diesem Primer von „synchroner" oder „asynchroner Kommunikation" die Rede ist, ist, soweit nicht anders vermerkt, gemeint, dass nicht auf eine Antwort von dritten Systemen gewartet wird, bevor ein Nutzer mit seiner Arbeit fortfahren kann. Einen „asynchronen" HTTP Client zu verwenden, der den Request Thread nicht blockiert, aber aus Nutzersicht kein Weiterarbeiten ermöglicht bis die Response des HTTP Calls feststeht, ist in diesem Sinne also nicht „asynchron" sondern „synchron".

[4] Message-oriented Middleware ist ein Sammelbegriff für Infrastrukturkomponenten, die sich auf den Austausch von Nachrichten spezialisiert haben. Zu dieser Kategorie zählen wir neben klassischen Message Queues wie ActiveMQ, ZeroMQ oder RabbitMQ auch auf das Datenstreaming optimierte Systeme wie Kafka.

[5] https://www.http-feeds.org/

mand). Sie werden auch verwendet, um vielerorts benötigte Daten unter den SCS zu verteilen. Bei dieser Technik, die auch „Event-carried State Transfer“[6] genannt wird, werden relevante Daten bei einer Änderung durch das Quellsystem[7] in ein *Event* gepackt, das interessierte SCS abonnieren können. Hierdurch können konsumierende Systeme eine lokale Kopie „fremder“ Daten vorhalten und diese laufend mit dem Quellsystem synchronisieren[^fn6].

Durch eigene Datenreplikate werden Zugriffe auf Fremddaten nicht nur sehr viel schneller, sondern auch zuverlässiger. Insbesondere im eigenen Request/Response-Zyklus können so synchrone Zugriffe über das Netzwerk vermieden werden, denn jeder synchrone Zugriff lässt die Latenz der eigenen Response steigen. Synchrone Zugriffe haben zusätzlich auch den negativen Effekt, dass bei eingeschränkter Verfügbarkeit des externen Systems die Funktion des SCS nicht mehr in vollem Umfang gewährleistet werden kann. Bei beidem handelt es sich um negative Effekte, die noch verschlimmert werden, wenn die aufgerufenen Systeme wiederum Systeme aufrufen, die andere Systeme aufrufen. In einem verteilten System bestimmt die Verfügbarkeit des *unzuverlässigsten* gemeinsam genutzten Services die bestmögliche Verfügbarkeit, die das Gesamtsystem garantieren kann. Durch eine Datenreplikation ist auch bei Ausfällen einzelner SCS für eine gewisse Zeit das weitgehend normale Arbeiten mit den anderen SCS möglich.

Die durch die ggf. leicht verzögerte Verarbeitung der Events entstehende „Eventual Consistency“ zwischen SCS ist dabei kein Problem, sofern sie bereits beim fachlichen Schnitt berücksichtigt wurde. Bei dieser fachlichen Analyse werden Funktionen, die garantierte und sofortige Konsistenz benötigen, immer zusammen in ein SCS paketiert werden oder durch Protokolle gesichert, die eine Konsistenz gewährleisten (z. B. durch die Verwendung von Sagas). Was es sonst noch bei der Integration durch Messaging zu beachten gibt, wird in Kapitel 5: Messaging genauer behandelt.

---

[6] https://martinfowler.com/articles/201701-event-driven.html

[7] Es ist eine gute Idee, für Daten nur ein System mit der „Datenhoheit“ über dieses Datum zu definieren. Dieses System hat stets den tatsächlich gültigen Wert in Form einer „Source of Truth“ und verfügt üblicherweise über Regeln, die für Änderungen gelten. Für andere SCS ist dieses führende System das Quellsystem für ihre Daten. Die Replikate kann man dann auch als eine Art „eagerly populated Cache“ betrachten.

Integration durch synchrone Aufrufe

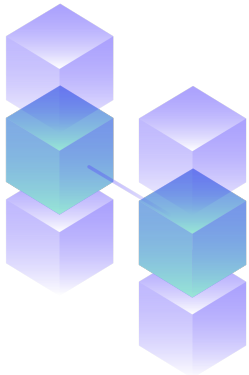

Abbildung 5.4: Ein SCS ruft die Schnittstelle eines anderen SCS synchron auf

Bei Protokollen wie HTTP, GraphQL oder gRPC sind synchrone Aufrufe ein übliches Muster. Sie wirken auf den ersten Blick „einfacher“ als eine Integration durch Messaging. Doch der erste Blick täuscht. Unsere Erfahrung hat gezeigt, dass der ausschließlich synchrone Integrationsstil zu Systemen führt, die mit Herausforderungen bei Latenz und kaskadierenden Fehlern beim Ausfall von Teilsystemen zu kämpfen haben. Folglich wird die vermeintlich einfache Entwicklung durch erstaunliche Komplexität im Betrieb erkauft.

Viele dieser Probleme lassen sich vermeiden, indem das System so gestaltet wird, dass nicht auf Antworten von anderen Systemen gewartet werden muss. Wirklich nötig ist das jedoch auch nur in seltenen Fällen: Eine Integration durch Messaging empfehlen wir dann nicht, wenn die schwachen Konsistenzgarantien inakzeptabel wären. Soll z. B. die Sperrung eines Nutzers in allen SCS sofortige Wirkung zeigen, kann es der bessere Trade-off sein, den Nutzerstatus bei jedem Zugriff synchron aus einem (hochverfügbaren!) Quellsystem abzufragen. Auch wenn ein

Zustand schwer vorberechnet, sondern nur dynamisch aus Queryparametern errechnet werden kann, wie z. B. bei einer Routenberechnung der Fall, ist Messaging eher ungeeignet.

Alle Dimensionen der Kopplung betrachten

Bei der Aufteilung eines Systems im Rahmen der Facharchitektur, wie in Kapitel 2: Domänenarchitektur beschrieben, sollten diese Herausforderungen und Kompromisse offen angesprochen und auf ihre fachliche Akzeptanz untersucht werden, um böse Überraschungen zu verhindern. Dabei stellt sich die Kopplung von verteilten Systemen als eine mehrdimensionale Herausforderung dar, die aus verschiedenen Aspekten betrachtet werden muss:

Was wäre, wenn wir eine Technologie wie die Message Queue wechseln würden? Was wäre, wenn sich Fachlichkeit X mit ihren Schnittstellen ändert? Was wäre, wenn das System Y nicht verfügbar wäre? Was ist, wenn ein System plötzlich Angular als Frontend einsetzen möchte? Diese technischen Fragen gilt es in Einklang zu bringen mit den fachlichen Fragen: Ist es wichtiger, dass Nutzer eine Änderung in einem SCS direkt im anderen nachvollziehen können oder ist es wichtiger, dass auch bei dem Ausfall eines SCS mit dem anderen weitergearbeitet werden kann? Wir empfehlen eine Betrachtung einer jeden Dimension der Kopplung, um fachlich und technisch akzeptierte Trade-offs zu erreichen.

Viele dieser vorgestellten Techniken wie synchrone HTTP-Aufrufe gehören für die meisten Teams zum selbstverständlichen Handwerkszeug. Asynchrones Messaging dagegen wird in vielen Projekten seltener eingesetzt — und damit sind auch die typischen Probleme und Lösungen weniger vertraut. Da wir Messaging als Integrationsform für SCS aber bevorzugen, lohnt sich ein genauerer Blick. Das folgende Kapitel 5 behandelt deshalb Techniken und Fallstricke rund um das Thema Messaging.

6 Integration durch asynchrones Messaging

Messaging klingt einfach: ein System schickt eine Nachricht, ein anderes empfängt sie. In der Praxis stellen sich dabei aber schnell konkrete Fragen — welche Infrastruktur brauche ich überhaupt? Was ist der Unterschied zwischen einem Event und einem Command? Und wie stelle ich sicher, dass keine Nachricht verloren geht? Dieses Kapitel gibt auf diese Fragen praxisnahe Antworten.

Pull vs. Push: Brauche ich eine Messaging-Middleware?

In vielen Unternehmen stellt sich zunächst die Frage, ob und, wenn ja, welche Infrastruktur benötigt wird um die SCS asynchron miteinander zu integrieren. Eventuell ist das aber eine Entscheidung, die man erst einmal aufschieben kann. Gerade wenn Systeme sich noch in der Aufbauphase befinden und die Last gering ist oder wenig Erfahrung mit der Entwicklung und dem Betrieb verteilter Systeme im Team vorhanden ist, kann es eine gute Idee sein, als Zwischenschritt eine Integration über RSS-artige Feeds von Messages zu gestalten. Durch den Einsatz von Feeds kann ohne ein größeres Commitment zu einer Infrastruktur erste Erfahrung mit dieser Integrationsform gesammelt werden.

> Ein HTTP-Feed funktioniert ähnlich wie die Verteilung von Blogartikeln oder Podcasts über Atom- oder RSS-Feeds. Wenn ein System eine neue Message zu veröffentlichen hat, fügt es einen weiteren Eintrag in den eigenen „Feed" ein. Interessierte Systeme können den HTTP-Endpunkt des Feeds aufrufen und dabei die ID der zuletzt konsumierten Message mitgeben, sodass sie nur wirklich „neue" Messages in der Antwort erhalten. Im Prinzip werden Nachrichten dann nach dem „Pull-Prinzip" im System verteilt.

Der Ansatz ist ohne zusätzliche Infrastruktur machbar. Es gibt jedoch einige Einschränkungen: Ohne eine zentrale Infrastruktur wie ein API Gateway oder einen Reverse Proxy müssen irgendwann alle konsumierenden Systeme alle produzierenden Systeme kennen. Bei 2-3 SCS ist das noch überschaubar, wird bei

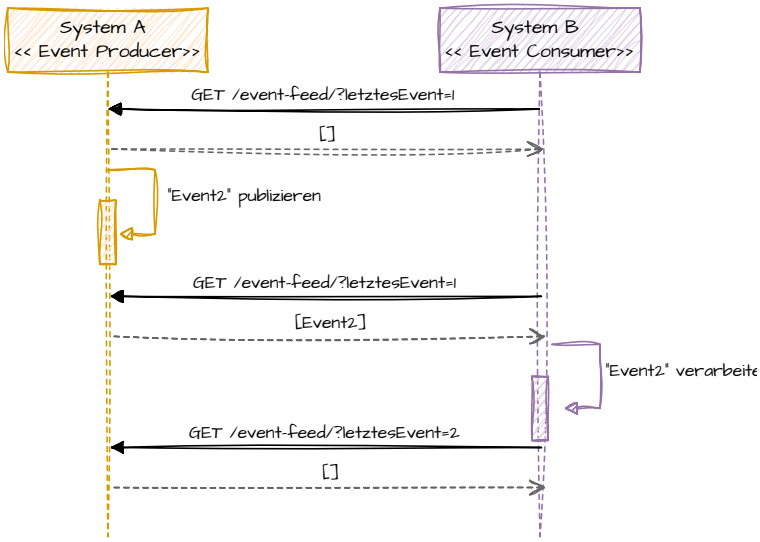

Abbildung 6.1: Ein Konsument ruft Events über eine HTTP-Feed-Schnittstelle des Produzenten ab.

5-10 Systemen aber bereits unübersichtlich. Ein weiterer Nachteil ist, dass bei horizontal skalierenden Systemen ein gleichzeitiges Polling aller Instanzen desselben SCS vermieden werden muss. Andernfalls führt das Polling zu Race Conditions beim Verarbeiten der Nachrichten. Durch das Polling entsteht zudem eine längere Zeit zwischen der Änderung im Quellsystem und der Synchronisierung im nachlaufenden System. Ebenso erzeugt das ständige Nachfragen der Systeme, auch wenn sich nichts geändert hat, unnötige Last. Zuletzt muss auch noch bei immer länger werdenden Feeds irgendwann über das Thema der Pagination oder über die Löschung von historischen Einträgen nachgedacht werden.

Eine „richtige“ Messaging Infrastruktur hingegen funktioniert nach dem Push-Prinzip und ermöglicht ein echtes „fire and forget“ - ein produzierendes SCS wird

weitgehend von den konsumierenden SCS entkoppelt, indem die Middleware für die Auslieferung der Nachrichten zuständig gemacht wird. Dabei gibt es neben den klassischen Message Queues wie RabbitMQ, ActiveMQ und ZeroMQ auch noch spezialisierte Lösungen wie MQTT, die insbesondere bei *occasionally connected clients* wie IoT-Geräten glänzen, oder moderne Tools wie Apache Kafka, die eine hohe Skalierbarkeit auch bei hoher schreibender Last optimiert sind. Auch Hyperscaler liefern mit Diensten wie dem Azure Service Bus oder Amazon SQS fertige Lösungen.

Glücklicherweise ist eine Messaging-Middleware von Natur aus nur sehr lose an Anwendungen gekoppelt. Sie kann deshalb vergleichsweise einfach ausgetauscht werden, falls sich eine Wahl als nicht optimal herausstellt. Auch kommt diese Form der Integration mit ihren eigenen Herausforderungen daher, die im Abschnitt „Reliable Messaging“ weiter erläutert werden.

> Wir empfehlen, mit einer möglichst einfachen Lösung zu beginnen und Erfahrungen zu sammeln und nur bei tatsächlichem Bedarf eine mächtigere Infrastruktur einzuführen. Eine mächtige, hochskalierbare Lösung wie Kafka ist gerade am Anfang oft stark überdimensioniert und lenkt mit ihren Komplexitäten nur vom wesentlichen Problem ab: fachliche Software zu produzieren.

Nachrichtentypen: Events und Commands

Sobald der Transportweg der Nachrichten geklärt ist, stellt sich die Frage, wie so eine Message aussieht. Die Frage „Handelt es sich bei der Nachricht um einen Fakt oder eine Bitte?“ hat sich dabei als sehr hilfreich erwiesen.

Hat ein System eine Bestellung entgegengenommen, ist das ein Fakt, auf den andere Systeme zwar reagieren können, aber den diese nicht ablehnen oder abstreiten können. In diesem Fall sprechen wir von einem „Domain Event“. Möchte ein System hingegen ein anderes um die Erstellung einer Bestellung bitten, hat das konsumierende System die Möglichkeit, die Anfrage zu prüfen und unter

bestimmten fachlichen Umständen (z. B. Betrug vermutet, Kundenkonto ist nicht gedeckt, kein Lagerbestand) abzulehnen.

Domain Events

Mit Domain Events kann ein anderes SCS darüber informiert werden, dass etwas fachlich Relevantes passiert ist.

Ein Domain Event besteht aus einem fachlichen Namen, der beschreibt, was genau passiert ist. Da es sich um bereits Passiertes handelt, sind diese Namen in der Vergangenheitsform (z. B. „Kunde hat gezahlt"). Zusätzlich befinden sich im Rumpf („body") eine Reihe an Nutzdaten wie z. B. eine Versicherungsnummer, eine neue Anschrift, „Gültig ab"-Datum. In Modellen werden Domain Events inzwischen häufig in Anlehnung an Alberto Brandolinis Event Storming mit einem orangefarbenen Klebezettel repräsentiert.

In typisierten Systemen wird der Name des Domain Events gleichzeitig als dessen Typ verwendet. Ein Blick auf den Namen/Typen des Events offenbart deshalb direkt, welche „Felder" im Rumpf der Nachricht zu erwarten sind. Eine wichtige Information für die Deserialisierung. Events werden typischerweise nach dem Publish-Subscribe-Muster verteilt. Interessierte Konsumenten abonnieren hierzu ein Topic unter dem Events eines bestimmten Typs oder zu einer bestimmten Entität veröffentlicht werden. Veröffentlicht der Produzent ein Event, werden alle registrierten Empfänger benachrichtigt, ohne dass der Produzent wissen muss, wer alles auf ein Event „horcht".

Neben dem Anstoßen von Folgeprozessen (z. B. der Kommissionierung nach einem Bestelleingang) werden Domain Events auch benutzt, um Daten zu verteilen, sodass Systeme lokale, nachlaufende Kopien von hoheitlich in einem anderen System verwalteten Daten anlegen können. Diese Kopien dienen wie Caches der verbesserten Performance und werden rein lesend benutzt. Schreibende Anfragen müssen weiterhin an das führende System (das die Datenhoheit besitzt) gehen. Diese schreibenden Anfragen wiederum können dann ein Domain Event auslösen, das alle eventuell nachlaufenden Kopien aktualisiert. Mehr zu dieser Technik im Abschnitt „Event-carried State Transfer".

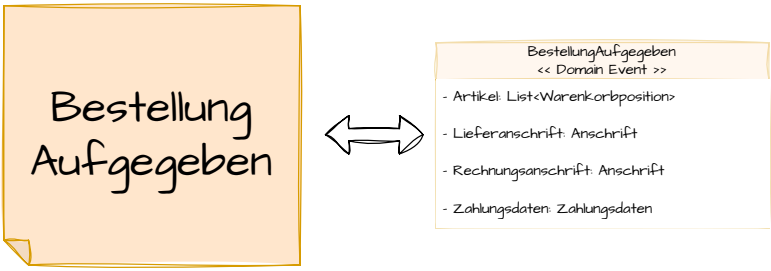

Abbildung 6.2: Ein Domain Event als Klassendiagramm mit dem entsprechenden orangenen Post-it

Commands

Ähnlich wie Domain Events haben Commands einen Namen/Typen und auch einen Rumpf mit Nutzdaten. Ihr Name ist jedoch im Imperativ formuliert („Lege eine Zahlung an“). Häufig werden in horizontal skalierbaren Systemen die Commands an eine Gruppe von Competing Consumers[1] in Form von mehreren Instanzen des gleichen SCS publiziert. Die Middleware kann dazu beispielsweise eine einfache Queue statt eines Publish-Subscribe-Patterns verwenden, sodass garantiert nur eine der Instanzen die Nachricht erhält.

Event-carried State Transfer

Wie bereits erwähnt, können Events nicht nur als Auslöser für Aktionen, sondern auch zur Synchronisation von Daten, die mehr als ein SCS benötigt, verwendet werden. Diese Technik ist auch als Event-carried State Transfer bekannt. Dabei werden bei Erstellung, Änderung oder Löschung durch das Quellsystem Events an alle interessierten Konsumenten versendet. Diese können dadurch ihre dezentralen Kopien aktuell halten. Ein gemeinsamer fachlicher Identifier hilft dabei, die Daten miteinander zu verknüpfen:

Diese reine Kopie der Daten ermöglicht, wie bereits beschrieben, eine größere Unabhängigkeit der SCS zur Laufzeit, sie ermöglicht aber auch noch eine größere

[1] https://www.enterpriseintegrationpatterns.com/patterns/messaging/CompetingConsumers.html

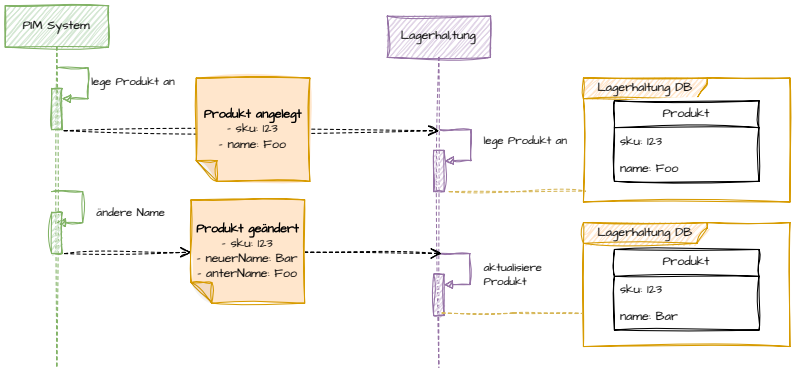

Abbildung 6.3: Ein Produkt wird in einem PIM System erstellt und aktualisiert, während ein Lagerhaltungssystem eine nachlaufende Kopie dieser Daten hält

Unabhängigkeit in der *Modellierung* des eigenen Modells dieser zentralen Entitäten. Beispielsweise kann unser Lagerhaltungssystem im obigen Beispiel eigene Attribute wie den aktuellen Lagerstand und das Regal, in dem das Produkt lagert, ergänzen. Die „fremden“ Attribute sind dabei „read only“ und können nicht direkt geändert werden, während die „eigenen“ Attribute in der Hoheit der eigenen Anwendung sind.

Reliable Messaging

Wie immer in verteilten Systemen kann auch beim Messaging einiges schiefgehen. Glücklicherweise gibt es für alle Fehlermodi gängige Lösungsmuster, die sich bewährt haben.

Eine Voraussetzung für ein gut funktionierendes Gesamtsystem ist, dass die Messaging Middleware nicht zu einem „Single Point of Failure“ wird. Glücklicherweise unterstützen nahezu alle gängigen Messaging-Systeme oder Services den Betrieb in einer redundanten Form (z. B. im Cluster). Damit nicht der Ausfall einer einzelnen Ressource (des Message Brokers) das komplette System zum Stehen bringt, sollten diese Funktionen unbedingt verwendet werden.

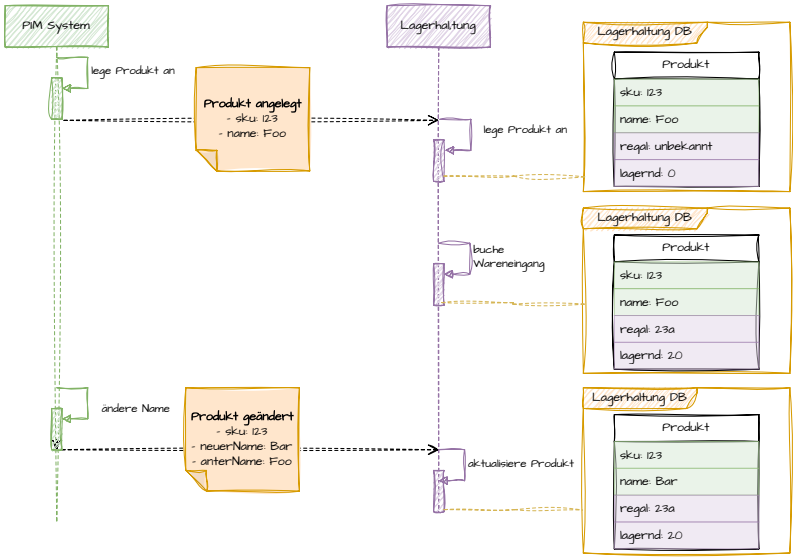

Abbildung 6.4: Unser repliziertes Produkt wird mit eigenen fachlichen Attributen „angereichert“

Message Broker agieren außerdem wie ein Puffer für Arbeit zwischen den einzelnen SCS. Übereifrige Produzenten von Messages können mit ihren Anfragen einen Konsumenten kaum überlasten, da er diese nur in seinem eigenen Tempo abarbeitet und bestätigt. Auch ein Re-Deployment eines Nachrichtenkonsumenten ist in der Regel kein Problem, sofern eine *Durable Subscription* im Broker erstellt wurde. Dabei merkt sich der Broker, dass auch bei einer Downtime die Nachrichten für den Konsumenten aufbewahrt und bei Verfügbarkeit zugestellt werden sollen.

Ein ärgerlicher Fehler entsteht dann, wenn fachliche Datenbanktransaktionen zurückgerollt werden, obwohl schon Events für diese Änderungen versendet wurden. Ein auf die Events horchendes Warenwirtschaftssystem könnte so fälschlicherweise Waren für Bestellungen reservieren, die dann nirgendwo im Quellsystem zu finden sind. Eine mögliche Lösung besteht darin, den Versand der Events mit in die Datenbanktransaktion einzubinden. Einige Broker unterstützen

von Haus aus Transaktionen[2], sodass nur bei erfolgreicher DB-Transaktion auch die Messages „committed“ werden können. Bei allen anderen Brokern kann das Transactional Outbox Pattern[3] einen ähnlichen Effekt erzielen. Viele Messaging Libraries kommen auch bereits mit einer Transactional Outbox, die nur genutzt werden muss.

Eine weitere häufige Quelle „verlorener“ Nachrichten ist eine Exception während der Verarbeitung eines Events. Damit dieses nicht still und leise „verloren geht“, unterstützen viele[4] Broker[5] ein explizites Bestätigen erfolgreich verarbeiteter Nachrichten. Erfolgt in diesem Modus die positive Bestätigung nicht innerhalb einer bestimmten Zeit, wird eine Nachricht in die sog. Dead Letter Queue weitergeleitet. Dort kann sie im Monitoring hervorgehoben, von einem Team inspiziert und nach Fehlerbehebung erneut an das Konsumentensystem gesendet werden.

Im obigen Kapitel wurden typische Techniken und Fehlerquellen, die bei Messaging auftreten können, behandelt. Für eine umfangreiche Betrachtung des Themas empfehlen wir das Buch „Enterprise Integration Patterns“[13].

Ähnlich wie beim Messaging gilt auch für die UI-Integration: Die Grundidee ist schnell erklärt, aber die konkrete Umsetzung birgt Herausforderungen für viele Teams. Das nächste Kapitel behandelt deshalb verschiedene Techniken zur Integration von Nutzeroberflächen.

---

[2] https://activemq.apache.org/components/classic/documentation/how-do-transactions-work

[3] https://web.archive.org/web/20260308173808/https://microservices.io/patterns/data/transactional-outbox.html

[4] https://activemq.apache.org/components/nms/msdoc/1.6.0/vs2005/Output/html/M_Apache_NMS_IMessage_Acknowledge.htm

[5] https://www.rabbitmq.com/docs/confirms#basics

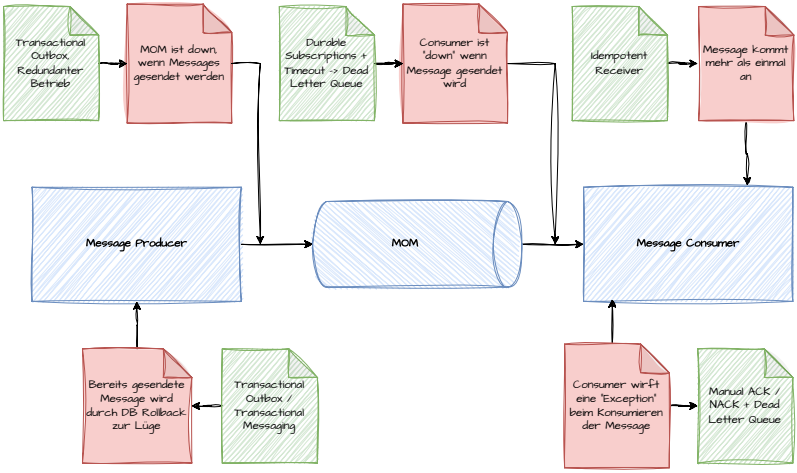

Abbildung 6.5: Übersicht gängiger Risiken (rote Klebezettel) beim Messaging sowie gängiger Lösungen (grüne Klebezettel) für diese Probleme.

7 UI-Integration

Die Modularisierung des Frontend-Codes, sodass dieser mit dem zugehörigen Backend zusammen eine entwickelbare Einheit ergibt, ist zentral für den Erfolg einer Self-contained Systems Architektur. Durch die SCS-Prinzipien ist eine einzelne, monolithische Frontend-Anwendung ausgeschlossen.

Ein einzelnes, unabhängiges Frontend für jedes SCS ist häufig jedoch auch keine Lösung für alle fachlichen Anforderungen. Die Antwort auf die richtige Integration findet sich deshalb häufig in der Mitte, bei weitgehend unabhängigen Frontends, die gegenseitig Teile einer „Composite UI“ austauschen und miteinander integrieren können.

In diesem Kapitel werden gängige Muster zur Integration von Frontends beschrieben. Die hier vorgestellten Muster bauen auf den „Web Frontend Integration Patterns“[14] von Till Schulte-Coerne und Tammo van Lessen auf.

Integration durch isolierte Frames

Eine der einfachsten Möglichkeiten, die UI einer anderen Anwendung einzubinden, ist das im HTML-Standard definierte `<iframe>` Element zu verwenden. Auch wenn iFrames einen vielleicht ungerechtfertigten, schlechten Ruf genießen, wird diese Einbindung für viele gängige und sehr mächtige UI-Integrationen etwa von Google Maps und YouTube verwendet, um ihre Webapplikationen im Kontext von Blogs und anderen Webseiten einzubetten.

Großer Vorteil dieses Ansatzes ist die starke Isolation der eingebetteten Seite, die durch den Browser garantiert wird. Sowohl das DOM als auch CSS und JavaScript werden dabei garantiert isoliert voneinander ausgeführt, wodurch absolut keine Koordination bei Entwicklung und Deployment nötig wird. Dennoch ist es möglich über die `postMessage()`[1] API Nachrichten zwischen der einbettenden und der eingebetteten Webseite auszutauschen, etwa um eingebetteten Inhalt zu steuern. Eine mächtige Kombination, gerade für komplexe, interaktive Komponenten wie die oben genannten Karten und Videoplayer-Elemente.

[1] https://developer.mozilla.org/en-US/docs/Web/API/Window/postMessage

```
<iframe width="560" height="315"
    src="https://www.youtube.com/embed/lJIrF4YjHfQ"
    title="YouTube video player"
    frameborder="0"
    allow="accelerometer; autoplay; clipboard-write;
    encrypted-media;picture-in-picture; web-share"
    referrerpolicy="strict-origin-when-cross-origin"
    allowfullscreen></iframe>
```

Der größte Nachteil an iFrames ist, dass sie sich nicht wie responsive Komponenten verhalten. Ohne eigens zur Anpassung geschriebenen Code nehmen iFrames einen festen Ausschnitt des Bildschirms mit fixer Höhe und Breite ein. Wenn Inhalte nicht in den iFrame passen, scrollt er unabhängig von der einbettenden Webseite. Außerdem brechen die iFrames mit dem Browsing-Modell; kann im iFrame ein Link angeklickt werden, bringt ein solcher Klick die Routing-Logik der einbettenden Applikation eventuell durcheinander. Dazu kommt ein deutlich erhöhter Ressourcenverbrauch, der mit der starken Isolation einhergeht. Bei Anwendungen, die eine Authentisierung von Nutzer:innen fordern, ist es oft nötig, alle Frontends unter der gleichen Domain zu betreiben, damit für den Inhalt des iFrames kein erneutes Login gefordert wird.

Trotz all dieser Einschränkungen gibt es viele Situationen, in denen iFrames ihrem schlechten Ruf als „veraltete Technik“ zum Trotz gute Ergebnisse bei der UI-Integration liefern. Sind iFrames aus Responsivitäts- und/oder Performancegründen aber keine Option, ist es an der Zeit, die weiteren UI-Integrationsmuster wie die im folgenden Abschnitt beschriebene Transklusion von Content in Betracht zu ziehen.

Transkludierter Content

Transklusion bedeutet, dass Inhalte per Referenz in ein anderes Dokument eingebettet werden. Der Inhalt existiert dabei nur einmal an seiner Quelle, wird aber an anderer Stelle angezeigt. Im Kontext von Web-Anwendungen laden wir beim Transkludieren HTML-Fragmente in das DOM einer anderen Webseite. Beispielsweise können die Beschreibungstexte eines Produkts inkl. deren Formatierung nur in einem PIM-System vorhanden sein, aber im Kontext eines Warenkorbes

angezeigt werden. Das System, das die Hoheit über diese Daten besitzt, stellt dabei verschiedene Fragmente unter einer eigenen URI zur Verfügung, die dadurch als Schnittstelle zwischen den beiden Systemen dient und auch Parameter wie eine SKU beinhalten kann.

Die Techniken zur Transklusion von Content sind fast so alt wie das Web selbst. Eine der ersten Techniken hierfür war der sog. Server-side Include. Bei Server-side Include werden anstelle der „fremden“ UI-Komponente Kommentare wie der folgende in das Dokument eingefügt:

```
<!--#include virtual="https://pim-system/products/details/42" -->
```

Ein Web Server wie Nginx wird bei einem solchen `virtual` include (sofern SSI konfiguriert ist) versuchen, die Inhalte von der angegebenen URL zu laden an dieser Stelle einzufügen, bevor er die angefragte Webseite ausliefert. Eine weitere Alternative dieses Ansatzes sind die Edge Side Includes, die häufig durch CDNs angeboten werden.

Beim ESI wird ebenfalls ein Reverse Proxy wie Varnish oder ein CDN verwendet, um einzelne Fragmente vor der Auslieferung der kompletten Webseite in das Dokument einzusetzen. Dabei liefern ESIs insbesondere besseres Fehlerhandling mit. Da beide Ansätze komplett auf der Serverseite operieren, muss der Webserver aber erst alle Include-Links auflösen, bevor er die HTTP Response an einen Client senden kann. Dadurch bestimmt der langsamste Include die Geschwindigkeit der gesamten Seite. Im schlimmsten Fall muss ein Timeout irrelevanter Informationen abgewartet werden, bevor der Nutzer die eigentlichen Inhalte der Seite zu sehen bekommt.

Dagegen ist die dynamische Transklusion durch das Frontend sehr viel vielseitiger. In diesem Fall werden AJAX-Anfragen verwendet, um den Content vom Server nachzuladen. Der Browser kann so auch ersten Content anzeigen, bevor die transkludierten Komponenten geladen werden. Eine beliebte Umsetzungsvariante, die eine deklarative Transklusion ähnlich zu der SSI/ESI Lösung anbietet, ist h-include[2]. Hier sieht die gleiche Transklusion wie im SSI-Beispiel so aus:

[2] https://gustafnk.github.io/h-include/

```
<h-include src="https://pim-system/products/details/42"
           alt="/errors/not-available/"></h-include>
```

So einfach diese Technik ist, leidet sie auch an wesentlichen Einschränkungen: Die Fragmente müssen vom einbindenden System richtig dargestellt werden. Dadurch entstehen nicht nur Abhängigkeiten zu gemeinsam genutzten CSS, sondern auch erhöhte Isolationsaufwände, damit eine Komponente, die ihre Styles „mitbringt“, nicht den Rest der Seite beeinflusst. Sollte ein Fragment einen angemeldeten Nutzer erwarten, müssen zudem gemeinsame Anmeldekontexte bestehen (also z.B. in Form eines geteilten Session-Cookies, das beide Backends verstehen). Das ist oft nur innerhalb der gleichen Domain möglich.

Einbindung von „Self-contained Components“

Eine weitere Variante der UI-Integration ist die Verwendung von in sich geschlossenen UI-Komponenten - hier nennen wir sie Self-contained Components. Dabei wird ein UI-Baustein mitsamt seiner gesamten Logik und Gestaltung in ein wiederverwendbares Modul gekapselt. Diese Kapselung erfolgt beispielsweise durch Web Components, insbesondere durch Custom Elements[3], die typischerweise als ECMAScript-Module ausgeliefert werden. Dabei wird das UI-Element in der Regel durch Einsatz des Shadow DOM[4] vom Rest der Webseite isoliert. Diese Komponenten agieren wie eine eigene, unabhängige Mikro-Anwendung, die an einer beliebigen Stelle in einer Anwendung platziert werden kann.

In dem folgenden Beispiel wird von einem SCS eine Custom Angular Component zur Anzeige eines Produktes mit eigenem, isolierten Style und JavaScript definiert:

[3] https://developer.mozilla.org/en-US/docs/Web/API/Web_components/Using_custom_elements

[4] https://developer.mozilla.org/en-US/docs/Web/API/Web_components/Using_shadow_DOM

```
@Component({
  selector: 'app-product-detail',
  template: `<div>{{ product.name }} [..]</div>`
})

export class ProductDetailComponent {
  @Input() productId!: string;
}
```

Diese Komponente wird dann als Web Component registriert:

```
import { createCustomElement } from '@angular/elements';

const ProductDetailElement = createCustomElement(ProductDetailComponent, {
  injector: injector
});
customElements.define('product-detail', ProductDetailElement);
```

Und kann dann durch andere SCS als Custom Element verwendet werden, um aus einer Produkt-ID eine umfangreichere Detailansicht eines Produktes mit Namen, Bildern und weiteren Details zu rendern.

```
<product-detail productId="42"></product-detail>
```

Auch hier geht die Isolation mit einem erhöhten Ressourcenverbrauch einher, der jedoch in der Regel deutlich niedriger als bei der Verwendung von iFrames liegt. Dennoch sollten bei der Definition der Komponenten besonders leichtgewichtige UI Libraries bevorzugt werden, damit nicht jede „fremde“ Komponente mehrere Megabyte JavaScript nachlädt. Auch wenn eine grundlegende Isolation hier unterstellt werden kann, muss man dennoch aufpassen, dass das fremde JavaScript keine Probleme im eigenen Kontext verursacht. Auch hier müssen alle Anwendungen unter der gleichen Domain laufen, damit Komponenten, die Nutzerdaten benötigen, mit einem geteilten Session Cookie funktionieren. Bestimmte Funktionen wie z. B. ein Routing sind zudem nur schwer mit der Self-contained Component umsetzbar.

Module Federation

Bisher haben wir Ansätze zur UI-Integration betrachtet, die weitgehend auf offenen Standards wie HTML beruhen. Es gibt jedoch beliebte proprietäre Frontend Libraries, die eine fachliche Modularisierung des Frontends ermöglichen. Ein solcher De-facto-Standard ist beim Erscheinen dieses Primers die Library Module Federation[5], eine Weiterentwicklung der Webpack Module Federation. Module Federation ermöglicht es, JavaScript-Module (Komponenten, Bibliotheken, ganze Routen) zur Laufzeit aus einem anderen, separat deploybaren Frontend zu laden – ohne dass beide Apps gemeinsam gebaut werden müssen. Eine Shell-App (Host) zieht sich dabei Code dynamisch von einem Remote Server, der unabhängig versioniert und deployt wird. Gemeinsame Abhängigkeiten, wie beispielsweise ReactJS, werden dabei nur einmal geladen.

Auch dieser Ansatz kommt mit typischen Herausforderungen. Statt nur auf Web-Standards zu setzen, ist hier die gemeinsam genutzte proprietäre Module Federation Library zwingend Teil der Makroarchitektur. Während durch Module Federation die Komponenten zwar zur Build Time entkoppelt sind, bleiben sie durch die gemeinsame Laufzeitumgebung aneinander gekoppelt. Dadurch kommt es zu technischer und organisatorischer Komplexität, die nicht zu unterschätzen ist. Bei der Fehlersuche macht es die gemeinsam genutzte Laufzeitumgebung schwieriger, Zusammenhänge nachzuvollziehen. Schnittstellen der Komponenten müssen zudem Kompatibilitätsgarantien abgeben oder versioniert werden. Bei dem Ressourcenverbrauch verhält sich der Ansatz ähnlich wie bei den Self-contained Components oder besser, sofern das mehrfache Laden von gemeinsam genutzten Libraries verhindert wurde. Auch hier müssen alle Anwendungen unter der gleichen Domain laufen, damit Komponenten, die Nutzerdaten benötigen, mit einem geteilten Session Cookie funktionieren.

Konsistenter Look and Feel

Unabhängig von der gewählten Methode zur UI-Integration soll die UI für den Endnutzer häufig ein stimmiges Gesamtbild ergeben - einen gemeinsamen „Look

[5] https://module-federation.io

and Feel“ haben. Zwei gängige Lösungen hier sind ein gemeinsamer Styleguide, der bestimmte Elemente wie Typografie, Schriftarten und Farben vorgibt oder aber auch eine gemeinsam genutzte Komponentenbibliothek. Ein Styleguide kann dabei unabhängig von der eigentlichen Implementierung der Komponenten ausformuliert werden, indem er nur beschreibt, wie Komponenten auszusehen haben, ohne eine genaue technologische Umsetzung vorzugeben. Das erlaubt neben höherer Langlebigkeit auch die Wiedererkennung über Web- und Plattformgrenzen hinweg.

Oft ist der schnellste Weg, einen gemeinsamen Look and Feel zu erzeugen jedoch, auf eine bestehende Komponentenbibliothek wie Bootstrap[6] oder Material UI[7] zurückzugreifen. Diese Komponentenbibliotheken können recht einfach an den eigenen Styleguide angepasst oder vorkonfiguriert bereitgestellt werden. Dabei decken sie mit ihren Widgets die meisten Nutzungsszenarien bereits ab. In hochspezialisierten Anwendungen oder großen Unternehmen könnte sogar die Schaffung einer eigenen Komponentenbibliothek Sinn machen.

Der gemeinsame Rahmen

Viele Webseiten benötigen gemeinsame Teile wie eine Navigation oder einen Footer-Bereich, die einheitlich über alle SCS hinweg angezeigt werden. Hier stellt sich die Frage: Wo können diese gemeinsam genutzten Komponenten verortet werden? Eine Möglichkeit besteht darin, gemeinsam genutzte Aspekte wie Navigation in das SCS zu legen, das die höchsten Anforderungen an diese hat. Hat beispielsweise ein Autohersteller sehr viele verschiedene Modelle, die in verschiedenen Ausprägungen in der Navigation verfügbar sein sollen, wäre die Anwendung, in der diese Modelle gepflegt werden, ein guter Ort für die Navigation. Die Navigations-Elemente würden dann über einen der beschriebenen Integrationsmechanismen in anderen SCS eingebunden werden.

Im anderen Extrem wäre der gemeinsam genutzte Rahmen eine eigenständige Anwendung, die einem bestimmten Team gehört und unabhängig vom Rest der SCS deploybar ist. Hierdurch entsteht jedoch ein Single Point of Failure, der

[6] https://getbootstrap.com/
[7] https://mui.com/material-ui/

durch die Architektur eigentlich vermieden werden soll. Für Systeme, die auch bei teilweisen Ausfällen noch funktionieren müssen, wäre der bessere Trade-off, die Duplikation und dadurch ggf. entstehende Inkonsistenzen zwischen SCS einfach in Kauf zu nehmen.

Wer bis hierher gelesen hat, kennt nun die Werkzeuge. Bleibt die ehrliche Frage: Lohnt sich das alles wirklich — oder hätte ein gut strukturierter Monolith auch gereicht?

8 Self-contained Systems and Agentic Engineering

Die bisherigen Kapitel haben gezeigt, wie SCS eine unabhängige Entwicklung ermöglichen. Die folgende Fallstudie betrachtet, wie diese Eigenschaften die Arbeit mit Coding-Agents unterstützen können.

TL;DR

Self-contained Systems schaffen gute Voraussetzungen für Agentic Software Engineering, weil fachliche Änderungen innerhalb klarer Systemgrenzen umgesetzt werden können. Ein Coding-Agent kann dadurch UI, Geschäftslogik und Persistenz gemeinsam bearbeiten und deren Zusammenspiel innerhalb des SCS durchgängig testen und überprüfen. Durch die unabhängige Deploybarkeit kann sich der Geschwindigkeitsvorteil der agentischen Entwicklung bis zur produktiven Nutzung fortsetzen.

Die Case Study

Das Unternehmen betreibt ein digitales Handels- und Vermittlungsgeschäft mit stark schwankender Nachfrage. Für die Auftragsabwicklung arbeitet es mit einem dezentralen Netzwerk von Fulfillment-Partnern zusammen. Dabei unterscheiden sich die Fulfillment-Partner unter anderem in ihrem Produktsortiment, ihren Produktions- und Lieferzeiten sowie ihren Auslieferungsmöglichkeiten. Kurze Lieferzeiten sind eine zentrale Anforderung. INNOQ berät das Unternehmen seit vielen Jahren und setzt dabei auf den SCS-Ansatz. Seit 2022 sind mehrere Self-contained Systems entstanden, die geschäftskritische Aufgaben übernehmen. Diese Architektur bestand bereits, bevor INNOQ 2025 begann, zusätzlich systematisch Agentic Software Engineering einzusetzen. Eines dieser Systeme beschäftigt sich mit der Verfügbarkeit von Produkten und deren Lieferbarkeit. Die folgende Fallstudie betrachtet eine fachliche Erweiterung dieses Verfügbarkeits-SCS.

Die Herausforderung: Bestehende Systeme weiterentwickeln

Ein zentraler Bestandteil der Systemlandschaft des Unternehmens ist weiterhin ein monolithisches SAP-System, dessen fachliche Weiterentwicklung nur langsam vorankommt. Einzelne fachliche Bereiche, darunter die Verfügbarkeit, wurden bereits in eigenständige SCS ausgelagert, um Abhängigkeiten zu reduzieren und Änderungen schneller umsetzen zu können. Auch das Fulfillment soll künftig aus dem SAP-System herausgelöst und in einem eigenen Self-contained System umgesetzt werden. Die bestehenden SCS müssen kontinuierlich an neue fachliche Anforderungen angepasst werden. Dabei gilt es, Änderungen zügig umzusetzen, ohne die Stabilität der produktiven Systeme zu gefährden. Eine solche Erweiterung betraf das Verfügbarkeits-SCS: Bei der Verfügbarkeitsprüfung sollten zusätzlich die Liefergebiete der Fulfillment-Partner berücksichtigt werden.

Ein Change: Lieferantenverfügbarkeit geografisch eingrenzen

Für eine bestimmte Produktkategorie gab es Fulfillment-Partner, die bis dahin deutschlandweit Aufträge ausführten: von der Produktion bis zur Auslieferung. Da die Liefergeschwindigkeit stark von der Entfernung zwischen dem Partner und dem Kunden abhängt, entstand die fachliche Anforderung, bei der Verfügbarkeitsprüfung auch die Liefergebiete der Fulfillment-Partner zu berücksichtigen. Das geschäftliche Ziel war, das Lieferangebot besser an die tatsächlichen Möglichkeiten der dezentralen Partner anzupassen und damit die Auslieferung am nächsten Werktag zu unterstützen. Je Fulfillment-Partner sollte eine feste Zuordnung zu einem Liefergebiet erfolgen. Damit sollte das Verfügbarkeitssystem für eine angefragte Lieferadresse nur diejenigen Fulfillment-Partner berücksichtigen, deren Liefergebiet diese Adresse umfasst. Eine regionale Zuordnung allein ist allerdings noch keine Liefergarantie. Für eine solche Zusage müssten beispielsweise auch Produktionszeiten, Annahmeschluss, Kapazitäten und Lieferkalender berücksichtigt werden. Die hier betrachtete Änderung betrifft zunächst nur die geografische Zuständigkeit eines Partners.

Eine fachliche Änderung „End to End“

Das Verfügbarkeits-SCS besitzt neben seiner klaren fachlichen Verantwortung eine eigene Persistenz, die Geschäftslogik, eine API und auch eine eigene UI. Damit hat der Agent ideale Ausgangsbedingungen, um diese Anforderung „Ende zu Ende“ umzusetzen. Die Geschäftslogik wird in Entities, Aggregates und Domain Services umgesetzt. Dabei kann der Agent den Kontext mit der bisherigen Fachlogik zur Verfügbarkeit als Startpunkt verwenden. Die Anzahl der zu analysierenden Codedateien bleibt begrenzt, da fachfremder Code gar nicht Teil der Betrachtung ist. Notwendige Schemaänderungen, um die Liefergebiete effizient speichern und abrufen zu können, kann der Agent gleich mitentwerfen – direkt vorbereitet für das nächste Deployment. Dafür muss er weder die System- noch die Teamgrenze verlassen. Über eine API kann eine Verfügbarkeitsprüfung durchgeführt werden. Auf diesem Weg integriert ein Online-Shop die Verfügbarkeitslogik. Zur Erweiterung gehört auch das Frontend, über das die Liefergebiete definiert und den Fulfillment-Partnern zugeordnet werden. Da es ebenfalls Teil des Verfügbarkeits-SCS ist, kann der Coding-Agent die Oberfläche zusammen mit der Datenhaltung und der Geschäftslogik entwickeln. Die fachliche Änderung bleibt damit über alle beteiligten Schichten hinweg innerhalb einer Systemgrenze. Auch beim Testen hilft es, dass Oberfläche, Geschäftslogik und Datenhaltung zum selben System gehören. Der Agent kann bestehende Tests ausführen und um Testfälle für die geografische Einschränkung ergänzen. Der fachlich begrenzte Umfang hilft dabei, die relevanten Regeln, Testdaten und erwarteten Ergebnisse überschaubar zu halten. Mit Werkzeugen für Browser Use, etwa `agent-browser`, konnte der Agent die Anwendung selbst bedienen: Er steuerte einen Browser, nahm Eingaben über die Oberfläche vor und prüfte deren Auswirkungen über die durch OpenAPI beschriebene API. Bei der Erweiterung um Liefergebiete testete er damit einen durchgängigen Ablauf innerhalb des Verfügbarkeits-SCS: 1. Der Agent startete das SCS lokal mit den benötigten Testdaten. 2. Über `agent-browser` öffnete er die neue Benutzeroberfläche, legte zwei Liefergebiete an und ordnete sie den entsprechenden Fulfillment-Partnern zu. 3. Anschließend stellte er über die API Verfügbarkeitsanfragen für Lieferadressen innerhalb und außerhalb dieser Gebiete. Anhand der OpenAPI-Spezifikation konnte er die dafür vorgesehenen Endpunkte und Datenstrukturen nutzen. Er verglich die Antworten mit den fach-

lich festgelegten Erwartungen und prüfte, ob die geografische Einschränkung wie vorgesehen wirkte.

Damit wurde für die geprüften Szenarien das Zusammenspiel von Oberfläche, Persistenz, Geschäftslogik und API überprüft: Wurden die eingegebenen Liefergebiete gespeichert und bei der Verfügbarkeitsauskunft berücksichtigt? „Ende zu Ende“ bedeutet hier, dass der Weg von der Pflege der Liefergebiete bis zu ihrer fachlichen Wirkung innerhalb des SCS geprüft werden kann. So konnte der Agent direkt am lokal laufenden System prüfen, ob seine Änderungen wie vorgesehen funktionieren. Eine einmalige, toolgestützte Prüfung durch den Agenten ersetzt natürlich keine regelmäßig ausgeführten Regressionstests. Sie ist jedoch ein wertvoller Baustein im Entwicklungsprozess: Der Agent kann eigenständig Feedback aus dem laufenden System einholen, seine Implementierung mit den fachlichen Erwartungen abgleichen und bei Abweichungen gezielt nachbessern. Das Beispiel verdeutlicht damit einen wichtigen Vorteil von Self-contained Systems: Der Agent kann die fachliche Änderung nicht nur über alle beteiligten Schichten hinweg umsetzen, sondern auch deren Zusammenspiel innerhalb derselben Systemgrenze überprüfen.

Unabhängig bis in den Betrieb

Die Eigenständigkeit des Verfügbarkeits-SCS setzt sich beim Deployment fort. Solange die bestehenden Schnittstellenverträge gewahrt bleiben, kann das verantwortliche Team die Erweiterung um Liefergebiete unabhängig von Releases anderer Systeme ausliefern. Die fertiggestellte und geprüfte Änderung muss damit nicht auf ein gemeinsames Release mit anderen Teams warten. Gerade wenn agentische Softwareentwicklung die Implementierung beschleunigt, gewinnt diese Unabhängigkeit an Bedeutung. Wenn die Auslieferung an gemeinsame Releasezyklen oder notwendige Anpassungen anderer Teams gebunden ist, führen schnellere Implementierungen nicht automatisch zu einer früheren produktiven Nutzung. Die Änderung ist zwar fertig, wartet aber noch auf ihre Auslieferung. Solche Wartezeiten begrenzen den Nutzen der durch Coding-Agents gewonnenen Geschwindigkeit und verzögern das Feedback aus dem Betrieb erheblich. Ein unabhängig auslieferbares SCS schafft dagegen die Voraussetzungen für kurze

Zyklen von der Umsetzung bis zur produktiven Nutzung. Das Team kann die Änderung nach erfolgreicher Prüfung und Freigabe bereitstellen, ihre fachliche Wirkung im Betrieb beobachten und daraus weitere Anpassungen ableiten. So kann aus schnellerer Implementierung auch schneller nutzbare Software werden: Der Geschwindigkeitsvorteil agentischer Entwicklung bleibt nicht auf die Arbeit am Code beschränkt.

Nicht mehr Kontext. Mehr Relevanz!

Bei der Erweiterung um Liefergebiete lagen die betroffene Verfügbarkeitslogik, ihre Datenhaltung sowie die zugehörige Oberfläche und API innerhalb desselben SCS. Der Agent konnte die Änderung damit in einem zusammenhängenden fachlichen Kontext bearbeiten. Entscheidend ist nicht allein, wie viel Code in sein Kontextfenster passt, sondern ob die für die konkrete Änderung benötigten Informationen verfügbar sind. Das bedeutet nicht, dass jedes SCS klein ist oder vollständig in ein Kontextfenster passen muss. Ebenso muss ein Agent in einem Monolithen nicht zwangsläufig den gesamten Code lesen. Ein erstrebenswertes Ziel ist aber, dass für eine fachliche Änderung möglichst wenig Wissen über fremde Verantwortungsbereiche benötigt wird.

Schnellere Implementierung allein reicht nicht

Schnellere Implementierung führt nicht automatisch zu früher nutzbarer Software. Wartet eine fertiggestellte Änderung auf Anpassungen anderer Systeme oder ein gemeinsames Release, bleibt ein Teil des möglichen Zeitgewinns ungenutzt. Die Erweiterung um Liefergebiete zeigt, wie ein Coding-Agent eine fachliche Änderung innerhalb eines SCS durchgängig bearbeiten und überprüfen kann. Unter Wahrung der bestehenden Schnittstellen kann das verantwortliche Team sie außerdem unabhängig ausliefern. Diese Eigenständigkeit schafft die Voraussetzung dafür, dass eine schnellere Implementierung auch früher in der Produktion ankommt.

9 Fazit

Stefan Tilkov beendete einen Talk zur Modularisierung von IT-Systemen einmal mit dem Satz „We love monoliths - so let's build a lot of them!“[1]. Erst nachdem der Satz eine Weile in mir wirken konnte, habe ich begriffen, dass er die kurz zuvor erwähnten Self-contained Systems meinte. Und er hatte recht. Eine andere Art, auf diesen Architekturstil zu blicken, ist es, ihn als eine Aufteilung von großen monolithischen Systemen in kleinere, handhabbare Deployment-Monolithen zu betrachten.

Ein Self-contained System lässt sich wie ein Monolith lokal testen, ohne dass dafür viel Infrastruktur oder Mocks anderer Systeme benötigt werden. Man kann viele fachliche Features durch Anpassungen in einem einzelnen Repository umsetzen und kann sich sicher sein, dass Frontend und Backend kompatibel sein werden. Man kann Refactorings von UI bis Datenbank durchführen und sich dabei von Tools unterstützen lassen. Genau wie in einem Monolithen kann man sich darauf verlassen, dass die Daten, die man benötigt, mit niedriger Latenz aus der eigenen Datenbank kommen.

Monolithen waren nie das Problem, es war ihre Tendenz zum unkontrollierten Wachstum. In einem Monolithen werden Dinge, die unabhängig sein können, miteinander stark verzahnt - ohne Rücksicht auf Modularität und Grenzen miteinander verwoben. Die Konsequenzen und Kosten hierfür werden erst viel später sichtbar.

Dieser Primer hat gezeigt, wie ein System gebaut werden kann, dass viele gute Eigenschaften eines Monolithen beibehält und dennoch unabhängig entwickelt werden kann. Dabei gibt es viele Aufgaben wie die Gestaltung der Domänenarchitektur, einer Makro- und Mikroarchitektur sowie der Systemintegration in all ihren Formen zu bewältigen. Es bleibt die Frage: Lohnt sich dieser Aufwand denn?

Wir sagen: Ja, überall dort, wo Systeme so groß geworden sind, dass die Aufwände für Abstimmung und Test enorm geworden sind. Und überall dort, wo verpasste Gelegenheiten, Funktionen schnell an den Markt zu bringen, eine monolithische

[1] https://www.innoq.com/de/talks/2017/11/microservices-patterns-und-anti-patterns/

Architektur an ihre Grenzen bringt. Dort sind die beschriebenen zusätzlichen Aufwände für Domänen-, Makro- und Mikroarchitektur und Integration der Systeme gerechtfertigt.

> Self-contained Systems lösen das Problem, eine Softwarearchitektur und Organisation im Einklang zu skalieren. Dabei vereinen sie die besten Eigenschaften eines Monolithen mit der unabhängigen Entwickelbarkeit eines verteilten Systems.

Egal, ob Sie nun beginnen, ein System „auf der grünen Wiese“ nach den Prinzipien der Self-contained Systems aufzusetzen oder ob Sie versuchen ein bestehendes Legacy System nach diesem Stil umzubauen, wir haben sehr gute Erfahrungen mit dieser Architektur gesammelt und empfehlen sie in solchen Situationen unbedingt in Betracht zu ziehen. Allen, die einen Migrationspfad aus einer Legacy-Architektur suchen, möchten wir den Artikel „Von Legacy-Monolithen zu Self-contained Systems“ [11] ans Herz legen.

Mit den Worten von Stefan:

> Wir lieben Monolithen - also lasst uns viele davon bauen!

Stefan Tilkov

Anhang

Danksagung

Für Stefan, ohne den dieses Buch weder gedacht noch geschrieben worden wäre.

Ich möchte mich an dieser Stelle bei allen bedanken, die Input und Feedback zu diesem Primer gegeben haben: Pascal Erb, Phillip Ghadir, Richard Gross, Stefanie Heinrich, Michael Plöd, Till Schulte-Coerne, Christian Stettler, Ben Wolf und Eberhard Wolff.

Abbildung 1.1: Kombination aus zwei Abbildungen von Martin Weck, Quelle: scs-architecture.org. Dieses Bild steht unter CC BY-SA 4.0. Abbildungen 1.2, 4.1, 4.2, 4.3 und 4.4 von Martin Weck, Quelle: scs-architecture.org, verwendet unter CC BY-SA 4.0. Abbildungen 2.1 und 2.2 mit freundlicher Genehmigung von Michael Plöd.

Über INNOQ

INNOQ

Wir beraten ehrlich, denken innovativ und entwickeln leidenschaftlich gern. Das Ergebnis: Erfolgreiche Softwarelösungen, Infrastrukturen und Geschäftsmodelle.

Als Technologieunternehmen fokussieren wir uns auf Strategie- und Technologieberatung, Softwarearchitektur und -entwicklung, Methoden- und Technologietraining sowie Plattform-Infrastrukturen.

Wir unterstützen mit rund 150 Mitarbeiter:innen an Standorten in Deutschland und der Schweiz Unternehmen und Organisationen bei Konzeption und Um-

setzung komplexer Vorhaben und der Verbesserung bestehender Softwaresysteme.

Wir engagieren uns in Open-Source-Projekten sowie dem iSAQB® e.V., und geben Wissen und Erfahrungen auf Konferenzen und Meetups sowie in zahlreichen Büchern und Fachartikeln weiter.

Besuchen Sie uns: **www.innoq.com**

Unsere Primer

Von Software Reviews über Domain-driven Design bis hin zu RAG: unsere Primer machen den Schnelleinstieg in IT-Themen möglich!

Hier können Sie alle unsere Primer kostenfrei als PDF downloaden: **www.innoq.com/primer**

Literaturhinweise

[1] A. Brandolini, *Introducing EventStorming: An Act of Deliberate Collective Learning*. Leanpub, 2021 [Online]. Verfügbar unter: https://leanpub.com/introducing_eventstorming

[2] A. Brandolini, „Discovering Bounded Contexts with EventStorming“, in *Domain-Driven Design: The First 15 Years*, Leanpub, 2018 [Online]. Verfügbar unter: https://leanpub.com/ddd_first_15_years

[3] D. L. Parnas, „On the Criteria To Be Used in Decomposing Systems into Modules“, *Communications of the ACM*, Bd. 15, Nr. 12, S. 1053–1058, 1972.

[4] M. Plöd, „Modernes Domain-driven Design“, INNOQ, Primer, 2025 [Online]. Verfügbar unter: https://www.innoq.com/de/books/modernes-domain-driven-design/

[5] S. Harrer, L. Visengeriyeva, und J. Christ, „Data Mesh Architektur“, INNOQ, Primer, 2022 [Online]. Verfügbar unter: https://www.innoq.com/de/books/data-mesh-primer/

[6] D. L. Parnas, „The Secret History of Information Hiding“, in *Software Pioneers: Contributions to Software Engineering*, M. Broy und E. Denert, Hrsg. Berlin, Heidelberg: Springer, 2002, S. 398–409.

[7] S. Tilkov, „Software Architecture, Processes, Organization—and Humans“. Talk at YOW! Online, Aug-2020 [Online]. Verfügbar unter: https://www.innoq.com/de/talks/2020/08/software-architecture-processes-organization-and-humans-yow/

[8] J. Humble und D. Farley, *Continuous Delivery: Reliable Software Releases through Build, Test, and Deployment Automation*. Addison-Wesley, 2010.

[9] N. Forsgren, J. Humble, und G. Kim, *Accelerate: The Science of Lean Software and DevOps: Building and Scaling High Performing Technology Organizations*. Portland, OR: IT Revolution Press, 2018.

[10] S. Newman, *Monolith to Microservices: Evolutionary Patterns to Transform Your Monolith*. O’Reilly Media, 2019.

[11] J. Seitz, „Von Legacy-Monolithen zu Self-contained Systems“, *iX Developer*, Nr. 14, S. 44, 2024 [Online]. Verfügbar unter: https://www.innoq.com/de/articles/2025/03/legacy-monolithen-scs/

[12] G. Hohpe, *Platform Strategy: Innovation Through Harmonization*. Leanpub, 2024 [Online]. Verfügbar unter: https://leanpub.com/platformstrategy

[13] G. Hohpe und B. Woolf, *Enterprise Integration Patterns: Designing, Building, and Deploying Messaging Solutions*. Addison-Wesley Professional, 2003 [Online]. Verfügbar unter: https://www.enterpriseintegrationpatterns.com

[14] T. Schulte-Coerne und T. van Lessen, „Web Frontend Integration Patterns“, in *Pattern Languages of Programs, People and Practices: 30th European Conference, EuroPLoP 2025, Irsee, Germany, July 2–6, 2025, Revised Selected Papers, Part I*, 2026, Bd. 16493.

Die Autoren

Johannes Seitz

Johannes Seitz arbeitet als Softwarearchitekt bei INNOQ. Sein Schwerpunkt liegt auf moderner Softwarearchitektur, insbesondere auf Domain-driven Design und DevOps. Als Coach und Trainer hilft er Teams dabei, Software nachhaltig zu bauen oder Legacy Software zu sanieren.

Florian Fritzsche

Florian ist Berater und Software-Entwickler mit mehrjähriger Berufserfahrung. Sein Themenschwerpunkte liegen im Bereich von Java und Domain-driven Design. Darüberhinaus beschäftigt er sich mit agentischer Software-Entwicklung.

Irgendwann beginnt jeder erfolgreiche Monolith zu knirschen: die Codebasis ist zu groß für ein einzelnes Team, das Deployment dauert zu lange, jeder Test berührt zu viele Bereiche, und vor jedem Release koordinieren sich vier Abteilungen. Das System funktioniert noch – aber jede Änderung wird teurer.

Self-contained Systems (SCS) adressieren genau diese Situation. Der Architekturstil zerlegt große Anwendungen in mehrere eigenständige Webanwendungen, von denen jede einen geschlossenen fachlichen Bereich abdeckt – mit eigener Nutzeroberfläche, eigener Geschäftslogik und eigener Datenhaltung. Ein kleines Team kann ein SCS unabhängig entwickeln, deployen und betreiben. Aus einem schwerfälligen Gesamtsystem werden mehrere handhabbare.

Dieser Primer führt durch die Entscheidungen, die dazu gefällt werden müssen: vom fachlichen Schnitt über die Trennung in Mikro- und Makroarchitektur bis zu konkreten Integrationstechniken für Backend und Oberfläche. Mit klaren Empfehlungen, ehrlichen Trade-offs und einem Blick für die Fallen, die unterwegs warten.

Für Architekt:innen, Tech Leads und erfahrene Entwickler:innen, die große Systeme strukturieren müssen und wissen wollen, wie es konkret geht.

innoq.com