Rolf Scheuch
(Rolf.Scheuch@opitz-consulting.com)
ist Diplom-Mathematiker und hat 1990 das IT-Beratungshaus OPITZ CONSULTING mitbegründet. Dort verantwortete er viele Jahre die Bereiche Business Development und Marketing. Seit 2011 ist er Chief Strategy Officer der Unternehmensgruppe und arbeitet zudem als Management-Coach und als Autor diverser Bücher und Publikationen zu Themenbereichen wie BPM, SOA oder Business Information Management.

Risikominimierung bei der Applikationsmodernisierung mittels Microservices

Das Thema der Modernisierung von Legacy-Systemen beschäftigt momentan die deutschen Unternehmen wie kaum ein anderes. Erst kürzlich stellten die Marktanalysten von Lünendonk fest, dass viele Altsysteme bezogen auf die Applikationslandschaft der Unternehmen eine „strukturelle Zukunftsunfähigkeit" bewirken. Denn diese integrierten und umfänglichen Systeme erweisen sich in vielen Fällen als änderungsresistent. Hier kann ein architektonisches Umdenken in Richtung Microservices ein vielversprechender Lösungsansatz sein. Der Artikel beschreibt die Problemstellung und mögliche Vorgehensweisen sowie Nutzen und Vorteile von Microservices-Architekturen bei der Transformation von Legacy-Systemen. Ist es sinnvoll, schon wieder eine monolithische Altapplikation durch einen neuen Monolithen mit moderner Technologie zu ersetzen? Oder liegt vielleicht in der Denkweise des „Big is Beautiful" das Kernproblem der heutigen, meist schwerfälligen Applikationsarchitektur? Danach vertieft der Artikel das Thema Microservices-Architekturen und zeigt, warum diese zu einer Risikominimierung führen und damit eine zukunftssichere Investition darstellen können.

Einführung

Das Thema der Modernisierung von Legacy-Systemen beschäftigt momentan die deutschen Unternehmen und insbesondere deren CIOs. Das Marktforschungsunternehmen Lünendonk spricht in seiner Studie „Software-Modernisierung" (vgl. [Lün15]) von einer „strukturellen Zukunftsunfähigkeit" bei vielen Altsystemen: „Trotz ihres aktuellen Nutzens für die Unternehmen ist Alt-Software strukturell zukunftsunfähig. Das liegt an ihren Charakteristika und den daraus erwachsenen Risiken für das eigentliche Geschäft" (vgl. [Lün15], S. 9).

Die Studie belegt diese Aussage mit Ergebnissen einer Umfrage bei einer ausgewählten Teilmenge deutscher Unternehmen im Jahr 2014 und beschreibt den Handlungsbedarf sehr plakativ. Nachfolgend möchten wir das Paradigma der Microservices-Architektur als einen möglichen Lösungsansatz zur Modernisierung und Ablösung von Altsystemen erläutern.

Modernisierung durch Reengineering

Tatsächlich ist die vollständige Ablösung des Altsystems in der Regel der einzige Weg, um die Zukunftsfähigkeit mit der notwendigen funktionalen Innovation zu erreichen. In der Regel sind drei Optionen denkbar und auch untereinander als Mischform sinnvoll.

Standardsoftware

Stellt das Unternehmen im Rahmen einer Untersuchung fest, dass alle wesentlichen Geschäftsprozesse zur Wertschöpfung durch den Einsatz einer Standardsoftware mit einem geringen Maß an Anpassungen (Customizing) unterstützt werden, so bietet sich die Einführung der Standardlösung an. Ob diese Lösung nun on-premise bzw. gehostet eingeführt wird oder als SaaS-Lösung ist sekundär. Die Vorteile einer Lösung, deren Wartung durch einen spezialisierten externen Anbieter erfolgt, liegen auf der Hand. Gleichwohl erfordert diese Lösung eine Transformation der Fertigungstiefe der eigenen IT. Nur wenige klassische Entwickler sind hier notwendig, stattdessen werden eher Berater gebraucht. Eine Umschulung in diese Richtung ist keine leichte Aufgabe.

Neuentwicklung

Eine kostenintensive, zeitaufwendige und risikoreiche Option ist die Neuentwicklung des Altsystems. Ob dies in Form eines Gewerkes oder mit eher agilen Ansätzen erfolgt, ist sekundär. Entscheidend für die strategische Entscheidung ist es, die Einzigartigkeit des Geschäftsmodells und der Geschäftsprozesse zur Wertschöpfung durch die individuelle Lösung auszuprä-

gen und somit einen – zumindest „gefühlten" – Wettbewerbsvorteil zu sichern und sukzessive durch die neue Plattform auszubauen.

Reengineering
Das Reengineering ist als Gegenentwurf zur vollständigen Neuentwicklung zu sehen – als iterativer Ansatz der Ablösung. Nach einer Analysephase des *Reverse-Engineering*, in der Verständnis und Transparenz der technischen Strukturen des Altsystems hergestellt werden, erfolgt im Sinne eines *Forward-Engineering* die sukzessive Umstellung (oder auch Neuentwicklung) der Funktionalitäten des Altsystems auf die neue Plattform. Die Herausforderung liegt hierbei in der Synchronisierung und Risikominimierung des produktive Betriebs beider Systemwelten: der im Zeitablauf mit jeder Iteration abnehmende Funktionalität des Altsystems bei wachsender Funktionalität der neuen Plattform. In der Fachwelt gibt es hierfür den einprägsamen Begriff „Legacy Application Strangulation" (vgl. [Ham13]), der die sukzessive Ablösung des Altsystems beschreibt.

Wie schon eingangs postuliert, muss man sich hier die Frage stellen, ob es weiterhin Sinn ergibt, eine monolithische Applikation – also ein sogenanntes „integriertes Gesamtsystem" – durch einen neuen Monolithen mit moderner Technologie zu ersetzen oder ob vielleicht in der Denkweise des „Big is Beautiful" das Kernproblem der Applikationsarchitektur liegt.

Dekomposition als Herausforderung

Aus architektonischer Sicht ist die Zerlegung des Altsystems in Komponenten eine wesentliche Herausforderung. Entschließt man sich, das Altsystem abzulösen, so ist die funktionale Dekomposition des Altsystems die erste Aufgabe und die Grundlage für die Bewertung aller weiteren Optionen. Etablierte Methoden für Analyse und Visualisierung gibt es bereits. Hier können auch Elemente der UML 2.0, wie Komponentendiagramme, UML-Objekte „Pakete" und „Sub-Systeme" sowie die Interaktion durch Kollaborationsdiagramme, genutzt werden.

Je enger die Komponenten, Geschäftsregeln und Datenzugriffe des Legacy-Systems miteinander verwoben sind, desto höher ist der Aufwand, das Legacy-System durch ein evolutionäres Vorgehen abzulösen und eine sinnvolle Roadmap hierfür zu definieren. Leider ist dieser Zustand meist die Regel, da das Altsystem eine monolithische Anwendung darstellt. Diese „integrierten Gesamtsysteme" haben alle Interfaces internalisiert und verbergen sie hinter einer Vielzahl von Routinen und Prozeduren.

War diese Architektur zum Entstehungszeitpunkt der Lösung oft eine gute Wahl, insbesondere da die „redundanzfreie Datenhaltung" in RDBMS ein architektonisches Ziel und die Integration durch fehlende Normierung und Standardisierung noch nicht so einfach war, so stellt der „Big is Beautiful"-Ansatz nun das eigentliche Problem dar. Diese architektonische Vision eines „integrierten Systems" behindert Softwarearchitekten heute bei der Absicht, zukunftssichere Lösungen zu erstellen.

Die Vorgehensweise zur Zerlegung eines Altsystems ist wiederum bekannt. Man beginnt mit der Identifikation der Sub-Systeme und beschreibt deren Systemgrenzen, um anschließend die Identifikation der übergreifenden Funktionalität und der Integrationsleistung zwischen den Sub-Systemen vorzunehmen.

Was ist ein Sub-System? Zur Beantwortung dieser zentralen Frage nutzen wir das Paradigma der Microservices-Architektur.

Microservices-Architektur

Microservices oder auch Microservices-Architekturen sind ein neuer Trend in der IT, womit wir wieder ein neues *Buzzword* hätten. Leider gibt es zu unserem jetzigen Kenntnisstand noch keine belastbare Definition. Somit fehlen auch klare Vorgaben für Kriterien (vgl. [Fow14]). Gleichwohl ist die Veränderung des Denkmusters „Monolith" zu einer Ansammlung von eigenständigen Sub-Systemen eine sinnvolle Sichtweise, um flexible und wartbare Lösungen als Alternative für das Altsystem zu schaffen.

Wir verstehen in Anlehnung an M. Fowler (vgl. [Fow14]) unter einem „Monolithen" eine Enterprise-Applikation, die eine eigenständige logisch ausführbare Einheit darstellt und somit als „Ganzes" deployed werden muss. Sämtliche notwendige Geschäftslogik liegt in der Applikation vor, inklusive einer Wiederverwendung von definierten Diensten. Dies sind die typischen Denkmuster der „integrierten Gesamtsysteme" mit den geschilderten Schwachstellen im Zeitablauf.

Wir möchten für eine zukunftssichere Applikationslandschaft jedoch Flexibilität erreichen, eine rasche Reaktionsfähigkeit auf veränderten Anforderungen und eine kostengünstige Wartung. Dies sind Ziele, die ein monolithischer Ansatz nicht erreichen kann.

Unter den ersten Unternehmen, die tatsächlich Microservices einsetzen, haben sich folgende Kriterien etabliert:

- Ein Microservice (Synonym: Sub-System) ist eine **selbstständig auslieferbare (deployable) Einheit, die** in der Regel unabhängig von den Release-Zyklen anderer Applikationen ist.
- Der Microservice ist in Komplexität, Größe und Funktionalität beherrschbar. Hier spricht amazon von der **„Two Pizza Rule"** und meint Projektteams in der Größe von 8-12 Personen (vgl. [Gil14]).
- Die Applikation ist als Produkt und nicht als Projekt zu sehen. Wiederum amazon spricht von der Doktrin **„You build it, you run it"** und hebt damit die Schranken von Entwicklung, Wartung und Betrieb auf (vgl. [Hum10]).
- Das für den Microservice verantwortliche Team besitzt selbst die **notwendige Domänenexpertise**, um im Sinne eines Produktmanagements die Weiterentwicklung über Budget und Portfolio zu steuern.

Diese Kriterien reichen aus unserer Sicht bereits aus, um einige entscheidende Schlussfolgerungen zu ziehen.

Die Reduzierung von Abhängigkeiten der Sub-Systeme untereinander, um eigenständige auslieferbare Einheiten zu ermöglichen, ist notwendig, auch falls dies redundante Funktionalität und redundante Datenhaltung mit sich bringt. In der Folge priorisiert die IT-Strategie eine harmonisierte IT-Landschaft nicht mehr so hoch, sondern legt den Fokus auf eine durchgehende Wertschöpfung durch die IT. Dies legt wiederum nahe, dass unterschiedliche Sub-Systeme unterschiedliche Softwareplattformen besitzen können.

Ein weiterer Aspekt ist, dass das Deployment der Sub-Systeme (Microservices) meist unabhängig voneinander erfolgen kann. Dies impliziert aus der technischen Sicht einen weitestgehenden Verzicht auf die synchrone Anbindung der Sub-Systeme untereinander, um Abhängigkeiten bei der Auslieferung und zur Laufzeit zu vermeiden. Endlich erhalten Unternehmen die Möglichkeit, voneinander unabhängige Release-Zyklen zu im-

plementieren und somit höhere Flexibilität und eine kürzere Time-to-Market von neuen Funktionalitäten zu erreichen und damit einen Mehrwert für den Fachbereich durch IT zu generieren.

Ein Microservice soll nur eine Business Capability unterstützen. Deshalb gibt es nach dem Muster „Single Responsibility Principle" nur einen Grund, in der veränderten Business Capability, einen Microservice zu ändern. Trotz der geschilderten „Two Pizza Rule" geht es bei einem Microservice nicht um Größe, sondern – wie geschildert – um die zusammenhängende und gekapselte Business Capability. Als Folge dieser Sichtweise benötigt ein Microservice, z. B. nach James Lewis, auch keine GUI (vgl. [Lew13]).

Die Sichtweisen **"You build it, you run it"**, in Kombination mit der beherrschbaren Größe des Microservices (**„Two Pizza Rule"**) und der Auffassung, einen Microservice als „Produkt" zu sehen, bedingen die Notwendigkeit eines Produktmanagements an Stelle einer eher singulären Projektsicht. Hintergrund dieser Überlegung ist das Versagen der klassischen Applikation-Lifecycle-Management-Ansätze in Bezug auf einen zielgerichteten Beitrag zur Wertschöpfung der Anwendung – gerade im Zeitablauf.

Die Erfolgsmessung der Wartung ist „projektorientiert". Zeit, Budget und Anforderungen sind fix. Hier steht die Effizienz der Abwicklung und nicht die Effektivität der Lösung für die Fachbereiche im Vordergrund. Die eher Hand-Over- und Quality-Gate-orientierten Entwicklungsprozesse sind zu langsam und es fehlt die gemeinsame Verantwortung für das „Produkt".

Hier können wir von agilen Ansätzen (vgl. [Hue12]) mit der vertikalen Skalierung von Aufgaben der Teams lernen. Zum Beispiel indem wir Domänenexpertise und Betriebs-Know-how in die Teams hineinziehen, sodass Anforderungen, Test, Qualität sowie Deployment in der Gruppe gelöst werden können. Ob dies notwendigerweise zu agilen Vorgehensweisen führen muss, bleibt zu diskutieren, ist aber aus unserer Sicht durchaus wahrscheinlich.

Abhängigkeiten als Herausforderungen

Obwohl wir die Microservices-Architektur in diesem Artikel sehr positiv beschrieben haben, gibt es dennoch eine Vielzahl an Hausaufgaben, die hier zu erledigen sind. Aus unserer Sicht steht die geforderte inhaltliche und technische Trennung der Sub-Systeme im Mittelpunkt einer übergreifenden Governance, die die Aktivitäten der einzelnen Produkt(entwicklungs)teams nicht durch „Bürokratie" lähmen darf.

Dabei geht es im Einzelnen um diese drei wesentlichen Herausforderungen:

- Abhängigkeiten bei der Integrationsleistung minimieren und steuern,
- Abhängigkeiten bei zentralen Objekten minimieren und steuern,
- "You build it, you run it"-Doktrin ermöglichen.

Abhängigkeiten bei der Integrationsleistung

In der Realität wird es zu Abhängigkeiten der Microservices untereinander kommen oder die Nutzung von externen Diensten (Web-Service-Aufrufe von Dritten etc.) wird nötig sein. Die Probleme sind vergleichbar mit der Herausforderung der horizontalen Skalierung bei agilen Ansätzen mit Scrum: Wie sichert man über eine leichtgewichtige Governance Abhängigkeiten der einzelnen Microservices untereinander ab?

Zurzeit erweitern wir die *DoD* (*Definition of Done*), damit die Integration zu den benachbarten Systemen auch in der Verantwortung der Teams liegt. Hierzu werden Teammitglieder aus den „benachbarten" Teams temporär für die Integrationsleistung eingebunden.

Dieser Ansatz orientiert sich an der Implementierung von „Feature Teams" im Scrum-Umfeld. Gemäß der „Two Pizza Rule" schneiden wir die Microservices in der Größe so, dass maximal zwei Teams benötigt werden. Somit bezieht sich der Ansatz der „Feature Teams" in diesem Kontext eher auf systemübergreifende Abhängigkeiten (vgl. [Lar10], S. 150ff).

Gerade durch die Isolierung der Microservices sollen der Aufwand der übergreifenden Koordination, unter Opferung einer harmonisierten Architektur, minimiert, Komplexität reduziert und Reaktionsgeschwindigkeit erhöht werden. Gleichwohl wird die Daten-, Applikations- und Prozessintegration eine bedeutendere Rolle erfahren und zu einer zentralen Herausforderung beim Zusammenspiel der Sub-Systeme werden.

Werden Microservices und SaaS-Lösungen als eigene Microservices bzw. Sub-Systeme kombiniert, liegt eine weitere Herausforderung in der Bereitstellung der notwendigen Integrationsleistung. Die Applikationslandschaft soll sich zukünftig stetig verändern und an den Anforderungen der Fachbereiche orientieren. Somit ist „Ruhe" ein Zeichen von fehlender Innovationskraft und nachlassender Unterstützung der Geschäftsprozesse. Aus diesem Grund sind Cloud-Ansätze, die als eine spezielle „Platform as a Service (PaaS)"-Lösung gegebenenfalls sogar die gesamte Integrations-Plattform zur Verfügung stellen können und elastisch auf Veränderungen reagieren, eine sinnvolle Alternative.

Abhängigkeiten bei zentralen Objekten

Durch die dezentrale Datenhaltung rückt das Datenmanagement zur Sicherung der Nutzung und Nutzbarkeit von zentralen Datenobjekten (Stammdaten, Referenztabellen etc.) in den Vordergrund. Bei dieser Herausforderung kann es sinnvoll sein, auf die Methoden und Vorgehensweisen des Stammdatenmanagements (Master Data Management) zurückzugreifen und für wenige offizielle und grundlegende Geschäftsobjekte eine Governance einführen (vgl. [Sch13], S. 93ff).

Gene Hughson nimmt hierzu Stellung: *„It should be obvious that some governance will be needed to untangle the monoliths and keep them so. The good news is that this particular style (of Microservices) provides the tools to do it incrementally"* (vgl. [Hug14]). Durch die inhaltliche Trennung der Microservices kann man gemeinsam Datensegmente definieren und die Sub-Systeme zuordnen, die für die Datenobjekte und Inhalte verantwortlich und führend sind. Master Data Management nimmt bei Einführung einer Microservices-Architektur eine bedeutende Rolle ein und unterstützt das Konzept des „bounded contexts" (vgl. [Esp14]).

Die zentralen Objekte sind nicht nur Daten, sondern auch übergreifende Dienste. Ein Beispiel ist die übergreifende Harmonisierung von Rechenkernen für Tarife in der Versicherungsbranche. Die Herausforderungen und auch die Gefahren in diesem Zusammenhang sind nicht neu. Hier können wir auf die bewährten Konzepte der SOA-Governance zurückgreifen (vgl. [Ber14], S.22ff).

Bei Microservices wird Codeverdopplung akzeptiert, um eine Kopplung über gemeinsame Komponenten zu verhindern. Allerdings ist es durchaus möglich, dass Microservices auf gemeinsame Libraries

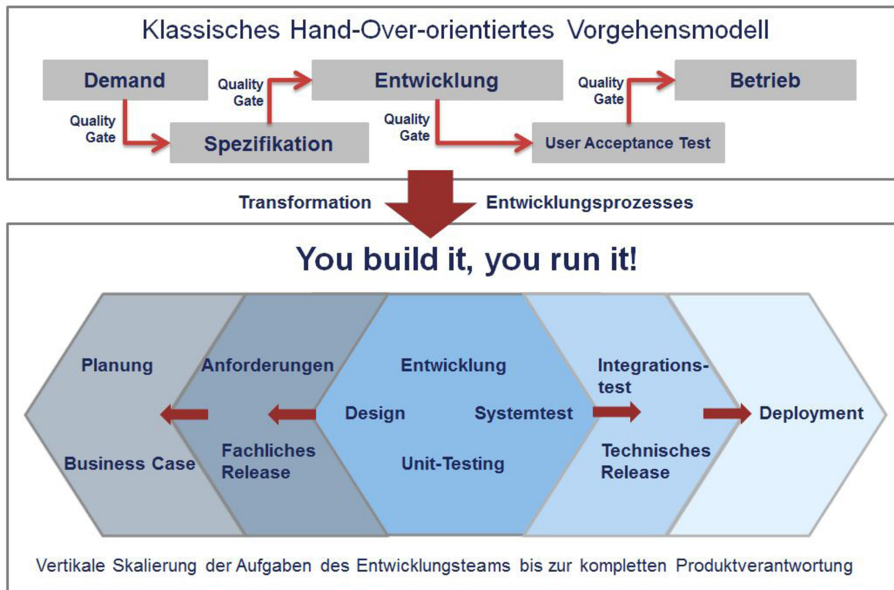

Abb.: Vom Hand-Over-Modell zu einem holistischen Ansatz

zurückgreifen. Dann werden grundlegende Komponenten wie 3rd Party Libraries (z. B. Open Source Libraries) verwendet. Dies ist einer der Gründe, warum Netflix viele seiner Projekte als Open Source auf GitHub bereitstellt. Aber auch die Bereitstellung auf einer firmeninternen Plattform ist möglich.

Auf der Seite der Datenhaltung dürfen sich Microservices-Datenbanksysteme teilen (etwa eine Oracle RDBMS-Installation), müssen aber eine unabhängige Datenhaltung (etwa durch die Implementierung eigener Schemata) beibehalten. Es werden keine Datenbankmittel (DB-Trigger, komplexe Views, DB-Links etc.) genutzt, um diese Trennung aufzuweichen. Microservices sprechen streng über APIs miteinander und teilen keine Daten, Schemata oder interne Objektrepräsentationen.

Inhouse SaaS – "You build it, you run it"

Das Paradigma „You build it, you run it" impliziert beim Microservices-Ansatz ein Umdenken hinsichtlich der Zuständigkeiten und der Aufgaben der IT. In der Regel erfolgte bei den klassischen aktuellen Hand-Over- und Quality-Gate-orientierten Ansätzen (siehe auch V-Modell, PRINCE2 oder ITIL-Ansätze) eine strikte Trennung von Entwicklung, Qualitätssicherung, Testmanagement und Betrieb. Wie am Fließband übergab ein Bereich sein Gewerk an die nächste Stelle und kommunizierte die Fertigstellung mit einer Abnahmeprozedur und oft recht aufwendigen Quality-Gates (vgl. Abbildung).

In der Folge erlahmte die IT-Organisation an den internen Prozeduren und Korrekturschleifen. Der typische Ausweg sind meist wenige, aber dafür entsprechend große Release-Wechsel – oft nur einmal pro Halbjahr. Hierdurch verlängert sich die Time-to-Market einer neuen Idee und dem Fachbereichen geht wertvolle Zeit für Innovationen und eine verbesserte Wertschöpfung in den Geschäftsprozessen verloren.

Projiziert man das klassische Verfahren beim Application Lifycycle-Management (ALM) in die Welt der Microservices, so widerspricht dies sämtlichen Versprechungen einer Microservices-Architektur und steht den in der Einführung erwähnten Treibern einer neuartigen Lösung für das Altsystem entgegen. Die Abbildung beschreibt die notwendige Transformation von einem eher klassischen Wasserfall-Modell mit definierten Quality-Gates und einem dokumentierten Hand-Over zu einem holistischen Ansatz, der typisch für die agilen Entwicklungsansätze ist und über die DoD (Definition of Done) eigenverantwortlich die Qualität sichert (vgl. [Sch13]).

Das neue Paradigma des „DevOps" verspricht hier eine Lösung. Entwicklung („Development"), Test und „Operations" (Betrieb) agieren in enger Abstimmung innerhalb eines Regelkreises (vgl. [Hue12, S. 34ff). Mit diesem Denkmuster sind jedoch auch Schwierigkeiten verbunden: Continuous Build und Deployment der Software müssen möglich sein, alle Testszenarien müssen automatisierbar und die Deployment-Prozesse ebenfalls automatisiert sein (vgl. [For06]).

Und noch nicht schwierig genug, kommt nun erst die eigentliche Herausforderung: Ein begleitendes Veränderungsmanagement muss sich um die Akzeptanz häufiger Release-Wechsel und damit verbundener Veränderungen bei den relevanten Anwendern und Entwicklern bemühen. Hier ist die Domänenkompetenz im Team von großem Vorteil. Da der Fachbereich eng eingebunden ist, sind häufige Release-Wechsel eher willkommen – schließlich wird mit diesen die Qualität gesichert. Ferner liegt nun auch das Produktmarketing als eine Aufgabe des Produktmanagements im Tätigkeitsbereich des Teams. Dieses muss seine Lösung intern „vermarkten" und permanent nach Ansätzen zur Verbesserung der Wertschöpfung suchen (vgl. [Ten13].

Das begleitende Veränderungsmanagement sollte dazu beitragen, dass die Teammitglieder ihre neue ganzheitliche Rolle verstehen, leben und auch *„wollen"*. Der isolierte Spezialist muss stärker als bisher auf die fachlichen Anforderungen eingehen, muss im Team mit engen Abstimmungszyklen arbeiten und seine Ansätze mit dem Betrieb abstimmen. Dies ist eine deutliche Veränderung des Rollenbildes und eher vergleichbar mit den bekannten Herausforderungen bei der Bildung schlagkräftiger agiler Teams.

Vielleicht wird dieser Sachverhalt noch greifbarer, wenn man Microservices mit ihrem Produktmanagement als Inhouse-SaaS-Lösung betrachtet.

Applikationsstrategie

Wie so oft, ist das „Neue" Fluch und Segen zugleich. Mit der Aufspaltung des Altsystems in eine Anzahl von weitestgehend autarken Sub-Systemen geht auch eine Veränderung der Applikationsstrategie einher. War der Monolith früher eine Applikation mit definierten (meist seltenen) Release-Zyklen, so hat nun jedes Sub-System über sein Produktmanagement eine eigene Strategie und verfolgt eine eigenständige Release-Politik. Diese Sichtweisen müssen über eine Governance zur Deckung gebracht werden, ohne die Eigenständigkeit und Einzigartigkeit der Sub-Systeme zu beschneiden und die gewonnenen Vorteile zu verlieren.

Die Analysten von Gartner haben 2012 die *„Pace-Layered Application Strategy"* eingeführt als eine Methodologie für die Kategorisierung, die Auswahl, das Management und die Governance von Applikationen, um Veränderungen im Business, Differenzierung im Markt und Innovationen zu befördern („Methodology for cate-

gorizing, selecting, managing and governing applications to support business change, differentiation and innovation").

Kernidee war, die bessere Unterscheidungsmöglichkeit von unterschiedlichen Applikationsstrategien bei verschiedenen Applikationsklassen zu verbessern. Hier unterscheidet Gartner die folgenden drei Klassen:

- **"Systems of Record"**: Systeme, die grundlegende und standardisierte Geschäftsprozesse, etwa über Standardsoftwaresysteme abbilden. In diese Kategorie fallen Altsysteme oder Legacy-Systeme. Die Evolution dieser Systeme ist langsam und geordnet mit meist wenigen Release-Zyklen. Dies ist, nach Gartner, auch richtig so, da sich diese Geschäftsprozesse nicht oder nur unter dem Druck neuer gesetzlicher Regelungen ändern.
- **„Systems of Differentiation"**: Systeme, die die Einzigartigkeit des Unternehmens in seiner Wettbewerbssituation ausprägen. Diese Systeme unterliegen einem konstanten Wandel, um die Einzigartigkeit des Unternehmens zu unterstützen.
- **„Systems of Innovation"**: Systeme, die neue innovative Geschäftsmodelle oder Experimente unterstützen sollen. Hier sind meist Lean-Startup-Ansätze nötig, um die Ideen der Fachbereiche durch IT-Lösungen auszuprägen. Die neuen technischen Möglichkeiten von Big Data und Digitalisierung sind gute Beispiele für solche neuartigen Systeme.

Durch die Aufspaltung des Monolithen in Sub-Systeme ist es nun möglich, pro Sub-System die passende Applikationsstrategie zu verfolgen. Waren alle Sub-Systeme in einem Monolithen „gefangen", so bestimmte die gemeinsame ausführbare Einheit mit seiner Release-Strategie die Geschwindigkeit aller logischen Sub-Systeme. Der „Langsamste" setzte sich durch. Hier unterstützt die Microservices-Architektur eine differenziertere Applikationsstrategie und trägt somit zur einen besseren, weil schnelleren, Bereitstellung neuer Funktionalität und einer besseren Unterstützung der Wertschöpfung in den unterstützten Geschäftsprozessen bei.

Fazit

Im Vordergrund der Ablösung von eher monolithischen Altsystemen steht die Frage nach der architektonischen Vision der Stakeholder. Falls „Big is Beautiful" die vorherrschende Meinung ist, ergibt der Microservices-Ansatz keinen Sinn. Erkennen die Verantwortlichen, dass die Zerlegung des Monolithen in eigenständige Produkte ein sinnvolles Muster darstellt, um die gesetzten Ziele der neuen Lösung zu erreichen, so kann der neue Ansatz vielversprechend sein.

Neben der Zerlegung des Altsystems in sinnvolle Einheiten, sollte das Augenmerk auf der veränderten Sicht des Softwareerstellungsprozesses liegen. Die Aspekte „You build it, you run it", „Two Pizza Rule" und Produktmanagement stellen dies prägnant dar.

Aus den geschilderten Aspekten der Microservices, wie einzelne Austauschbarkeit und Release-Fähigkeit, ergeben sich Risikominimierung und Zukunftssicherheit schon implizit. Damit gehören diese zu den allgemeinen Vorteilen der Microservices-Architektur. Wir werden diesen Artikel in Kürze mit aktuellen Fallstudien zu unseren Erfahrungen bei Transformationen mittels Microservices-Architektur fortsetzen.

Literatur

[Ber14] Bernhard, Sven; Microservices architecture – thoughts from a SOA perspective, SOA Magazine, 2014

[Esp14] Esposito, Dino; Saltarello, Andrea; Discovering the Domain Architecture, https://www.microsoftpressstore.com/articles/article.aspx?p=2248811, (Download April 2015)

[For06] Schwaber, Carey; The Changing Face of Application Life-Cycle Management. 10/2006, Forrester Whitepaper

[Fow14] Fowler, Martin; Microservices, http://martinfowler.com/articles/microservices.html (Download April 2015)

[Gil14] Gillett, Rachel, Productivity Hack Of The Week: The Two Pizza Approach To Productive Teamwork, http://www.fastcompany.com/3037542/productivity-hack-of-the-week-the-two-pizza-approach-to-productive-teamwork (Download April 2015)

[Ham13] Hammant, Paul; Legacy Application Strangulation: Case Studies, http://paulhammant.com/2013/07/14/legacy-application-strangulation-case-studies/ (Download April 2015)

[Hue12] Hüttermann, Michael; Agile ALM, Manning, 2012

[Hug14] Hughsoh, Gene; Microservices and Data Architecture – Who Owns What Data? https://genehughson.wordpress.com/2014/06/20/microservices-and-data-architecture-who-owns-what-data/ (Download April 2015)

[Hum10] Humble, Jez et al.; Continuous Delivery: Reliable Software Releases through Build, Test, and Deployment Automation, Addison-Wesley, 2010

[Lar10] Larman, Craig; Voss, Bas; Practices for Scaling Lean and Agile Development: Large, Multisite, and Offshore Product Development with Large-Scale Scrum, Addison-Wesley, 2010

[Lew13] Lewis, James; PodCast http://www.se-radio.net/2014/10/episode-213-james-lewis-on-microservices/ (Download im April 2013)

[Lün15] Lünendonk® Whitepaper; Software-Modernisierung, Lünendonk, 2015

[Sch13] Scheuch, Rolf; Warum versagen typische ALM-Ansätze?, IM +io Magazin, 2013

[Ten13] Teng, Ethan et al; Agile Application Lifecycle Management (ALM), Redefining ALM with five key practices, ToughtsWorks Whitepaper