Fakultät für Wirtschaftswissenschaften

Master-Thesis

Migration von Bestandssystemen

Abschlussarbeit zur Erlangung des Grades eines

Master of Science (M.Sc.)

in Wirtschaftsinformatik

der Hochschule Wismar

eingereicht von:	Julia Kordick geboren am 3. Februar 1991 in Bonn Studiengang Wirtschaftsinformatik
Matrikelnummer:	265522
Erstgutachter:	Prof. Dr. Uwe Lämmel
Zweitgutachter:	Prof. Dr. Erhard Alde

Wismar, den 31. März 2020

Inhaltsverzeichnis

Abbildungsverzeichnis

Tabellenverzeichnis

Abkürzungsverzeichnis

DAA Data Access Allocator

DB Datenbank

GI Gesellschaft für Informatik

IS Informationssystem

M Migration

MVPP Model-View-Presenter-Pattern

NE Neuentwicklung

SOA Serviceorientierte Architektur

SPoF Single Point of Failure

1 Einleitung

„It is the goal of every competent software developer to create designs that tolerate change. This seems to be an intractably hard problem to solve. So hard, in fact, that nearly every system ever produced suffers from slow, debilitating rot. The rot is so pervasive that we've come up with a special name for rotten programs. We call them: Legacy Code.
Legacy code. The phrase strikes disgust in the hearts of programmers. It conjures images of slogging through a murky swamp of tangled undergrowth with leaches beneath and stinging flies above. It conjures odors of murk, slime, stag- nancy, and offal. Although our first joy of programming may have been intense, the misery of dealing with legacy code is often sufficient to extinguish that flame." [Feathers, 2004, S. XIII - XIV]

Dieses Zitat von [Feathers, 2004] ist drastisch, beschreibt das Gefühl der Frustration, wenn SoftwareentwicklerInnen, -architektInnen, ProjektleiterInnen und AnwenderInnen in ihrem beruflichen Alltag mit einem historsich gewachsenen Bestandssystem in Berührung kommen, jedoch treffend.

Der Umgang mit gewachsenen Bestandssystemen ist in Konzernen und auch in kleinen und mittelständischen Unternehmen keine Besonderheit. Häufig haben diese monolithische Strukturen und wickeln einen sehr hohen Anteil des operativen Betriebs ab.

1.1 Motivation

Eine Studie aus dem Jahr 2018 zeigt, dass in jedem fünften Unternehmen mehr als 75% und in jedem dritten Unternehmen zwischen 50% und 75% solcher Bestandssysteme im produktiven Einsatz sind. Bei einem Drittel der befragten Unternehmen decken diese Systeme mindestens 50% der geschäftskritischen Prozesse ab. 60% der Firmen geben an, dass der Aufwand für die Wartung im Vergleich zu anderen Systemen höher ist [IDG, 2018, S. 22-23].

Zusätzlich haben Unternehmen das Problem, dass ihnen schon zum jetzigen Zeitpunkt und vor allem in Zukunft nicht mehr genügend IT-Spezialisten zur Wartung und auch für Modernisierungsprojekte zur Verfügung stehen werden. Dies liegt darin begründet, dass die notwendigen Mitarbeiter in den Ruhestand gehen werden [IDG, 2018, S. 38].

Aufgrund dessen wird die Dringlichkeit zur Modernisierung von Bestandssystemen höher.

Ziel einer Modernisierung sind nicht allein die Senkungen von Wartungs- und Betriebskosten. Auch die Sicherheit und Geschäftsprozesse, sowie die Verfügbarkeit und Zuverlässigkeit der betroffenen Systeme sollen sich verbessern [IDG, 2018, S. 26].

Die Aufgabenstellung einer Systemmodernisierung führt zu der Frage nach der notwendigen Vorgehensweise. Wird eine vollständige *Neuentwicklung* angestrebt oder versucht, Teile der unternehmenskritischen Bestandssysteme durch eine *Migration* zu erhalten? Code von Bestandssystemen wird als „Wertstoff" [IDG, 2018, S. 8] angesehen, ebenso wie die abgebildeten Geschäftsprozesse. In durch Aufsichtsbehörden überwachten Geschäftsfeldern sind diese etablierten Prozesse geprüft und abgenommen, was einen weiteren Risikofaktor in der Entscheidungsfindung darstellt. Weitere Faktoren, die die Entscheidung beeinflussen, sind hohe Projektlaufzeiten, Kosten und der kurzfristig zu erwartende "Return of Invest". [Vrgl. [IDG, 2018, S. 7, 8, 28]]

Eine verbreitete Modernisierungsmaßnahme ist beispielsweise die schrittweise Erweiterung und Umstellung der Bestandsanwendungen durch *Microservices*. Diese bieten auf den ersten Blick im Gegensatz zu den monolithischen Bestandssystemen viele Vorteile durch die Strukturierung der Anwendung durch Dienste. Diese sind übersichtlich und leicht (weiter-) entwickelbar, langfristig wartbar und unabhängig voneinander skalierbar. Sie bringen jedoch einen nicht zu unterschätzenden Infrastruktur-Overhead mit sich. (s. 2.1.4)

Auf der Wissensbasis und Dokumentation bereits durchgeführter Projekte, unabhängig von der Größe des Unternehmens und der Bestandssysteme selbst, lassen sich in gewachsenen IT-Architekturen, Anwendungslandschaften und bereits durchgeführten Migrationen deutliche Parallelen im Verlauf von Migrationsprozessen feststellen. Eine Kategorisierung und Abstrahierung von Wissen, Strukturen und Phasen, dieser sich in Zukunft voraussichtlich noch sehr häufig wiederholenden Aufgabenstellung, ist somit der nächste und logische Schritt, zur Unterstützung von EntwicklerInnen, Software-ArchitektInnen und IT-ProjektleiterInnen in der Praxis.

1.2 Ziel

Ziel dieser Arbeit ist eine Evaluierung von Möglichkeiten und Strategien für verschiedene Modernisierungsvorhaben.

Um dieses Ziel zu erreichen werden zunächst die gängigsten Software-Architekturen, sowie die Arten von Bestandssystemen und ihre Eigenschaften aufgelistet und beschrieben. Danach folgt die Beschreibung und Diskussion der Möglichkeiten, um Bestandssysteme abzulösen. Anschließen werden die Strategien, die für Modernisierungsprojekte in Betracht gezogen werden können beschrieben. Nach der Aufarbeitung dieser Möglichkeiten werden sie in einen Kontext gesetzt, um eine qualitative Bewertung der Strategien für verschiedene Konstellationen von Migrationsprojekten auszuarbeiten. Zuletzt wird diese Bewertung dann anhand eines Beispiels validiert.

Das Ergebnis soll eine Unterstützung bei Migrationen bieten, dabei bis zum nötigen Grad abstrahiert und trotzdem noch ausreichen konkret als Leitfaden, Beschreibung und praktisches Werkzeug, das für möglichst viele Problemstellungen Lösungsansätze zur Ablösung von Bestandssystemen bietet, dienen.

1.3 Abgrenzung

Bestandssysteme, ihre Entstehung und Modernisierungsprojekte haben immer auch einen organisatorischen und personellen Aspekt, der in dieser Ausarbeitung nicht betrachtet wird. Auch der Hardware-Aspekt im Sinne des horizontalen und vertikalen Skalierens wird außen vor gelassen. Nur Konzepte untrennbar gekoppelter Hardware/Software-Kombinationen werden erläutert.

1.4 Ansatz & Aufbau der Arbeit

Die vorliegende Arbeit baut im Wesentlichen auf praxisnaher und theoretischer Literatur auf.

Kapitel 1 beschreibt die Motivation für diese Ausarbeitung, sowie das angestrebte Ziel.

Kapitel 2 beinhaltet die theoretischen Grundlagen zu Software-Architekturen und Bestandssystemen.

Kapitel 3 listet Optionen zur Modernisierung von Bestandssystemen auf und diskutiert diese.

Kapitel 4 beschreibt Strategien für die Modernisierungsvorhaben von Bestandssystemen und setzt diese in Zusammenhang mit den Modernisierungsoptionen.

Kapitel 5 bewertet qualitativ auf Basis der vorliegenden Bestandssystemstruktur die Strategien auf dem Weg zum Modernisierungsziel.

Kapitel 6 validiert die vorgenommene Bewertung exemplarisch anhand von anonymisierten Beispielen aus der Praxis.

Kapitel 7 fasst die Ergebnisse in einem Fazit zusammen.

2 Grundlagen

2.1 Software-Architektur

Die *Software-Architektur* eines Systems beschreibt dessen Komponenten, sowie deren Beziehungen zueinander und zu ihrer Umgebung. Sie dient als eine Blaupause des Systems und legt die Aufgaben zu der Entwicklung des Systems fest und bestimmt maßgeblich über Leistungsfähigkeit, Flexibilität, Erweiterbarkeit, sowie die Sicherheit des Systems. [Vrgl. [GI and Hasselbring, 2006], [Carnegie Mellon University, 2017]]

Es wird also deutlich, dass die Modellierung einer durchdachten *Software-Architektur* die Grundlage für ein qualitatives Softwaresystem oder eine Anwendungslandschaft ist.

[Hofmann et al., 2019] nennen drei Grundforderungen an eine Architektur, die gemeinsam als deren Qualitätsmerkmal dienen. Weiterführende Punkte, wie Wartbarkeit und Änderbarkeit ergeben sich laut ihm implizit bei deren Einhaltung:

- die Architektur ist leicht **verständlich**, so dass man ihr folgen kann
- sie ist **benutzbar** und hilft den EntwicklerInnen sich zurechtzufinden und ihre Arbeit effizient zu erledigen
- sie ist **fehlertolerant** und unterstützt EntwicklerInnen bei der Vermeidung von Fehlern in der Implementierung

Komplexität

Das gewünschte Ergebnis der Einhaltung der genannten Grundforderungen lässt sich zusammenfassen in dem Bestreben, Komplexität möglichst nicht entstehen zu lassen. Da Software aufgrund der immensen Anzahl seiner Elemente jedoch eine Grundkomplexität aufweist, gilt es diese durch eine passend gewählte *Software-Architektur* beherrschbar zu halten und besonders den EntwicklerInnen zu helfen, den größtmöglichen Überblick zu behalten. Haben EntwicklerInnnen diesen Überblick, so sinkt die Wahrscheinlichkeit, dass Fehler bei Änderungen in dem Softwaresystem entstehen, so dass auch der Aufbau *technischer Schulden* (s. 2.1.1) und somit auch *Architekturerosion* tendenziell verlangsamt werden kann. [Vrgl. [Lilienthal, 2017a]]

Der Begriff der Komplexität wurde durch die *Komplexitätstheorie*, einem Teilbereich der theoretischen Informatik, im allgemeinen Informatik-Kontext gängig. Diese beschäftigt sich mit der Messung

	Essenziell	Akzidentell
Probleminhärent	■ Komplexität der Fachdomäne	■ Missverständnisse über die Fachdomäne
Lösungsabhängig	■ Komplexität der Technologie und der Architektur	■ Missverständnisse über die Technologie ■ Überflüssige Lösungsanteile

Abbildung 2.1: Komplexität [Lilienthal, 2017a, S.10]

von Algorithmen-Komplexität und der „Grenze zwischen dem mit realistischen Ressourcenbedarf Machbaren und dem nicht effizient Machbaren" [Wegener, 2013, S.2]. Unabhängig von dem Gegenstand der zu messenden Komplexität sind sich [Wegener, 2013, S. 3] und [Lilienthal, 2017a, S. 10] einig: Eines der zentralen Probleme, das oft bei der Schätzung von Komplexität auftritt, ist, dass initial häufig von einer falschen, unterschätzen Schwierigkeit des zu lösenden Problems ausgegangen wird. Dies ist eine Konsequenz aus dem Zusammenhang der Schwierigkeit von Problem und Lösung. Eine iterative, näherungsweise Bestimmung verspricht in der Praxis deswegen realistischere Ergebnisse der Einschätzung.

[Lilienthal, 2017a, S. 9–11] beschreibt die Dimensionen von Komplexität folgendermaßen: Sie unterscheidet zwischen der *probleminhärenten*, *lösungsabhängigen*, *essenziellen* und *akzidentellen* Komplexität. Die Abbildung 2.1 zeigt den Zusammenhang dieser Begriffe. Die *probleminhärente* Komplexität ergibt sich aus dem Anwendungsproblem, für dessen Lösung das Softwaresystem gebaut wurde. Die *lösungsabhängige* Komplexität bezieht sich genau auf diese Lösung. Wie bereits beschrieben, bedingen diese beiden Komplexitäts-Dimensionen sich gegenseitig. Ein zusätzlicher Einflussfaktor auf die *lösungsabhängige* Komplexität sind Erfahrung und Methodenfestigkeit des Entwicklungsteams. Der Richtwert für eine gute Lösung ist deren angemessene Komplexität zur Lösung des zu Grunde liegenden Problems. Unter der *essenziellen* Komplexität versteht man die unvermeidliche Komplexität, die zur Lösung eines Problems notwendig ist. Im Gegensatz dazu steht die *akzidentelle* Komplexität, die die überflüssigen Komplexitätsanteile bezeichnet und somit solche, die zu vermeiden und reduzieren sind. *Akzidentelle* Komplexität kann aus Fehlern in Analyse und Implementierung entstehen. Die angemessene Komplexität besteht somit zu einem möglichst hohen Anteil aus *essenzieller* und einem möglichst geringen Anteil aus *akzidenteller* Komplexität.

2.1.1 Technische Schulden

Technische Schulden sind im Verlauf der (Weiter-) Entwicklung eines Softwaresystems eine gängige Größe zur Messung der (technischen) Qualität eines Softwaresystems und dessen Architektur, bieten aber auch Einblicke in organisatorische Abläufe.

"Der Begriff 'Technische Schulden' ist eine Metapher [..]. [Sie] entstehen, wenn bewusst oder unbewusst falsche oder suboptimale technische Entscheidungen getroffen werden. Diese falschen

oder suboptimalen Entscheidungen führen zu einem späteren Zeitpunkt zu Mehraufwand, der Wartung und Erweiterung verzögert.¨ [Lilienthal, 2017a, S. 4]

Ursachen für *technische Schulden*, die sich meist gegenseitig beeinflussen, sind [Lilienthal, 2017a, S. 7 ff.]

- das **"Programmieren-kann-jeder"**-Phänomen taucht gehäuft in naturwissenschaftlichen Disziplinen auf. Die Programmierenden, KundInnen und das Management sind sich häufig nicht der Unterschiede von Programmieren und *Software-Architektur* und dem grundsätzlichen Aufbau von Systemlandschaften bewusst.

- **steigende Architekturerosion** durch bewusste und unbewusste Abweichung von den Vorgaben der Architektur. Bei der bewussten Abweichung liegt meist das Problem vor, dass die Architektur aufgrund von fehlerhafter Einschätzung zu Problem und Lösung, den Anforderungen nicht gerecht wird. Aufgrund von fehlender Zeit wird das Problem schnell gelöst und keine Änderung und Anpassung der Architektur vorgenommen. Unbewusste Abweichungen finden meist aufgrund von nicht vorhandenem Wissen über die Architektur statt.

- die **Komplexität und Größe von Systemen** und deren unvorhersehbares und unterschätztes Wachstum anhand von Problemen und deren Lösung.

- **Unverständnis von Management und Kunden**, dass eine sich weiter entwickelnde Architektur eine Investition in die Zukunft ist, die Qualität des Softwaresystems erhöht und auf lange Sicht Geld sparen wird.

Im Rahmen der Bewertung des Zustandes von Bestandssystemen stehen primär zwei Arten von technischen Schulden im Fokus: [Vrgl. [Lilienthal, 2017b], [Lilienthal, 2017a, S. 14]]

- **Implementierungsschulden**, sogeannte Code Smells, wie lange Methoden oder auch redundanter Code

- **Design- und Architekturschulden**, die uneinheitliches, komplexes und nicht zusammenpassendes Design von Klassen, Paketen und Modulen beschreiben

Weitere Arten von technischen Schulden sind:

- **Testschulden**, die das Fehlen von (aussagekräftigen) und automatisierten Tests bedeuten

- **Dokumentationsschulden**, also fehlende oder veraltete Dokumentation, an der keine Übersicht über die Architektur abgeleitet und Entwurfsentscheidungen nachvollzogen werden können

Bewertung

Die Bewertung technischer Schulden lässt sich nach [Lilienthal, 2017b] anhand folgenden Kriterien vornehmen:

- Eine gute **Modularität** zeichnet sich durch sinnvoll zusammenhängende Klassen, Komponenten und Module aus, die eine Einheit darstellen. Bausteine einer Ebene sollten sich zu Schichten zusammenfassen lassen. Die fachlichen und technischen Module sollten ein ausgewogenes Größenverhältnis haben. Ebenso ist die (möglichst lose) Kopplung der Module relevant.
- Klar strukturierte **Hierarchien** vereinfachen es die vorliegenden (Teil-)Elemente zu verstehen und nachvollziehen zu können.
- **Musterkonsistenz** bedeutet die schematische Zusammenfassung von typischen Eigenschaften gleichartiger Dinge auf einer höheren Abstraktionsebene, um Fragen und Probleme schneller nachzuvollziehen und zu verarbeiten.

Im Folgenden werden einige architekturelle Grundkonzepte und *Software-Architekturen* beschrieben, die in der Praxis gängig sind.

2.1.2 Serviceorientierte Architektur (SOA)

Anders als der Name *Serviceorientierte Architektur (kurz: SOA)* vermuten lässt, handelt es sich nicht um eine normierte Architektur, wie beispielsweise *Schichtenarchitekturen* oder auch *Microservices*. Eine *SOA* und die ihr zu Grunde liegende "Serviceorientierung ist ein Paradigma, das den Rahmen für [..] Handeln vorgibt" [Josuttis et al., 2009], also ein technologieunabhängiges Architekturmuster, auf denen viele, auch die genannten konkreten Architekturen, basieren. [Vrgl. [GI et al., 2005]]

Der Grundgedanke einer *SOA* sind die Kapselung der technischen Komponenten und die Trennung von fachlichen Zuständigkeiten. Sie beschreibt den Aufbau einer Anwendungslandschaft aus einzelnen fachlichen Anwendungsbausteinen. Diese erfüllen jeweils eine klar definierte fachliche Aufgabe. Eine lose Kopplung wird durch das Zurverfügungstellen von Funktionalitäten über Services sichergestellt. Services können als Element eines oder mehrerer Prozesse verwendet werden und werden über einen einheitlichen Mechanismus angesprochen, der plattforumunabhähgig ist und technische Details kapselt. [Vrgl. [GI et al., 2005], [Josuttis et al., 2009]]

Vorteile & Herausforderungen

Aus den Prinzipen des SOA-Manifest [Josuttis et al., 2009] lassen sich folgende Vorteile von *Serviceorientierten Architekturen* ableiten: [Vrgl. [GI et al., 2005]]

- neue oder angepasste Prozesse lassen sich schnell durch die Kombination bestehender Services realisieren

- durch Kapselung von Implementierungsdetails wird Komplexität reduziert

- die Wiederverwendung fachlicher Komponenten und deren einheitliche Integration reduziert langfristig Wartungs- und Entwicklungskosten

- lose Kopplung und die fachliche Partitionierung begünstigt die Erweiterung und auch schrittweise Ablösung von Bestandssystemen, so sind neue Services nicht von den Technologien des Altsystems abhängig und das Altsystem kann ohne Einschränkungen weiter betrieben werden

- durch Standardisierung der Sevices können Geschäftsprozesse gegebenenfalls einfach ausgelagert werden

Die Herausforderungen im Umgang mit *Serviceorientierten Architekturen* sind: [Vrgl. [GI et al., 2005], [Josuttis et al., 2009]]

- der Einsatz einer *SOA* bietet keine Musterlösung für die Umsetzung der Anforderungen einer Anwendungslandschaft an ihre Infrastruktur

- bei der Abbildung von zukünftig umzusetzenden Anforderungen müssen Organisation und Fachabteilungen vorausschauend planen und auch zum gegenwärtigen Zeitpunkt unbeteiligte Abteilungen in Design und Planung einbeziehen, um Verletzungen des *SOA*-Paradigmas zu vermeiden

- die standardisierte Kommunikation zwischen den Services, die die lose Kopplung bedingen, hat häufig negative Auswirkungen auf die Kommunikationsgeschwindigkeit

- durch die Kapselung von Implementierungsdetails kann es zu Verständnisproblemen der EntwicklerInnen kommen und diese beschneiden gegebenenfalls die Möglichkeit der Anwendung von performanteren Methoden und Techniken

2.1.3 Schichtenarchitekturen

Schichtenarchitekturen (auch: N-Tier/Layer oder Multi-Tier) sind das in der Literatur am häufigsten beschriebene und in der Unternehmens-Praxis eingesetzte Softwareentwicklungs-Konzept. Sie sind der "De-Facto-Standard" [Brautigam, 2019] und es ist schwierig Anwendungen in Unternehmen zu finden, die diesem nicht entsprechen. Wie im Namen bereits enthalten, strukturieren sie sich in Schichten, die Softwaresystembausteine in sich zusammenfassen. [Vrgl. [Brautigam, 2019], [Marston, 2012]]

Vorteile & Herausforderungen

[Marston, 2012] und [Brautigam, 2019] beschreiben die Vorteile der Verwendung einer 3- beziehungsweise N-Tier-Architektur, wie folgt:

- durch die Trennung von Geschäftslogik und Darstellung wird die Anwendung **flexibler** bei Änderungen
- Änderungen an den Komponenten eines Layers haben keinen Effekt auf die anderen Layer, was eine einfache **Wartbarkeit** sicherstellt
- die Separierung der Anwendung in Layer vereinfacht die Entwicklung **wiederverwendbarer** Komponenten
- N-Tier Architekturen ermöglichen die Verteilung der Komponenten der Anwendung, womit sie **skalierbar** wird
- durch redundantes Deployment wird die **Ausfallsicherheit** des Systems erhöht
- auf Basis einer bestehenden N-Tier-Architektur wird die Entwicklung eines **Frameworks** begünstigt, so kann die Grundstruktur der wiederverwendbaren Komponenten ausgeliefert oder generiert werden, was den Entwicklungsaufwand neuer, sowie die Wartung bestehender Komponenten reduzieren kann
- die verbreitete Fähigkeiten-Verteilung von SoftwareentwicklerInnen in Frontend, Backend und Datenbank-EntwicklerInnen begünstigt die unabhängige Arbeit an den Layern dieser ähnlich aufgeteilten Architektur

Auf der Basis von praktischen Erfahrungen mit Schichtenarchitekturen formuliert [Brautigam, 2019] folgende Herausforderungen bei der Arbeit mit Schichtenarchitekturen:

- die **Wiederverwendbarkeit** der Komponenten und auch von Teilen der Geschäftslogik haben sich als wenig praktikabel erwiesen, was auch durch den Trend der Microservices kam, der auf separierte, jedoch ersetzbare, anstatt wiederverwendbare Komponenten setzt
- die Weiter- und Durchleitung von Daten verschlechtert die **Ausführungsgeschwindigkeit** der Anwendungsprozesse
- die Implementierung der gesamten Geschäftslogik in einem separierten Layer verbietet es Logiken, die gegebenenfalls von einer gut konfigurierten Datenbank beziehungsweise deren Zugriffs-Layer schneller abzuarbeiten wären, dort zu implementieren

Strukturen

Die Schichten der Schichtenarchitekturen werden hierarchisch sortiert, so dass tiefere nicht auf höhere Schichten zugreifen dürfen, um so Aufwärts-Beziehungen auszuschließen. Im Falle einer strengen Schichtung dürfen höhere Schichten nur auf die jeweils direkt darunter liegende Schicht zugreifen. So wird es beispielsweise bei der Entwicklung von Betriebssystemen, zur Schaffung von höheren Sprachebenen praktiziert. In herkömmlichen Softwaresystemen liegt in der Regel keine strenge Schichtung vor, so dass höhere Schichten auf mehrere darunter liegende Schichten zugreifen können. Schichtungen gliedern sich üblicherweise in technische und fachliche Dimensionen. [Vrgl. [Lilienthal, 2017a]]

Technische Schichtung

Bei einer technischen Schichtung werden die Bausteine eines Softwaresystems anhand ihrer technischen Verantwortlichkeit sortiert und hierarchisiert. Die auf diese Weise entstehenden Schichten geben den technischen Aufbau des Softwaresystems wieder. Zu deren Nutzung stellen die Schichten Schnittstellen bereit, wobei zyklische Abhängigkeiten darauf hinweisen können, dass die Modularität der Schichten nicht ausreichend gelungen ist. [Vrgl. [Lilienthal, 2017a]] Woran sich häufig orientiert wird, ist die grundsätzliche Aufteilung in Präsentations-, Applikations-, Fachdomänen- und Infrastrukturschicht [Evans and Fowler, 2004]. Das N in N-Tier beziehunsgweise -Layer steht für eine beliebige Nummer (technischer) Schichten, wobei 1- bis 3-Layer gängig sind. Bei Architekturen mit mehr als drei Schichten besteht die Gefahr von Performance Problemen. (s. 2.3, 2.4, 2.5) [Vrgl. [Marston, 2012]]

Fachliche Schichtung

Sobald ein Softwaresystem eine entsprechende Größe erreicht, ist eine technische Schichtung häufig nicht mehr ausreichend, so dass eine zusätzliche Hierarchisierung auf fachlicher Basis erfolgen muss, um die Komplexität der Architektur weiterhin beherrschen zu können. Auch für die Module dieser Schichtung werden Regeln für ihre Beziehungen definiert. Analog zur technischen Schichtung sind Beziehungen nur in eine Richtung erlaubt. Für diese fachliche Schichtung wird das Softwaresystem zusätzlich in vertikale Module eingeteilt (s. 2.2). In jedem fachlichen Modul finden sich die Anteile der technischen Schichten, die zum jeweiligen Geschäftsfeld des Moduls gehören. [Vrgl. [Lilienthal, 2017a]]

Infrastrukturschicht

Für die Nutzung der Infrastrukturschicht reichen die Regeln der nicht-strengen technischen Schichtung häufig nicht aus. Die Folgen sind möglicherweise unplanmäßige und schwer zu erkennende Beziehungen zwischen den Schichten. Um dem entgegen zu wirken bietet es sich an, die Infrastrukturschicht als das unterste Modul der fachlichen Schichtung zu implementieren, so dass

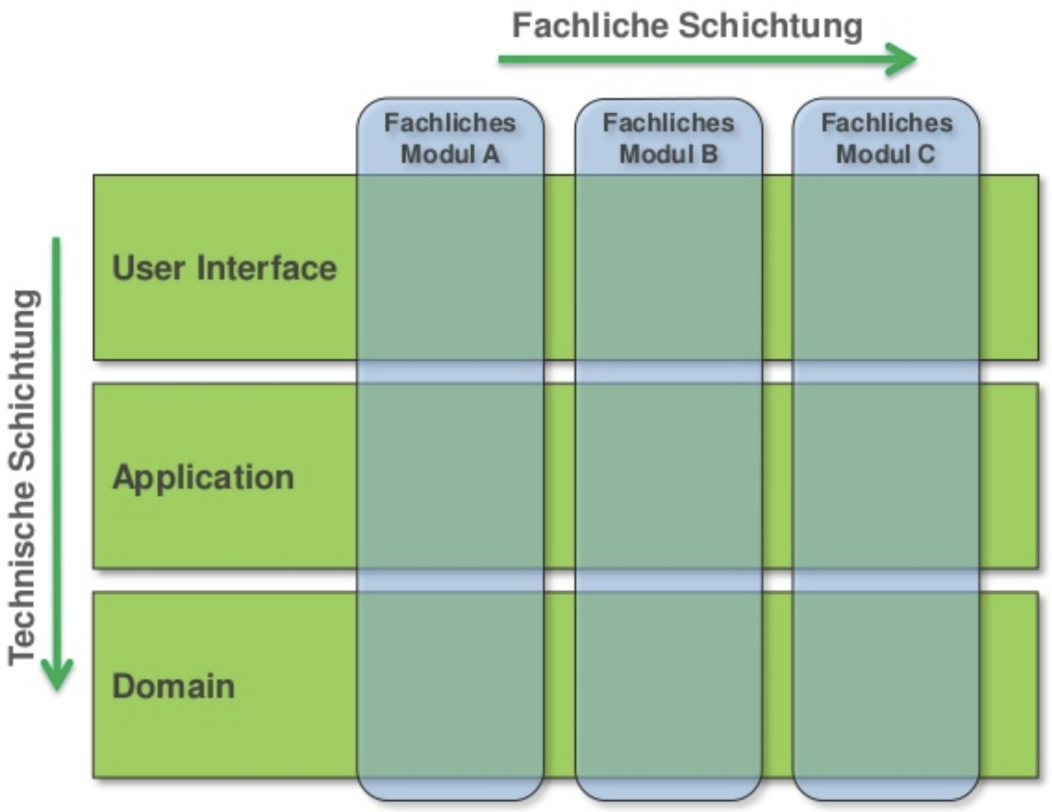

Abbildung 2.2: Schichtenarchitekturen [Lilienthal, 2017a]

dessen Elemente in der richtigen technischen Schicht, und nur dort, verwendet werden können. [Vrgl. [Lilienthal, 2017a]]

2.1.4 Microservices

Microservices werden als das Komplement zu *Monolithen* angesehen.

Ihr Kernkonzept ist Modularisierung basierend auf den drei Aspekten der *UNIX*-Philosophie: [Raymond, 2003]

- "Write programs that do one thing and do it well."
- "Write programs to work together."
- "Write programs to handle text streams, because that is a universal interface."

Der Begriff Microservices lässt sich anhand einiger grundlegender Kriterien präzisieren. Sie sind ein Konzept zur Modularisierung, um so große Softwaresysteme in kleinere, beherrschbare Strukturen überführen zu können und beschreibt darüber hinaus aber auch die Infrastruktur und den Umgang mit ihr als Teil des Services. Diese Module können unabhängig voneinander ausgeliefert werden, sind in sich geschlossen und erfüllen eine Aufgabe für ihre Umgebung. Idealerweise sind sie auch noch statuslos. Folglich können auch Änderungen unabhängig von anderen Services produktiv

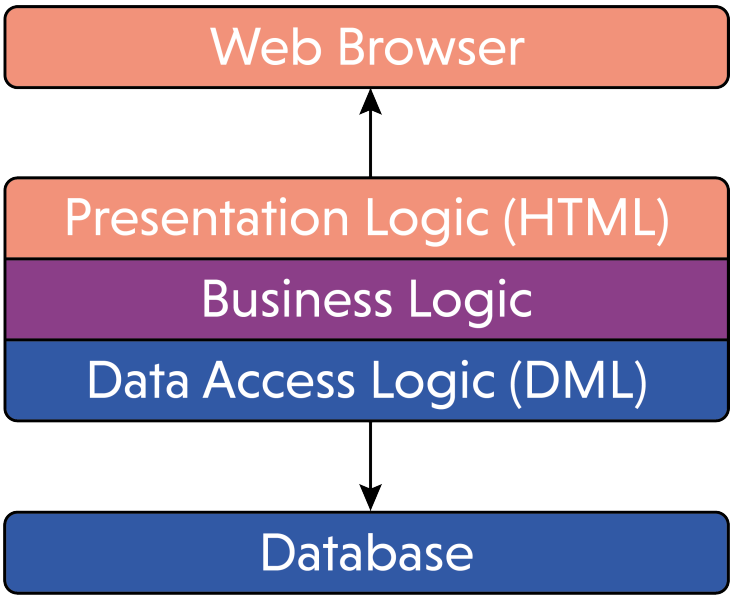

Abbildung 2.3: Bsp. für eine 1-Tier Architektur [Vrgl. [Marston, 2012]]

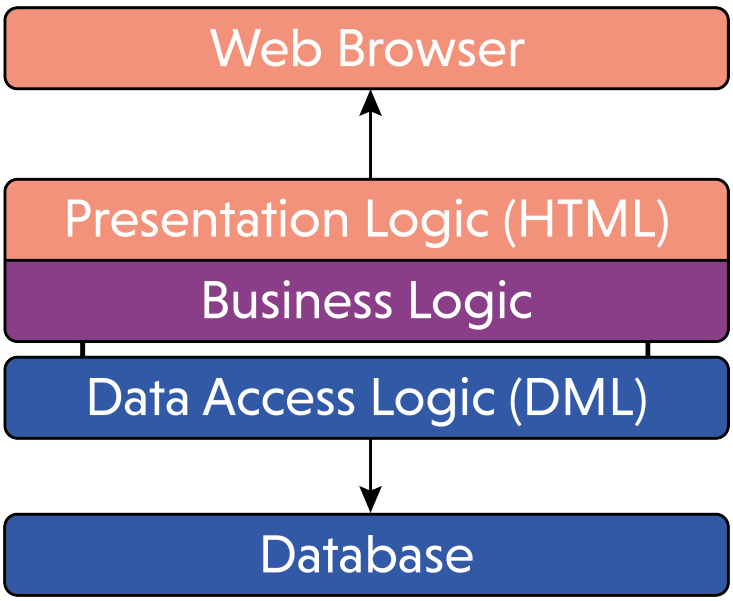

Abbildung 2.4: Bsp. für eine 2-Tier Architektur [Vrgl. [Marston, 2012]]

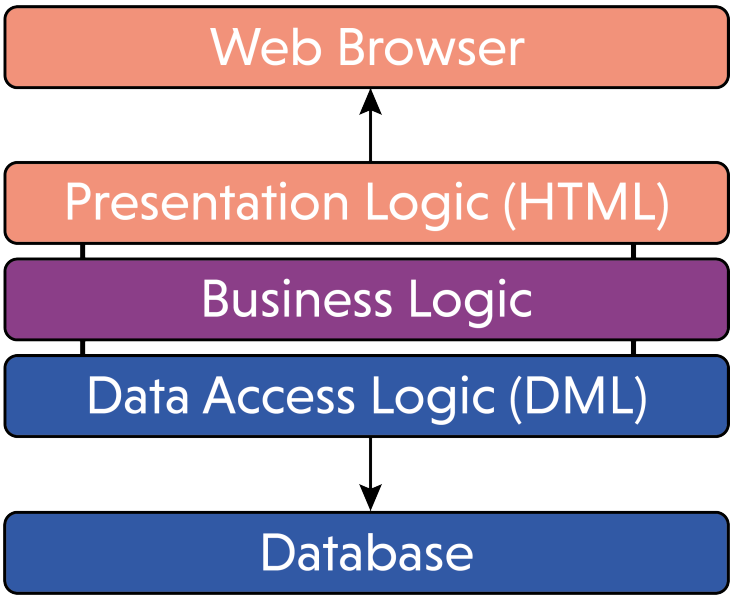

Abbildung 2.5: Bsp. für eine 3-Tier Architektur [Vrgl. [Marston, 2012]]

geschaltet werden. Bei der Implementierung von Microservices gibt es keine technologischen Restriktionen. Jeder Service kann anhand seiner Anforderungen optimiert und auf einer eigenen technischen Basis entwickelt werden. Analog dazu verfügt jedes Modul über eine eigene, private Datenbank oder ein separiertes Teil-Schema einer gemeinsamen Datenbank. Die einzelnen Services kommunizieren miteinander über ein Netzwerk mit Hilfe von Protokollen, die eine lose Kopplung begünstigen und unterstützen, wie beispielsweise *REST*. [Vrgl. [Wolff, 2016], [Roden, 2017a], [Wolff, 2017]]

Vorteile des Einsatzes von Microservices sind: [Vrgl. [Wolff, 2016], [Roden, 2017a], [Hofmann et al., 2019], [Wolff, 2017]]

- **Modularisierung**: Durch Unterstützung des *Teile und Herrsche*-Paradigmas wird der hohe Modularisierung-Grad erreicht. Dieser segmentiert Probleme in beherrschbare Teilprobleme, also Module und erschwert das Entstehen ungewollter Abhängigkeiten zwischen den Modulen und auch den Entwicklungsteams. Optimalerweise bieten die Module auch die Möglichkeit der Wiederverwendung, was Potenzial für Kosteneinsparungen bei der Entwicklung aufzeigt. Auch der Sicherheitsaspekt wird durch die Modularisierung begünstigt. Durch die Möglichkeit die Services durch Firewalls zu trennen, können Kompromittierungen auf einzelne Services eingegrenzt werden.

- **Austauschbarkeit**: Da Microservices sich gegenseitig über explizite Interfaces ansprechen, kann ein Service, solange er das gleiche Interface verwendet, jederzeit durch einen anderen ersetzt werden. Diese Möglichkeit erleichtert auch das Risikomanagement, da das Risiko einer falschen (technologischen) Entscheidung meist auf ein Modul eingegrenzt werden kann und dieses im Zweifelsfall neu implementiert und direkt ausgetauscht werden kann.

- **Nachhaltige Entwicklung**: Modularisierung steuert aktiv gegen *Architekturerosion*, die sich in langen Projekten häufig zunehmend einschleicht. Die Gebundenheit an gegebenenfalls veralte-

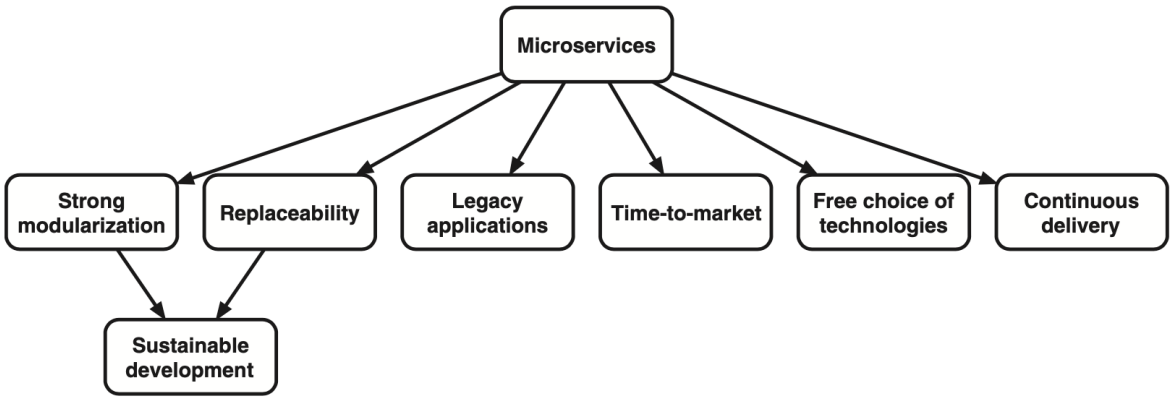

Abbildung 2.6: Vorteile von Microservices [Wolff, 2016, S. 5]

te Technologien und die Risiken im Zusammenhang mit dem Ausbau veralteter Komponenten aus Altsystemen, werden durch den Vorteil des Austauschbarkeit kontrollierbar.

- **Erweiterung von Altsystemen**: Mit Hilfe von Microservices wird die Erweiterung von Altsystemen vereinfacht. Technologisch unabhängig von dem Quellcode des Systems können sie Teile von dessen Aufgaben aufgreifen oder Aufrufe an das Bestandssystem vor dessen Einstieg anpassen. So kann das Bestandssystem nicht nur in Teilen erweitert, sondern auch schrittweise abgelöst werden.

- **Time-to-Market**: Die Möglichkeit, einzelne Services in einer *Microservices*-Architektur unabhängig von anderen Services produktiv setzen zu können und der damit verbundene Vorteil, dass mehrere Teams parallel an verschiedenen Services arbeiten können, ermöglicht eine kurze *Time-to-Market*-Zeitspanne und begünstigt zusätzlich agile Prozesse.

- **Skalierung**: Entsprechend der Nutzung können einzelne Services gezielt skaliert werden, anstatt das gesamte System skalieren zu müssen.

- **Technologie**: Da jeder Service unabhängig voneinander agiert, kann jeder auf der technologischen Basis entwickelt werden, die für den abzubildenden Anwendungsfall optimal ist. Auch der Einsatz und das Ausprobieren neuer Technologien wird durch Microservices begünstigt, da das kalkulierte Risiko sich nur auf den betroffenen Service beschränkt. So entsteht Raum für Innovation und das Abschalten veralteter Technologien wird begünstigt.

- **Continuous Delivery**: Die übersichtliche Größe und Unabhängigkeit von Microservices, verkürzt den *Continuous Delivery*-Prozess und die fehlerfreie, anschließende Auslieferung ist einfacher sicher zu stellen. Im Fehlerfall schlägt nur der betroffene Service fehl, während die anderen weiterhin verfügbar bleiben.

Herausforderungen beim Einsatz von Microservices sind: [Vrgl. [Wolff, 2016], [Wolff, 2017], [Hofmann et al., 2019]]

- **Unklare Abhängigkeiten**: Trotz der Modularisierung ist die Konstellation der Services für die Architektur und deren Funktionen wichtig. Welcher Service welchen Service aufruft ist jedoch schwierig bis nicht nachvollziehbar.
- **Komplexes Refactoring**: Nach der erstmaligen Festlegung, wie die Modularisierung umgesetzt werden soll, gestaltet sich ein Refactoring, besonders, wenn Funktionalitäten zwischen Services verschoben werden sollen, als herausfordernd.
- **Domänen-Architektur**: Die Modularisierung von Microservices bedingt ebenfalls die Team-Aufteilungen und hat somit, besonders im Fehlerfall, direkte Auswirkungen auf die Organisation. Eine Aufteilung der Services auf verschiedene Arbeitsbereiche sichert so die unabhängige Entwicklung der Services. Es werden daher gegebenenfalls auch viele strukturelle Rahmenbedingungen zwingend erforderlich, um langfristig mit *Microservices* erfolgreich zu sein.
- **Infrastruktur verteilter Systeme**: Eine *Microservices*-Architektur besteht aus vielen einzelnen Modulen, die ausgeliefert, kontrolliert und betrieben werden müssen, was die Komplexität durch die Vielzahl von Operationen der Kommunikation zwischen den Services erhöht. Um die Infrastruktur ausreichend überwachen zu können, werden automatisierte Mechanismen benötigt, auch um Netzwerk-Probleme schnell identifizieren zu können. Darüber hinaus sind Operationen, die über ein Netzwerk geleitet werden, meist langsamer als solche, die in einem geschlossenen Prozess abgearbeitet werden.

2.1.5 Monolithen

Als *Monolithen* werden große Softwaresysteme bezeichnet, die nur im Ganzen ausgeliefert werden können. Sie müssen als Ganzes alle Phasen der *Continuous Delivery* und *Integration* Pipeline durchlaufen. Aufgrund ihrer Größe kann dieser Prozess deutlich mehr Zeit in Anspruch nehmen als bei kleineren Systemen, was die Flexibilität und somit auch die verursachten Kosten negativ beeinflusst. Als Monolith werden auch sogenannte *Deployment Monolithen* bezeichnet, die intern eine modulare Struktur aufweisen, bei welcher jedoch alle Module gemeinsam produktiv geschaltet werden müssen. *Monolithen* werden als Komplement von *Microservices* bezeichnet. [Vrgl. [Wolff, 2016]]

[Barashkov, 2019] beschreibt folgende Vorteile im Umgang mit monolithischen Architekturen:

- die CI/CD-Pipeline muss nur einmalig aufgebaut werden, wodurch der DevOps-Verwaltungsaufwand und somit Kosten minimiert werden können
- eine durchdachte, schmal geschnittene Anwendung kann kosteneffizient sein
- die Entwicklung einer monolithischen Anwendung ist einfacher und schneller, da wenige Schnittstellen beachtet werden müssen, alle benötigten Abhängigkeiten in der Anwendung selbst zur Verfügung stehen und das Architektur-Modell übersichtlich ist

Darüber hinaus nennt [Barashkov, 2019] folgende Herausforderungen:

- die Größe des Monolithen beeinflusst maßgeblich die Zeit, die für CI/CD in Anspruch genommen werden muss
- die Anwendung hat einen Single Point of Failure, was im Fehlerfall bedeutet, dass sie als Ganzes fehlschlägt und nicht klar ersichtlich ist, an welcher Stelle der Fehler aufgetreten ist
- durch die Einschränkung, dass Monolithen nur im Ganzen ausgeliefert werden können ist weitestgehend nur eine vertikale Skalierung möglich
- da eine monolithische Anwendung nicht auf mehrere kleine Server verteilt werden kann, steigen die Kosten der benötigten Hardware in Abhängigkeit zur Größe der Anwendung

2.2 Bestandssysteme

In dieser Ausarbeitung ist der Begriff des Bestandssystems das Synonym für veraltete Softwaresysteme, wie sie im Folgenden definiert und beschrieben werden. In der Praxis gibt es jedoch auch solche Systeme, die die Merkmale eines Bestandssystems aufweisen, für ihren Zweck jedoch vollkommen angemessen sind und keine Modernisierungsmaßnahmen benötigen, also den Legacy-Status (bisher) nicht erreicht haben.

Der Legacy-Begriff steht für eine veraltete Methode, Technologie oder ein Softwaresystem „of, relating to, or being a previous or outdated computer system" [Merriam-Webster, 2020].

Bei einem Bestandssystem oder auch Altsystem (engl. Legacy-System) handelt es sich um ein (historisch) gewachsenes Soft- und Hardwaresystem bzw. eine Unternehmensanwendung. In der Regel laufen sie (in Teilen) noch auf Großrechnern [Kaps, 2017a].

Jede Anwendung kann ein Legacy-System werden und das tatsächliche Alter eines Systems steht nicht in direktem Zusammenhang mit dem sogenannten Legacy-Status.

Manche Systeme, beispielsweise im Start-Up-Umfeld, schaffen es bereits frühzeitig in den Legacy-Status. Dies geschieht durch extrem schnelles Prototyping, die Bereitstellung von funktionierender, jedoch nicht nachhaltig entwickelter Software und der sich daraufhin anhäufenden *technischen Schulden* und die beschleunigte *Architekturerosion*, ohne qualitative Tests. Gründe dafür sind die Pflichten, GeldgeberInnen schnell zufrieden stellen und das eigene Geschäft zum Laufen bringen zu müssen [Basedow, 2017]. Die Hoffnung und Annahme, dass Systeme zu einem späteren Zeitpunkt neu implementiert werden, um so technische Schulden zu bereinigen, tritt in der Praxis so gut wie niemals ein. Dies liegt daran, dass die Bereitschaft der GeldgeberInnen lediglich für funktionierenden und nicht für wart- und erweiterbaren Code zu bezahlen, weit verbreitet ist [Roden, 2017b].

Klassische *Bestandssysteme* wachsen über Jahre oder Jahrzehnte gemeinsam mit den sich ändernden Anforderungen an ihre Funktionalität. Sie werden kontinuierlich an aktuelle Gegebenheiten angepasst und entfernen sich durch viele, kleine und schnelle Änderungen weiter von der zugrundeliegenden Spezifikation [Erdle, 2005]. Aus Budget- und Zeitgründen haben neue Features zu beinahe jeder Zeit Vorrang vor Wartungsarbeiten und Refactoring. [Vrgl. [Rinne and Springer, 2017]]

Unabhängig des Alters eines Bestandssystems teilen sie sich daher häufig einen großen Anteil der folgenden Merkmale:

- hohe Komplexität
- starke Kopplung
- zahlreiche Schnittstellen zu anderen Anwendungen
- fehlende oder nicht aktuelle Dokumentation von Anwendungen, Funktionen, Schnittstellen und Abhängigkeiten
- im Quellcode verborgene, ursprüngliche Anforderungen
- überdurchschnittlich stark vom eingesetzten Framework abhängige Implementierungen
- veraltete Betriebs- und Entwicklungsumgebungen und Technologien
- Mangel an Expertenwissen
- eine große Menge inkonsistenter und redundanter Daten

Diese Merkmale stehen in direktem Zusammenhang mit den kritischsten Problemen von und mit *Bestandssystemen*: [Vrgl. [Bisbal et al., 1998], [Basedow, 2017], [Erdle, 2005]]

- die verwendete, veraltete Hardware ist langsam, zieht somit Geschäftsprozesse in die Länge und ist teuer in ihrem Unterhalt
- Softwarewartung und Fehlerbehebung ist kostenintensiv und mit steigender Systemkomplexität steigt dieser Kostenfaktor unkontrolliert
- mit der fehlenden Dokumentation geht meist auch das mangelnde Verständnis über das System einher
- die Erweiterung und Anpassung von Bestandssystemen wird ebenfalls zunehmend komplexer
- EntwicklerInnen beginnen Wartungsarbeiten aktiv zu vermeiden, aus Angst, die unübersichtlichen Funktionen unbenutzbar zu machen
- aufgrund dessen verringert sich die Anzahl der Personen, die Wissen über Wartung und Anpassungen haben
- die Einarbeitung neuer MitarbeiterInnen wird organisatorisch schwieriger und durch den zusätzlichen Zeitaufwand kostenintensiver

Die Merkmale sind ebenfalls Gründe, dass die sich aus den Problemen ableitbare Notwendigkeit zur Erneuerung, so weit wie möglich hinausgezögert wird. Ein weiterer Grund ist der hohe Zeitaufwand, aus dem nicht unmittelbar ein Mehrwert garantiert werden kann, der aber trotzdem hohe Kosten und teilweise sogar Umsatzeinbußen verursachen kann. [Vrgl. [Kaps, 2017a], [Basedow, 2017]]

Grundsätzlich unterscheidet man Bestandssysteme anhand ihres Aufbaus. Es wird zwischen strukturierten, semi-strukturierten und unstrukturierten Systemen (s. *Monolithen*) unterschieden. Anhand

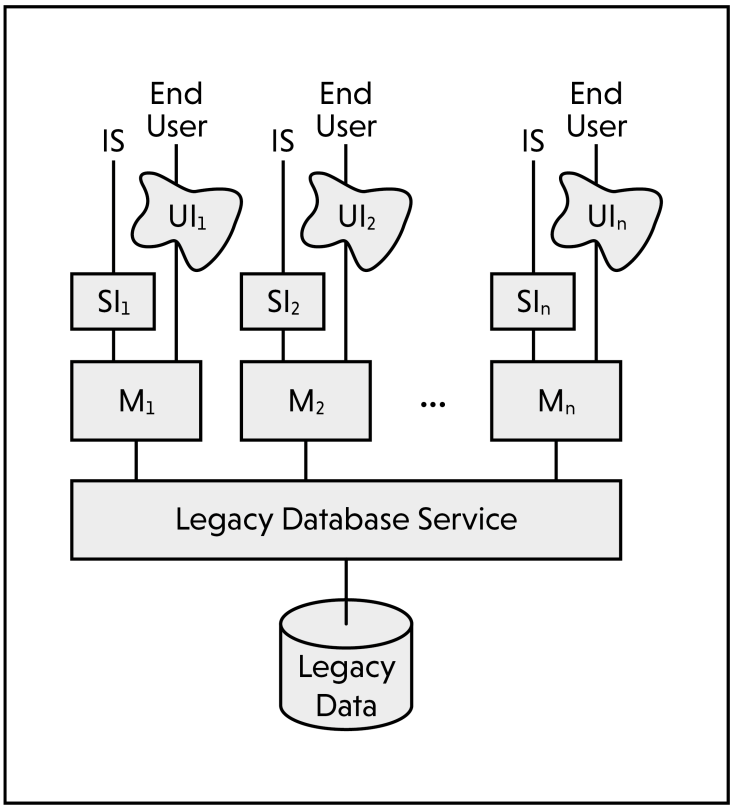

Abbildung 2.7: Architektur strukturierter Bestandssysteme [Vrgl. [Brodie and Stonebraker, 1993, S. 4]]

dessen und der Bewertung der technischen Schulden können gegebenenfalls Entscheidungen bezüglich einer *Migrationsstrategie* oder auch zu einer *Neuentwicklung* abgeleitet und begründet werden. [Vrgl. [Brodie and Stonebraker, 1993]]

2.2.1 Strukturierte Systeme

Bestandssysteme mit einer strukturierten Architektur (s. 2.7) zeichnen sich durch eine klare Trennung von Nutzerschnittstellen (UI), Anwendungslogik (M), Datenbankservices und Systemschnittstellen (SI) aus und nur die Datenbankservices haben Datenzugriff. Aufgrund ihrer Modularität eignen sie sich am besten für Modernisierungsvorhaben [Brodie and Stonebraker, 1993, S. 4].

2.2.2 Semi-strukturierte Systeme

Semi-strukturierte Bestandssysteme haben eine komplexere Struktur als strukturierte Systeme (s. 2.8). Die Anwendungslogik und Datenbankservices sind nicht voneinander separierbar und haben Zugriff auf den Datenbestand, während Nutzer- und Systemschnittstellen modularisiert sind. Aufgrund ihrer semi-optimalen Struktur wird ihre Analyse aufwändiger und schwieriger und ein Modernisierungsvorhaben fehleranfälliger [Brodie and Stonebraker, 1993, S. 5].

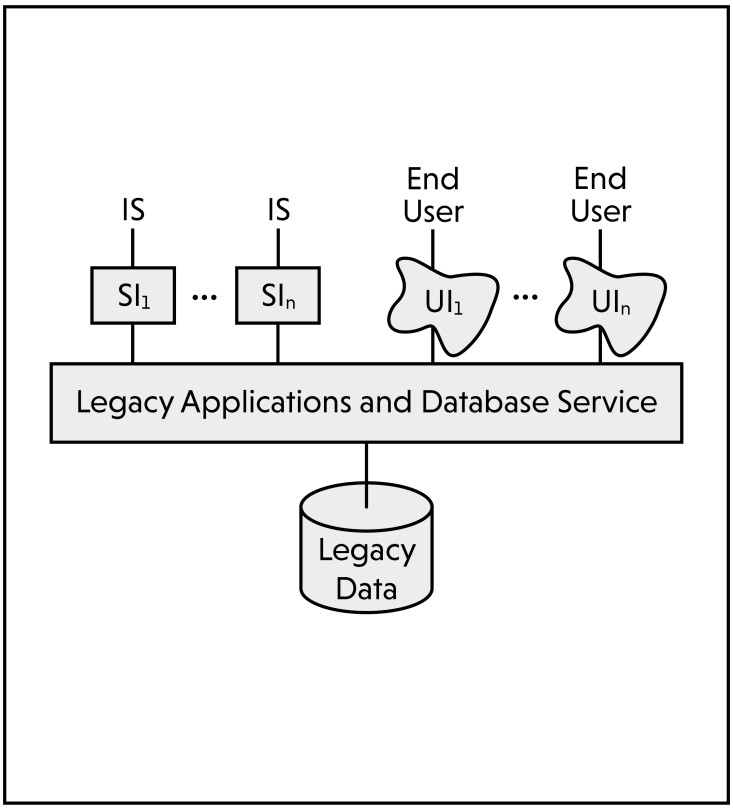

Abbildung 2.8: Architektur semi-strukturierter Bestandssysteme [Vrgl. [Brodie and Stonebraker, 1993, S. 5]]

2.2.3 Unstrukturierte Systeme

Der schlechteste Ausgangspunkt für die Modernisierung eines Bestandssystems sind Systeme mit unstrukturierten Architekturen [Brodie and Stonebraker, 1993, S. 5]. Sie sind eine *Blackbox*, deren Komponenten sich nicht separiert betrachten lassen (s. 2.9).

In der Praxis sind reine strukturierte, semi-strukturierte und unstrukturierte Architekturen selten. Durch ihre Weiterentwicklung entstehen Mischformen, wie beispielsweise in Abbildung 2.10 dargestellt.

2.3 Systemarchitekturen

Im Zusammenhang mit *Bestandssystemen* sind solche Hardware/Software-Kombinationen beziehungsweise Systemarchitekturen besonders relevant, die eine mögliche Modernisierung durch Weiterentwicklung oder Ablösung überdauert haben, da die veralteten Technologien untrennbar mit der dazugehörigen Hardware verbunden sind.

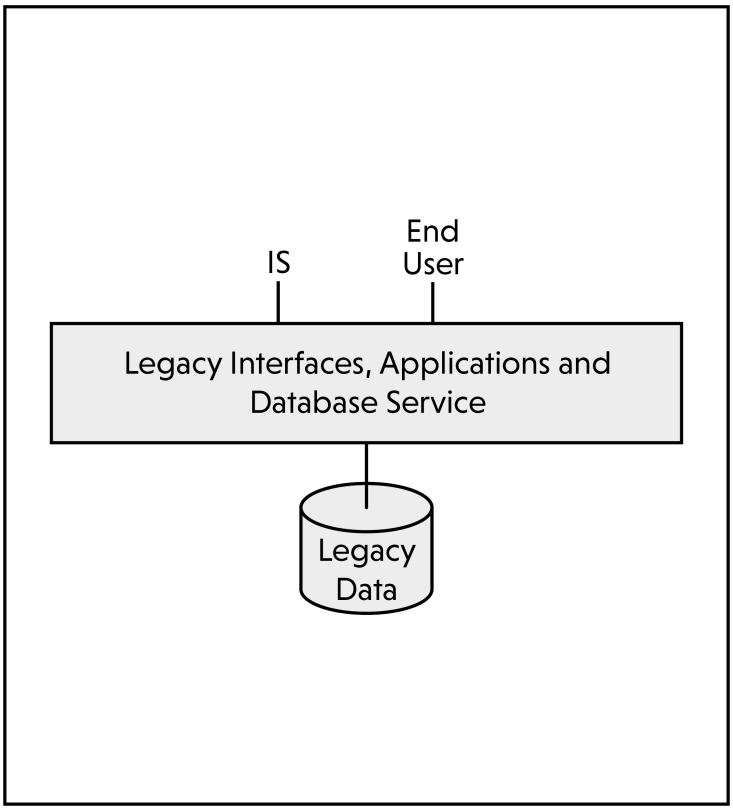

Abbildung 2.9: Architektur unstrukturierter Bestandssysteme [Vrgl. [Brodie and Stonebraker, 1993, S. 5]]

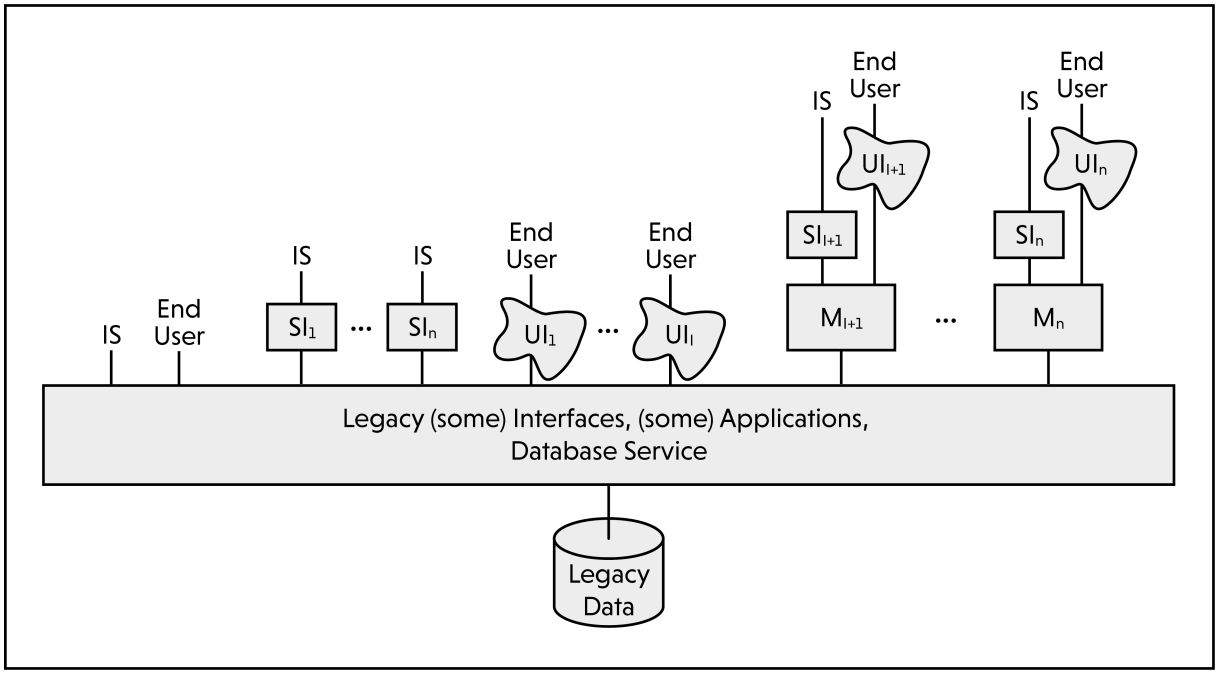

Abbildung 2.10: Architektur hybrider Bestandssysteme [Vrgl. [Brodie and Stonebraker, 1993, S. 6]]

2.3.1 Host/Mainfraime

In Geschäftsbereichen von Unternehmen, in denen die Massendatenverarbeitung und komplexe naturwissenschaftliche Berechnungen schon früh eine zentrale Rolle in Geschäftsprozessen einnahmen, wie beispielsweise in der Finanz- und Versicherungsbranche oder in der Forschung, wurden historisch und werden noch heute ein Großteil der Transaktionen auf einem sogenannten *Mainframe* (auch: Host) beziehungsweise Großrechner durchgeführt. Diese entstanden ursprünglich in den 1960er Jahren. Ein *Mainframe* ist ein leistungsstarker Zentralrechner auf dem alle Daten gespeichert und jegliche Programmlogik ausgeführt wird. Ihre gesamte Architektur ist auf einen spezifischen Zweck ausgelegt. An einen *Mainframe* sind Terminals angeschlossen, die allerdings keinerlei eigene Logik enthalten, sondern nur zur Absetzung von Befehlen an den Großrechner vorgesehen sind. Eigenschaften, die für die Nutzung von Großrechnern sprechen, sind:

- eine hohe Ein- und Ausgabeleistung
- effiziente Speicherverwaltung bei Transaktionen und Datenverwaltung
- hohe Verfügbarkeit
- Zuverlässigkeit

Ein *Mainframe* verursacht hohe Kosten in Anschaffung, Wartung und Betrieb. Im Falle der Abschaffung eines unternehmensinternen Großrechners entstehen Kosten durch die Bezahlung pro ausgeführter Transaktion auf dem *Mainframe* des jeweiligen Anbieters. Skalierung ist somit eine Frage des Budgets. [Vrgl. [Bogdan and Spruth, 2013], [Prüger, 2017]]

Mainframes sind eine dem Zweck angemessene Lösung, die in den vergangenen Jahrzehnten technologischen Fortschritten unterworfen war, so dass das Konzept des *Mainframe* weiter fortbesteht, der Großrechner selbst jedoch mittlerweile meist ein Teil der Cloud ist. [Vrgl. [IBM, 2020]]

Gemeinsam mit den Großrechnern entstanden die ersten Softwaresysteme in höheren Programmiersprachen, wie *COBOL* und *FORTRAN*, die deren Funktionen optimal unterstützen [Leitenberger, 2012]. Genauso wie der *Mainframe* durchliefen auch sie technologische Veränderungen, so dass beispielsweise *COBOL* mittlerweile objektorientiert verwendet werden kann.

Laut der Studie des [IDG, 2018] haben 70% der befragten Unternehmen noch *Mainframes* im Einsatz. Aus den Daten geht jedoch nicht hervor, ob an diesen zwischenzeitlich Modernisierungsmaßnahmen vorgenommen wurden. Von den 70% gab ein Drittel an, den *Mainframe* kurz bis langfristig ablösen zu wollen, wobei nur 29% angaben, dass sie genug interne Spezialisten besäßen, um ein Modernisierungsprojekt aus eigener Kraft bewältigen zu können [IDG, 2018, S. 35, 38].

2.3.2 Datenbank-Monolithen

Eine weitere Variante für Monolithen, die in der Praxis existieren, sind sogenannte Datenbank-Monolithen. Sie bauen in den Grundzügen auf dem Konzept eines Mainframe auf. Die gesamte Anwendungslogik wird von der Datenbank abgebildet und die angeschlossenen Clients dienen

zur Absetzung von Befehlen an die Datenbank und der Aufbereitung der anzuzeigenden Daten. Solche Datenbank-Monolithen sind meist durch die verwendete Hardware an die zugehörigen proprietären Programmiersprachen des Herstellers gebunden. Ein populäres Beispiel sind Oracle Datenbanken und PL/SQL (Procedural Language/Structured Query Language) [Oracle Corporation, 2020]. Obwohl die Verbindung von Abfragesprachen wie SQL und Programmierspachen mit modernen Konzepten, wie beispielsweise Objektorientierung, ein logischer Modernisierungsschritt ist, stehen Datenbank-Monolithen aufgrund ihrer Struktur vor ähnlichen Problemen, wie Mainframes und klassische monolithische Anwendungen. Sie verursachen durch Betrieb und Wartung hohe Kosten, sind schwer und nur kostspielig erweiter- und skalierbar, die ursprünglichen Anforderungen liegen verborgen in unstrukturiertem, komplexen Quellcode und die Implementierungen sind nicht von der Hardware trennbar.

2.3.3 Client/Server-Modell

Das Client/Server-Modell ist ein klassischer Ansatz zur Strukturierung verteilter Systeme. Es basiert darauf, dass ein Client über ein Netzwerk hinweg eine zur Verfügung gestellte Funktion eines Servers aufruft. Der Client ist somit der Dienstnutzer und der Server der Diensterbringer. Bestandteile des Modells können beide Rollen einnehmen. Der Ansatz des Client/Server-Modells findet sich in aktuellen Weiterentwicklungen wieder [Schill and Springer, 2007, S. 14]. Die typische Umsetzung des Modells erfolgt in Form einer Client/Server-Architektur. Die einzelnen Clients und Server bilden dabei meist recht grobgranulare Prozesse ab, haben einen eigenen Adressraum und spezifische Verwaltungsinformationen [Schill and Springer, 2007, S. 14]. Aus diesem Grund ist die Erzeugung und Anpassung einzelner Instanzen verhältnismäßig aufwändig. Durch die flexible Rollenverteilung der Elemente lässt sich eine Client/Server-Modell basierende Architektur gut kaskadieren und so flexibel auf große, hierarchisch gestaltete Systeme anwenden. Dies führt zur Aufteilung beider Ebenen in mehrstufige Architekturen. Auch weisen sie traditionell eine klare Trennung zwischen Clients, Anwendungslogik auf Serverseite und der Datenbene auf [Schill and Springer, 2007, S. 13-14].

2.3.4 Applicationserver

Applicationserver sind in mehrstufigen Architekturen die konkrete Realisierung von Middleware [Schill and Springer, 2007, S. 13]. Sie sind Arten von Servern, die umfassende Lösungen zur Ausführung von Anwendungslogik zur Verfügung stellen. Dazu gehören beispielsweise Laufzeitumgebungen, Mechanismen zur Kommunikation und Unterstützung bei der Verwaltung persistenter Daten, sowie die Steuerung von Transaktionen und deren Überwachung. Zur Laufzeit unterstützen Applicationserver typischerweise Komponentenmodelle wie EJB (Enterprise Java Beans) oder .NET [Schill and Springer, 2007, S. 400]. Darüber hinaus bieten Applicationserver durch Werkzeuge Unterstützung beim Softwareentwurf, der Überführung des Entwurfs in die Entwicklung, sowie der Installation und Ausführung von Softwarekomponenten. Darüber hinaus bieten sie

teilweise auch Schnittstellen zur Kopplung von Bestandssystemen an die moderneren Middleware-Umgebungen, die sie primär unterstützen. Dieser Vorgang wird Enterprise Application Integration (EAI) genannt [Schill and Springer, 2007, S. 24].

3 Optionen zur Modernisierung von Bestandssystemen

Anhand des zu bewertenden technischen Zustands des Bestandssystems ergeben sich unterschiedliche Optionen, wie und in welchem Umfang eine Ablösung durchgeführt werden kann. Auf oberster Ebene werden die Optionen grob nach Neuentwicklung und Migration unterschieden. Allerdings stellt eine Migration, ab einem gewissen Grad von durchgeführten Änderungen, faktisch eine Neuentwicklung unter dem Deckmantel des Begriffs einer Migration dar. [Vrgl. [Lilienthal, 2017b]]

Im Folgenden werden Ablösungsmöglichkeiten und -strategien von Bestandssystemen aufgelistet und beschrieben. Die Bewertung der Sinnhaftigkeit und Notwendigkeit der Anwendung dieser Möglichkeiten und Strategien wird im Folgekapitel vorgenommen.

3.1 Standardsoftware

Losgelöst von Migration und Neuentwicklung wird in der Literatur auch häufig die Ablösung von Bestandssystemen durch lizenzierte Standardsoftware erwähnt, was in der Praxis für die meisten Organisationen jedoch keine Option ist, da ihre speziellen Anforderungen oft nicht betriebsbereit zur Verfügung gestellt werden können [Kaps, 2017b]. Standardsoftware ist dann eine gute Wahl, wenn die Geschäftsprozesse sich einfach in die Anwendung einpflegen lassen und die Differenz der zur Verfügung gestellten Funktionen und den Anforderungen dabei möglichst gering gehalten werden kann [Ballüder et al., 2020].

Um diese Differenz auszugleichen bieten die Softwarehersteller, zusätzlich zu ihrem Anwendungsportfolio, Beratungsleistungen, Implementierungsteams für Integration und individuelle Anpassungen und Support-Verträge [SAP Germany GmbH, 2020].

Vorteil der Ablösung des Systems durch eine Standardsoftware ist, dass die technische Aufarbeitung und Sanierung des Bestandssystems entfällt [Lilienthal, 2017b]. Dadurch ist der Zustand des abzulösenden Bestandssystems bei diesem Modernisierungsvorgehen nicht relevant.

3.1.1 Software-Bibliotheken

In Softwaresystemen finden sich häufig kleinteilige, händisch umgesetzte Standard-Funktionen und -Elemente, die Bestandteil von verbreiteten, umfassenderen Drittanbieter- oder Open-Source-Bibliotheken sind. Durch den Einsatz solcher Bibliotheken kann die Summe des selbst geschriebenen Codes verringert werden, was wiederum das Risiko von Implementierungsfehlern verringert. Darüber hinaus müssen für den verwendeten Quellcode keine zusätzlichen Tests implementiert werden, da dieser von den Community-EntwicklerInnen getestet wird. Auch haben Bibliotheken, die von einer größeren Community gepflegt werden, in der Regel einen höheren Funktions- und Reifegrad, als Eigenentwicklungen. Dieser umfassende, zusammenhängende Funktionsumfang zu einem spezifischen Thema kann beim Einsatz daher auch zukunftsperspektivistisch die (Weiter-) Entwicklung und Wartung von Softwaresystemen unterstützen. [Vrgl. [Basedow, 2017]]

3.2 Neuentwicklung

Bei einer Neuentwicklung handelt es sich um Softwareentwicklung "auf der grünen Wiese". Bestehenden Daten werden neue Strukturen zugewiesen, wobei deren semantische Bedeutung erhalten bleibt. Das Wissen um bisherige Prozesse wird verwertet und der Quellcode komplett neu geschrieben [Kaps, 2017b]. Eine Neuentwicklung ist ein kosten- und zeitintensives Vorhaben, das Fachabteilungen und EntwicklerInnen gleichermaßen intensiv fordert und ein hohes Risiko birgt [Ballüder et al., 2020].

Eine Neuentwicklung ist dann besonders sinnvoll, wenn sich Geschäftsmodelle, wie beispielsweise klassische Versicherungspolicen zu einem On-Demand-Modell entwickeln sollen [Ballüder et al., 2020].

[Lilienthal, 2017b] behauptet, dass eine Ablösung durch eine Neuentwicklung den Vorteil hat, dass keine technische Aufarbeitung und Sanierung des Bestandssystems vorgenommen werden muss und folglich der Zustand des Bestandssystems bei dieser Option nicht relevant ist. Tatsächlich lässt sich, bezogen auf das allgemeine Vorgehen in dieser Ausarbeitung, diese Aussage pauschal ohne die Spezifizierung einer Migrationsstrategie (s. 4) nicht bestätigen.

3.3 Migration

Eine Migration wird dem Feld des Software (Re-)Engineering zugeordnet. Es ist eine technische Transformation mit klar definierten Anforderungen, die durch das Altsystem beschrieben werden. Die Fachlichkeit verändert sich in der Regel nicht. [Vrgl. [Kaps, 2017b], [Gimnich and Winter, 2005]] Das System und seine Daten sollen in neue Umgebungen überführt werden. Dabei sollen alle Funktionalitäten beibehalten, der Ressourcenverbrauch vermindert und Wart- bzw. Änderbarkeit der Software signifikant verbessert werden. Außerdem soll das System dann den gestiegenen, neuen Anforderungen gerecht werden können. [Vrgl. [Schmidt and Bär, 2014], [Brodie and Stonebraker,

1993]] Die System- und die Datenmigration sind eigenständige, doch eng miteinader verknüpfte Prozesse. [Vrgl. [Erdle, 2005]]

Nach [Gimnich and Winter, 2005] unterteilen die sich gegenseitig bedingenden Migrationsprozesse feiner und unterscheiden zwischen technischen Migrationen:

- **Hardware-Migration**, bspw. die Überführung vom Mainframe auf Unix
- **Migration der Laufzeitumgebung**, bspw. Wechsel des Betriebssystems oder des Datenbankmanagementsystems
- **Architektur-Migration**, bspw. vom Monolith zur Mehrschichtenarchitektur
- **Migration der Entwicklungsumgebung**, beispielsweise von COBOL zu Java

und Migrationen von Softwareaspekten:

- **Datenmigration**
- **Programmmigration**
- **Benutzungsschnittstellen-Migration**

Neben ihrem primären Ziel, das Altsystem abzulösen, erarbeiteten [Erdle, 2005] und [Kaps, 2017a] auf der Grundlagenliteratur von [Brodie and Stonebraker, 1993] folgende zusätzliche Mindestanforderungen an eine Migration, sowie zusätzlich zu bedenkende Fragestellungen:

- **Garantie des Betrieb**: Während der Migration muss bis zur endgültigen Ablösung und Abschaltung des Altsystems ein ununterbrochener, sicherer und zuverlässiger Betrieb gewährleistet werden, da schon kürzeste Ausfälle zentraler Systeme zu massiven finanziellen Verlusten führen können.
- **Einplanung aktueller und zukünftiger Anforderungen**: Da eine Migration im schlechtesten Fall Jahre dauern kann, müssen aktuelle und in naher Zukunft umzusetzende Anforderungen schon während der Migration ebenfalls eingeplant und umgesetzt werden, so dass das Zielsystem nicht kurz nach Fertigstellung erneut angepasst oder unter Umständen migriert werden muss.
- **Risikoeinschätzung von Änderungen**: Mit der Komplexität einer Migration, also der Anzahl der durchgeführten Änderungen, steigt das Risiko für Fehler. Der logische Schluss ist daher, den Umfang der Migration so gering wie möglich zu halten, um das Risiko kontrollieren zu können. Die frühzeitige Integration von automatisierten Testroutinen kann helfen, um Fehlerentwicklung vorzubeugen und negative Auswirkungen von Änderungen zu minimieren. Grundsätzlich sind nur die Änderungen vorzunehmen, die notwendig sind, um aktuelle und zukünftig zu erwartende Anforderungen abzudecken.

  - **Code (nicht) ändern**: Wenn keine Änderung der benutzten Programmiersprache nicht vorgesehen ist und keine Notwendigkeit der Implementierung neuer Funktionalitäten besteht, kann eine direkte Codeübernahme durchgeführt werden. Jede Änderung erhöht das Risiko der Migration. Wenn jedoch, mit angemessenem Aufwand, durch Änderungen am Code, die Migration vereinfacht oder untersützt werden kann, können solche Änderungen trotzdem durchgeführt werden. In der Praxis scheitert die Übernahme von Code jedoch häufig aufgrund der schlechten Qualität des Altsystems.

- **Anwendung moderner Technologien und Methoden**: Moderne Methoden und Technologien sollten so angewendet werden, dass das Zielsystem so flexibel wie möglich gestaltet werden kann und potenziell negative Auswirkungen minimiert werden können. Zukünftige Änderungen und Entwicklungen können so erleichtert werden. Systemwerte, wie Performance und Sicherheitslücken, die die Entscheidung der Migration begünstigt haben, können so meist direkt positiv beeinflusst werden.

- **(Nach-)Dokumentation**: Fehlende oder nicht aktuelle Dokumentation von Anwendungen und Funktionen ist eines der größten Probleme von Altsystemen. Eine Nachdokumentation, um eine Bestandsaufnahme des Systems machen zu können, muss aufwändig mittels Reverse-Engineering-Technik erfolgen. Aus den gewonnenen Informationen über die Funktionen und die Architektur des Altsystems lassen sich Grundlagen für die Entwicklung neuer Software ableiten, eine Bewertung des (zu übernehmenden) Funktionsumfangs wird möglich und Dateninhalte können (auf Weiterverwendung) analysiert werden.

Ähnlich, wie bei der Neuentwicklung, ist es von der gewählten Migrationsstrategie abhängig, ob der Zustand des zu migrierenden Systems Auswirkungen auf die Migration hat.

[Kaps, 2017a] nennt noch weitere übergreifende Themen, die vor einem Migrationsbeginn zu bedenken sind:

- **umfassende Projektplanung, -leitung, -management und -kalkulation**, um das Migrationsvorhaben und die Kosten, die es verursacht, überwach- und kontrollierbar zu halten

- **Risikomanagement** und in die damit zusammenhängenden Routinen für die Behandlung von Fehlern und Störungen im Betrieb

- **Prozess(re)modellierung** in Zusammenarbeit mit Fachbereichen und AnwenderInnen, um bestehende Prozesse verbessern oder ablösen zu können

- **CI/CD: Continuous Integration and Delivery**, um Änderungen regelmäßig überprüfen und automatisiert in den bestehenden Quellcode integrieren zu können

- **Änderungsmanagement**, somit die Koordinierung von Anforderungen während einer Migration

- **Qualitätssicherung**, um das Risiko erneut ein Legacy-System zu entwickeln zu senken

- **Dokumentation**, soviel wie nötig und so wenig wie möglich

3.3.1 Oberflächen

Die Migration beziehungsweise Überarbeitung von Oberflächen ist erfahrungsgemäß nach spätestens drei bis fünf Jahren notwendig. Die Sehgewohnheiten und Erwartungen von Nutzern an eine Bedienoberfläche ändern sich in dieser Zeitspanne so sehr, dass eine Neugestaltung des Graphical User Interface (GUI) erforderlich wird. [Aschoff, 2017] nennt als Gründe die ständigen Weiterentwicklungen der GUI-Konzepte, wie Flat und Material-Design, sowie Card Layouts und GUI-Pattern, wie Progress Steps, Infinte Lists und viele weitere. Diese Wahrnehmungsänderungen gab es schon immer, zusätzlich beschleunigt werden sie durch die Erwartungshaltung der Nutzer durch die unmittelbare Umsetzung der Neuentwicklungen in Social-Media-Anwendungen und im E-Commerce.

Trotzdem sind Nutzer gleichzeitig eng verbunden mit ihrer täglichen Routine, so dass es das Ziel sein muss, ein Oberflächen-Redesign so weit aufzuteilen, dass der Umfang der gleichzeitigen Änderungen möglichst gering ausfällt. Das Redesign lässt sich in zwei voneinander unabhängigen Dimensionen zerlegen, die den Funktionsumfang bestimmende Breite und die Tiefe, die den Grad des Eingriffs in bestehende Softwareschichten definiert. Für die Aufteilung in der Breite empfiehlt es sich, beispielsweise anhand des Model-View-Presenter-Patterns (MVPP) einen Mindestfunktionsumfang für den ersten Schritt der Migration zu definieren. Zur Orientierung in und Gliederung der Tiefendimension benennt [Aschoff, 2017] das Gradual-Feature-Release-Pattern. Dieses besteht aus vier aufeinander aufbauenden Stufen, die jeweils tiefer in den bereits bestehenden Code eingreifen.

1. **Facelift**: Funktionen, die die Funktion der Software nicht beeinflussen, wie Änderungen am Layout (Flächen, Formen, Farben, Positionen) und der Austausch von Bildern, Grafiken, Icons, sowie die Anpassung sonstiger kleinerer Effekte.

2. **Quick Wins**: Kleine Verbesserungen der Usability, wie die Vereinheitlichungen von Bezeichnungen, die Anpassung der Reihenfolge von Elementen in Formularen oder die Änderung von Zeilen- und Spaltenbreiten in Tabellen. Der Übergang vom Facelift zu den Quick Wins ist fließend.

3. **Usability-Optimierungen**: Tiefergreifende Oberflächenoptimierungen, wie Änderungen an Menüstrukturen um Klickwege, auch in die Programmtiefe, zu verkürzen oder die Überarbeitung von Formularstrukturen. Der Übergang von Quick Wins zu den Usability-Optimierungen ist ebenfalls fließend.

4. **UX-Verbesserungen**: Um die User Expierence zu verbessern, müssen häufig bestehende Prozesse und Workflows komplett oder in Teilen überarbeitet werden.

3.3.2 Prozessevaluierung

Die Evaluierung bestehender Prozesse ist ein Thema, das im Zusammenhang mit der Modernisierung von Softwaresystemen häufig genannt wird, beispielsweise bei [Kaps, 2017a], der auch Bewertungswerkzeuge, wie die Portfolio-Analyse nennt. Eine Prozessevaluierung ist jedoch ein

organisatorisches Thema und somit aus der Betrachtung und Bewertung dieser Ausarbeitung ausgeschlossen.

3.4 Datenbankmigration

In der Literatur beinhaltet der Prozess einer Migration oder Neuentwicklung in den meisten Fällen auch den Vorgang einer Datenbankmigration. Das Ziel ist, durch die Erneuerung des Datenbankschemas und des Datenbankmanagementsystems (DBMS) das Verwalten von Daten nach modernen Standards und auf den Anwendungsfall zugeschnitten zu optimieren. Ein vorbereitender Schritt zu einer Datenbankmigration ist der (temporäre) Einsatz eines modernisierten Datenzurgiffslayers (auch: Database Gateway). In der Praxis kommt es jedoch vor, dass die eingesetzten Gateways nach der System-Migration bestehen bleiben, somit Teil des Zielsystems werden. Gründe dafür können ein für den Anwendungsfall bereits optimiertes Datenbankschema, die hohe Qualität und Leistungsfähigkeit des implementierten Gateways und auch Bemühungen Kosten einzusparen sein.

3.4.1 Vorwärtsmigration (Database First)

Bei einer Vorwärtsmigration wird initial vor dem Beginn der System-Migration die Datenbank migriert. Nach der Übertragung der Altdaten auf die neue Datenbank wird diese für das Altsystem, über die Implementierung sogenannter Forward Gateways, zugänglich gemacht. Diese werden erst nach der vollständigen System-Migration deaktiviert. Vorteile dieser Methode sind die uneingeschränkte Möglichkeit der Nutzung zusätzlicher Funktionen des neuen Datenbanksystems, auch in Verbindung mit dem Altsystem und die auf das neue Datenbanksystem abgestimmte Entwicklung des Zielsystems. Ein Nachteil sind die hohen Aufwände, mit denen die Gateways im Nachhinein nur noch angepasst werden können, weshalb vor der Migration eine exakte Planung des neuen Datenbanksystems stattfinden muss. Ein weiterer entscheidender Nachteil ist die Zeitspanne, in der die Migration der Daten durchgeführt wird. Es besteht die Möglichkeit, dass die Datenmenge zu groß ist, um diese in angemessener Zeit zu übertragen. Da während dieser Übertragung keine Änderungen an den Daten vorgenommen werden können, ist eine Systemdowntime des Altsystems notwendig, was bei unternehmenskritischen Anwendungen nicht abschätzbare finanzielle Ausfälle bedeuten kann. Auch die Forward Gateways können zu einem Nachteil werden, wenn die abzubildenden Funktionen ein so hohes Komplexitätslevel haben, dass die initialen Entwicklungsaufwände unkontrollierbar werden und die Implementierung der Gateways sogar gänzlich scheitern könnten. Um dieses Scheitern so gut es geht abzuwenden, empfiehlt es sich dieses Vorgehen nur auf Altsysteme anzuwenden, die von vornherein eine definierte und strikte Trennung von Anwendung und Datenbankschnittstelle haben. [Vrgl. [Erdle, 2005], [Schmidt and Bär, 2014], [Kaps, 2017b], [Kaps, 2017a]]

3.4.2 Rückwärtsmigration (Database Last)

Bei einer Rückwärtsmigration wird zunächst die System-Migration durchgeführt und erst nach dessen Vervollständigung werden die Datenbestände auf die neue Datenbank übertragen. Während der System-Migration greifen die neu implementierten Funktionalitäten auf die Altdaten über sogenannte Backward oder Reverse Gateways zu. Bei diesem Vorgehen ist eine iterative System-Migration möglich. Der Vorteil einer Rückwärtsmigration ist, dass das Altsystem bis zur endgültigen Abschaltung ohne Modifikationen weiter genutzt werden kann. Auch bei der Rückwärtsmigration kann die die kosten- und zeitintensive Abbildung der Funktionen des neuen Datenbanksystems ein entscheidender Nachteil sein, so dass auch diese Migration nur auf Altsysteme angewendet werden sollte, die von vornherein eine definierte Datenbankschnittstelle besitzen. [Vrgl. [Erdle, 2005], [Schmidt and Bär, 2014]]

3.4.3 Allgemeine Migration

Bei einer allgemeinen Migration handelt es sich um eine Verschmelzung von Vorwärts- und Rückwärtsmigration. Über einen Coordinator und eine Mapping Table werden Anfragen von Alt- und Zielsystem an die entsprechenden Forward beziehungsweise Reverse Gateways weitergeleitet, die dann ihrerseits die Verbindung mit der Datenbank herstellen. Der größte Vorteil einer allgemeinen Migration ist die Vermeidung von Ausfällen im Betrieb in jeder Phase, so dass sie auch auf unternehmenskritische Anwendungen verwendet werden kann. Außerdem nutzt sie alle Vorteile der Vorwärts- und Rückwärtsmigration, wobei sie deren Nachteile größtenteils kompensiert. Dennoch entfallen nicht die gegebenenfalls hohen Implementierungsaufwände der Gateways. [Vrgl. [Erdle, 2005]]

3.4.4 Zusätzliche Problemstellung

Unabhängig von der Systemmigration und der gewählten Datenbankmigrationsmethode muss die Möglichkeit von Datenanpassungen betrachtet werden. Zu diesen Anpassungen zählen unter anderem Konsolidierungen oder auch Datentypänderungen. Es muss somit nicht nur ein Mapping der Datenbankschemata erfolgen, sondern auch eine Umschlüsselung von existierenden zu zukünftigen Daten und Datenformaten umgesetzt werden, was die Entwicklung von Gateways zusätzlich erschwert, da diese zusätzlich zum Datenbankzugriff auch die Daten-Transformation abbilden müssen. [Vrgl. [Erdle, 2005]]

	Database First	**Database Last**	**Allgemeine Migration**
Vorteile	- Nutzung der neuen DB-Funktionen mit dem Altsystem - auf die Datenbank abgestimmte Entwicklung des Zielsystems	- Altsystem kann bis zur Abschaltung unverändert genutzt werden - begünstigt eine iterative System-Migration	- Vermeidung von Ausfällen im Betrieb des Altsystems in jeder Phase - Kompensation der meisten Database-First und -Last Nachteile
Nachteile	- hohe Entwicklungs-aufwände der Gateways - exakte und vollständige Planung erforderlich - Zeitspanne der Daten-Migration - nur auf Altsysteme mit definierter DB-Schnitt-stelle anwendbar	- hohe Entwicklungs-aufwände der Gateways - Zeitspanne der Daten-Migration - nur auf Altsysteme mit definierter DB-Schnitt-stelle anwendbar	- hohe Entwicklungs-aufwände der Gateways

Tabelle 3.1: Vergleich Datenbankmigrationen

4 Strategien für Modernisierungsvorhaben

Der Begriff der Migrationsstrategie (auch: Übergabe- oder Transformationsstrategie) wird unabhängig von der Modernisierungsmaßnahme verwendet, somit auch unabhängig davon, ob es sich um einen Austausch durch Standardsoftware, eine Migration oder Neuentwicklung handelt. Die Migrationsstrategie wird auf das jeweilige Szenario angepasst, um deren Vorteile möglichst schnell und einfach zu erreichen. [Vrgl. [Wolff, 2017]]

Alle in diesem Abschnitt zitierten Quellen und ihre Interpretationen des Umgangs mit Migrationsstrategien basieren maßgeblich auf der Grundlagenliteratur von [Brodie and Stonebraker, 1993].

4.1 Big Bang/Cold Turkey

Die Begriffe *Big Bang* beziehungsweise *Cold Turkey* beschreiben die Ablösung eines Altsystems, durch eine komplette Neuentwicklung, mit dem Ziel die Vorteile moderner Entwicklungsmethoden, Architekturen, Tools, Datenbanken sowie Hardware Plattformen in Gänze nutzen zu können. Im Rahmen dieser Strategie wird eine vollständige Neuentwicklung des Systems durchgeführt. Die Entwicklung und der Test des Zielsystems finden parallel zum Betrieb des Altsystems statt. Da die kontinuierliche Verbesserung der bestehenden Geschäftsprozesse im Zeitraum der Entwicklung meist nicht ausgesetzt werden kann, weil die aktuellen Bedürfnisse des Unternehmens erfüllt werden müssen, werden diese gleichzeitig im Neu- und Altsystem umgesetzt. Nach Abschluss der Entwicklungsarbeiten und erfolgreichem Test wird das Altsystem deaktiviert und das Zielsystem übernimmt seine Anforderungen. Der komplette Systemwechsel als finaler Schritt, somit wenn der Big Bang der *Single Point of Failure* ist, stellt ein enorm hohes Risiko dar und kann zu existenzgefährdenden Situationen führen [Kaps, 2017b]. Im Falle einer Datenbankmigration muss die Übertragung aller Daten auf einmal direkt vor dem Systemwechsel geschehen (Database-Last). Die benötigte Zeit für diesen Vorgang hängt maßgeblich von der Masse der zu migrierenden Daten und den gegebenenfalls durchzuführenden Transformationen ab. Während dieser Zeit sind keine Änderungen an den Daten möglich und somit das Altsystem nicht benutzbar. [Vrgl. [Brodie and Stonebraker, 1993], [Kaps, 2017b], [Schmidt and Bär, 2014], [Erdle, 2005]]

4.1.1 Bewertung

Bei der Verfolgung der Big Bang/Cold Turkey-Strategie können bei einem optimalen Ablauf die Kosten für eine komplette Neuentwicklung durch ein schnelles Entwicklungsprojekt verhältnismäßig gering gehalten werden, obwohl der Neuentwicklung gerade diese als negative Eigenschaften

nachgesagt werden. Als Neuentwicklungs-Vorteil kann die Entwicklung *auf der Grünen Wiese*, also die absolute technische Freiheit zur Lösung der Anforderungen genutzt werden. Für diese Strategie ist der Zustand des Altsystems nicht relevant. Durch den interaktionslosen Parallelbetrieb zwischen Alt- und Zielsystem entstehen keine zusätzlichen Entwicklungsaufwände. Bei der Realisierung ist jedoch die Entwicklung von Datengateways notwendig, wenn eine Datenmigration vorgenommen werden soll. Problematisch am Parallelbetrieb ist die Notwendigkeit der Umsetzung von Anforderungsänderungen in beiden Systemen. Die kritischen Risikofaktoren dieser Strategie sind jedoch andere. Der erste Risikofaktor ist der vollständige Umstieg zu einem zentralen Zeitpunkt, der im Fehlerfall kritische finanzielle Ausfälle verursachen kann. Der zweite Faktor ist gegebenenfalls die Datenbankmigration, die durch die notwendige Gewährleistung des Betriebs der Altanwendung bis zur Umstellung als umzusetzende Database-Last-Strategie vorgesehen ist, so dass der Betrieb während der Datenmigration pausiert werden muss. Dies kann bei geschäftskritischen Anwendungen zu extremen finanziellen Verlusten führen und somit für die Organisation nicht akzeptabel sein. Eine Alternative wäre die Anwendung der allgemeinen Migrationsstrategie, die den Betriebsausfall verhindern würde, jedoch zusätzliche Entwicklungsaufwände für die benötigten Gateways und den Coordinator bedeuten. Vorteil eines erfolgreichen *Big Bang* ist die sofortige Sicht- und Anwendbarkeit des Nutzen für die Anwender und die Organisation, die aufgrund des großen und eventuell zeitintensiven Projektvorhabens kurzweilig sein kann, wenn das Modernisierungsvorhaben während des Einsatzes technisch oder architekturell wieder überholt ist.

4.2 Chicken Little

Die Chicken Little-Strategie zerlegt die Gesamtmigration in Einzelmigrationen, die inkrementell und iterativ durchgeführt werden können. Diese kleinen, einzelnen Schritte sind eine etablierte Best Practice, da sie das Risiko von Fehlschlägen kontrollierbar machen und der Migration allgemein reduzieren [Wolff, 2017]. Voraussetzung für eine erfolgreiche Durchführung ist jedoch ein tiefgreifendes Verständnis des Altsystems, um dieses in unabhängige Module zu zerlegen und insbesondere die Datenbankschnittstelle herauslösen zu können. Zur Übersetzung und Abschirmung von Daten zwischen den Komponenten von Alt- und Zielsystem, sowie zur Koordination der betroffenen Komponenten und zur Sicherstellung der Konsistenz bei Änderungen kommen temporäre Gateways zum Einsatz. Im Falle einer Datenbankmigration sind sowohl eine Vorwärtsmigration (Database-First), als auch eine Rückwärtsmigration (Database-Last) möglich sowie eine Verschmelzung dieser beiden Verfahren (allgemeine Migration). Auch die Chicken Little-Strategie bedeutet eine komplette Neuentwicklung des Systems. [Vrgl. [Brodie and Stonebraker, 1993], [Kaps, 2017b], [Gimnich and Winter, 2005], [Schmidt and Bär, 2014], [Erdle, 2005]]

4.2.1 Bewertung

Die Chicken Little-Strategie macht sich die Vorteile der inkrementellen und iterativen Softwareentwicklung zu Nutze. Sie macht durch viele, kleine Einzelmigrationen das Risiko des Modernisierungsvorhabens kontrollier- und steuerbar und erhöht die Flexibilität bei sich ändernden Anforderungen,

die direkt im Zielsystem umgesetzt werden können. Die Risikokontrolle kann allerdings nur effektiv auf Altsysteme, über die ein tiefgreifendes Verständnis vorherrscht, angewendet werden, um es so um es so modularisieren zu können, dass diese Einzelmigrationen ermöglicht werden können. Dies sind Schritte zu einer kompletten Neuentwicklung. Es entstehen zusätzliche Entwicklungsaufwände für die Gateways zur Kommunikation zwischen Alt- und Zielsystem. Diese können beliebig komplex werden und deren Fehlfunktionen sind die wesentliche Risikoquelle des Chicken Little-Ansatzes. Im Falle einer Datenbankmigration kann eine beliebige Strategie gewählt werden, die an ihren eigenen Vor- und Nachteilen bewertet werden kann. Der Projektverlauf ist im Verhältnis zu einem erfolgreichen *Big Bang* länger. Es besteht jedoch ein geringeres Risiko, welches sich genau wie die weiteren Vorteile durch eine hohe Anzahl von Interaktionsvorgängen zwischen Alt- und Zielsystem erkauft wird. Der Nutzen der Modernisierung und somit des Zielsystems steigt während des Entwicklungsprozesses kontinuerlich für die Anwender und die Organisation.

4.3 Butterfly

Bei der Butterfly-Strategie handelt es sich um eine Mischung des Big Bang- und des Chicken Little-Ansatzes. Die Daten und deren Handhabung und letztendlich die Migration dieser Daten sind bei dieser Strategie zentral. Eine Kooperation zwischen Alt- und Zielsystem wird als nicht notwendig angesehen, somit müssen keine Gateways entwickelt und verwendet werden. Angelehnt an das Database-First-Vorgehen werden die Daten des Altsystems eingefroren und notwendige Veränderungen in temporären Speichern vorgenommen. Der Data Access Allocator leitet Zugriffe entsprechend durch. Informationen, auf die bisher noch nicht zugegriffen wurden, werden aus der Datenbasis geholt und gemeinsam mit den Änderungen in den temporären Speicher geschrieben. Wenn bereits auf die Daten zugegriffen worden ist, wird der Zustand der Daten aus dem temporären Speicher verwendet. Die Migration der Datenbank wird iterativ und inkrementell durchgeführt. Zunächst werden die schreibgeschützten Daten in das neue Datenbankschema überführt und dann die temporär gespeicherten Daten nach migriert. Im letzten Schritt erfolgt dann die Ablösung des Altsystems durch das Zielsystem. Dies erfolgt gemeinsam mit der letzten Überführung des temporären Speichers in das neue System, der die Konsistenz des Datenbestands zwischen Alt- und Zielsystem sicherstellt. Im Fehlerfall leitet der Data Access Allocator die Zugriffe auf den zuletzt verwendeten temporären Speicher um. [Vrgl. [Brodie and Stonebraker, 1993], [Schmidt and Bär, 2014], [Erdle, 2005]]

4.3.1 Bewertung

Die Verwendung der Butterfly-Strategie nutzt die Vorteile der Big Bang- und Chicken Little-Strategie und wendet diese auf das Vorhaben der reinen Datenmigration an. Da für diese keine Kommunikation zwischen Alt- und Zielsystem nötig ist, fallen dafür auch keine Entwicklungsaufwände an. Für die iterative, inkrementelle Datenüberführung über die Temporärspeicher muss jedoch der Data Access Allocator implementiert werden, dessen Komplexität maßgeblich von dem Zustand der zu migrierenden Daten und den notwendigen Anpassungen bei der Überführung abhängt. Die

Entwicklung kann daher eine beliebiges Komplexitätslevel annehmen und auch die Kosten für den benötigten temporären Speicher können ein kritischer Faktor sein. Nachdem der Aufwand und das Risiko der Entwicklung des Data Access Allocators bewältigt wurde, wird durch das inkrementelle Vorgehen auch bei der Butterfly-Strategie das Risiko der weiteren Migrationsvorgänge beherrschbar. Ob die Systemumstellung ebenfalls inkrementell oder zu einem zentralen Zeitpunkt stattfindet, ist bei dieser Methode anhand der jeweils innewohnenden Vor- und Nachteile zu bewerten. Von dieser Umstellung hängt dann auch der Anstieg des Nutzens für die Organisation und die Anwender ab. Der Vorteil dieser Methode ist der ständige Zugriff auf den Datenbestand während der Migration und im finalen Schritt kann die Umschaltung zum Zielsystem ohne Unterbrechung des Betriebs vorgenommen werden. Nachteil der Strategie ist die Abhängigkeit des benötigten Temporärspeichers von der womöglich hohen Aktivität im Altsystem. Trotz der Einsparung der Entwicklung der Gateways muss der Aufwand für die Data Access Allocator Komponente bedacht werden.

4.4 Validierung von Migrationsstrategien und Modernisierungsoptionen

Im Folgenden werden die beschriebenen Migrationsstrategien in Kombination mit den Modernisierungsoptionen auf Sinnhaftigkeit überprüft. Die Ergebnisse wurden in der Tabelle 4.1 zusammengefasst.

4.4.1 Standardsoftware

Bei der Umstellung auf eine Standardsoftware ist der Zustand des abzulösenden Bestandssystems nicht relevant, da eine Kommunikation zwischen den Softwaresystemen nicht benötigt wird. Der zu bedenkende Punkt ist die notwendige Datenbankmigration, um die bestehenden Daten für die Anbindung an das gekaufte System optimal zur Verfügung stellen zu können. Es kommen auf Basis dieser Anforderungen zwei Strategien zur Umstellung auf eine Standardsoftware in Frage.

Im Rahmen der Big Bang-Strategie kann mit einer Standardsoftware auf die Entwicklung des Zielsystems komplett verzichtet werden. Anhand der Gestaltung des Datenbestandes des Altsystems und der Art, wie die Standardsoftware Daten entgegennimmt, kann eine Datenbankmigration erforderlich sein, die in dieser Strategie grundsätzlich vorgesehen ist. Eine Alternative wäre die Implementierung eines Datenzugriff-Gateways, die kein Bestandteil der Big Bang-Strategie ist, doch bei einer flexiblen Auslegung dieser, als Alternative zu der Datenbankmigration verwendet werden könnte.

Die Anwendung der Butterfly-Strategie eignet sich besser. Diese ist gänzlich auf die benötigte Datenmigration ausgelegt. Die Migration kann gemeinsam mit der Inbetriebnahme der Standardsoftware beginnen. Durch den Data Access Allocator können die Daten dann schrittweise in die neue Umgebung übertragen werden, während das Bestandssystem, nachdem der DAA die Arbeit aufgenommen hat, direkt abgeschaltet werden kann. Die Abschaltung des DAA erfolgt dann nach der vollständigen Datenmigration.

Die Chicken Little-Strategie ist als inkrementelles und iteratives Vorgehen für die Ablösung eines Softwaresystems vorgesehen. Somit ist sie im Zusammenhang mit der Umstellung auf eine Standardsoftware nicht zielführend.

4.4.2 Neuentwicklung

Bei einer Neuentwicklung ist der Zustand des Bestandssystems gegebenenfalls in Abhängigkeit zu der gewählten Modernisierungsstrategie relevant. Bei dieser geht es primär um die Verwertung bisheriger Prozesse und der Neustrukturierung der bestehenden Daten, wobei der Quellcode komplett neu geschrieben wird. Diese Anforderungen können grundsätzlich mit der Unterstützung aller betrachteten Strategien erfüllt werden.

Bei der Big Bang-Strategie wird die geforderte komplette Neuentwicklung des Zielsystems durchgeführt. Da keine Kommunikation zwischen Alt- und Zielsystem stattfinden muss, ist der Zustand des Altsystems aus technischer Sicht nicht relevant. Ob die Datenmigration nach der Database-Last-Strategie durchgeführt wird oder die zusätzlichen Entwicklungsaufwände in Kauf genommen werden, um den womöglich inakzeptablen, langen Betriebsausfall durch den Einsatz einer allgemeinen Datenmigration zu umgehen, ist eine organisatorische Entscheidung.

Chicken Little eignet sich gemeinsam mit einem iterativen und inkrementellen Entwicklungsprozess gut zur Implementierung einer Neuentwicklung. In diesem Rahmen lassen sich auch die vorgesehenen Datenbank-Gateways und deren abzubildenden Anforderungen aufbauen und weiterentwickeln. Im Rahmen dieser Strategie ist durch die schrittweise Neuentwicklung und Ablösung des Altsystems die Interaktion von Alt- und Zielsystem notwendig. Aus diesem Grund ist der Zustand des Altsystems ausschlaggebend für die erfolgreiche Durchführung der Strategie, so dass gegebenenfalls vor dem Beginn der Entwicklung eine (Teil-)Sanierung des Altsystems notwendig sein könnte.

Bei Anwendung der Butterfly-Strategie kann die iterative und inkrementelle Datenmigration mit einem passenden Entwicklungsprozess kombiniert werden, so dass die Ablösung des Altsystems mit der letzten Datenüberführung abgeschlossen werden kann und der entwickelte Data Access Allocator zeitgleich abgeschaltet werden kann. Eine Kommunikation zwischen Alt- und Zielsystem ist beim Butterfly nicht vorgesehen, so dass die Neuentwicklung unabhängig vom Altsystem stattfinden kann und der Zustand des Altsystems nicht ausschlaggebend für die Modernisierung ist.

4.4.3 Migration

Bei einer Migration handelt sich in der Regel ebenfalls um eine komplette Neuentwicklung unter Einbeziehung des Altsystems. Ziel ist es, möglichst viele Teile des Bestandssystems soweit es geht einzubeziehen, zu übernehmen oder einfach ins Zielsystem zu übertragen. Zur Durchführung einer Migration eignen sich zwei Strategien.

Aufgrund der Freiheit in Bezug auf die Softwaremigration kann die Datenmigration mit Hilfe der Butterfly-Strategie durchgeführt werden. Obwohl bei dieser Strategie keine Interaktion von Alt-und

Zielsystem vorgesehen ist, kann man im Rahmen einer flexiblen Erweiterung, ähnlich wie im Fall der Neuentwicklung, ein iteratives und inkrementelles Entwicklungsvorgehen anwenden, um die Migration zu realisieren, um so die komplette oder in Teilen geplante Ablösung des Bestandssystems gemeinsam mit der letzten Datenübertragung des Data Access Allocators durchzuführen.

Die Chicken Little-Strategie eignet sich durch die eingeplante Kommunikation zwischen Alt- und Zielsystem besonders gut für eine iterativ und inkrementell durchgeführte Migration. Dadurch wird der Zustand des Bestandssystems jedoch relevant und es müssen gegebenenfalls (Teil-)Sanierungen vorgenommen werden, bevor die ersten Teilmigrationen vorgenommen werden können. Durch die Freiheit dieser Strategie in Bezug auf die Datenmigration bietet sich zur Vermeidung von Betriebsausfällen eine allgemeine Datenmigration an, die eine iterative Systemmigration ebenfalls begünstigt. Durch den gewählten Entwicklungsprozess bleibt das Risiko der zusätzlichen Entwicklungsaufwände ebenfalls kontrollierbar.

Aufgrund der nicht vorgesehenen Interaktion zwischen Alt- und Neusystem und den schrittweise vorgesehenen Einzelmigrationen eignet sich die Big Bang-Strategie für eine Migration tendenziell nicht.

	Standardsoftware	**Neuentwicklung**	**Migration**
Big Bang	+	+	–
Chicken Little	–	++	++
Butterfly	++	++	+

Tabelle 4.1: Validierung von Strategien und Optionen

5 Bewertung der Modernisierungsszenarien

Die Entscheidung, ob und wie eine Modernisierung durchgeführt wird, ist eine Entscheidung, die auf Basis von verschiedenartigen Fragestellungen getroffen wird. In dieser Ausarbeitung wird auf Basis der technischen Entscheidungskriterien eine qualitative Bewertung vorgenommen. Diese technischen Kriterien haben Auswirkungen auf organisatorische und wirtschaftliche Faktoren, wie beispielsweise Kosten, Zeit und Risiko(management). Diese werden häufig erwähnt, weil sie im Umgang mit Bestandssystemen und deren Ablösung untrennbar miteinander verbunden sind, jedoch nur bedingt in die Bewertung einfließen. Die Entscheidungsfragen für Wirtschaftlichkeit und Organisation und auch eine quantitative Bewertung sind nicht Teil dieser Arbeit.

Das Kapitel ist anhand der jeweiligen Ausgangslage, also dem Zustand des zu modernisierenden Bestandssystems, strukturiert (s. 2.2.1, 2.2.2, 2.2.3). Für jeden Zustand werden exemplarisch drei der relevantesten Architektur-Muster im Unternehmenskontext als angestrebtes Ziel der Modernisierung angenommen (s. 2.1.3, 2.1.4, 2.1.5) und dann in Zusammenhang mit den bereits zuvor in Kontext gesetzten Modernisierungsstrategien (s. 4.1, 4.2, 4.3) und Ablösungsoptionen (s. 3.2, 3.3) gebracht. Da die Umstellung auf eine *Standardsoftware* aufgrund der meist unbekannten und auch nicht beeinflussbaren Architektur keiner Zielarchitekur zugeordnet werden kann, werden diese als separates Ziel betrachtet.

5.1 Modernisierung strukturierter Bestandssysteme

Strukturierte Bestandssysteme haben den großen Vorteil der bereits bestehenden Modularität seiner Komponenten und somit auch ihrer Schnittstellen und Datenbankservices. Wenn die Struktur eines Bestandssystems modular ist und die Komponenten eine gute Qualität aufweisen, stellt sich die Frage, ob eine Modernisierungsmaßnahme überhaupt notwendig ist, wenn es nicht um spezifischere Themen geht. Dazu gehört die Ablösung eines Mainframe, eine breit gefächerte Architekturvereinheitlichung oder der Versuch der Nutzung spezifischer technologischer Vorteile.

5.1.1 Standardsoftware

Bei der Ablösung eines Bestandssystems durch eine Standardsoftware ist der Zustand des abzulösenden Systems nicht relevant, genauso wenig wie die Erreichung eines konkreten Architekturmusters. Das Ziel ist es ein Kaufsystem mit einer möglichst geringen Differenz zwischen den abzubildenden Prozessen und den Anforderungen für die Ablösung zu finden. Wie das eingekaufte

System aufgebaut ist, kann weder eingesehen, noch beeinflusst werden. Trotzdem bieten strukturierte Systeme Vorteile im Falle einer Datenbankmigration, um diese auf den Gebrauch mit der Standardsoftware abzustimmen.

Die erste Option ist die Ablösung mittels der *Big Bang*-Strategie, also eine *Database Last*-Migration mit dem verbundenen Systemausfall während der Datenübertragung. Im Zusammenhang mit den strukturierten Systemen kann es sinnvoll sein, einen Teil der bestehenden Datenbankservices wiederzuverwerten und anstatt der Datenbankmigration ein Gateway für die Kommunikation zwischen Datenbank und Standardsoftware zu ziehen, was die klassischen Implementierungsrisiken mit sich bringt. Besser würde sich die *Butterfly*-Methode mit ihrem iterativen und inkrementellen Datenmigrationsverfahren eignen. Teile des strukturierten Systems können für die Entwicklung des *Data Access Allocators* verwendet werden, um so einen Betriebsausfall zu vermeiden und die Standardsoftware direkt nach der Entwicklung produktiv einsetzen zu können.

5.1.2 Schichtenarchitekturen

Da es sich bei Schichtenarchitekturen um strukturierte Systeme mit einem hierarchischen Aufbau handelt, lohnt sich vorab eine genaue Betrachtung der bisherigen Beschaffenheit des Bestandssystems. Aus ihr lassen sich Orientierungen für das Grobkonzept und die Hierarchien der Zielarchitektur ableiten. Abgestimmt auf die Modernisierungsmaßnahmen können sogar einzelne Komponenten des Altsystems wiederverwendet und entsprechend der technischen und auch fachlichen Schichtung in Modulen restrukturiert und integriert werden. Dies hängt von der Qualität der einzelnen Komponenten ab. Vorteilhaft sind im Zusammenhang mit strukturierten Systemen die separierten System- und Nutzerschnittstellen, sowie die Abgeschlossenheit von Anwendungslogik und Datenbankservices.

Wenn ein strukturiertes System vorliegt, ist die Frage, ob es ein sinnvolles Vorgehen ist, *auf der grünen Wiese* eine komplette Neuentwicklung zu beginnen. Das Wissen um bisherige Prozesse wird zwar verwertet, die Entwicklung findet jedoch vollkommen unabhängig von dem Zustand des Altsystems statt. Wenn dieses jedoch in einem guten Zustand ist und die Ablösung aus einem anderen Grund stattfindet, lohnt es sich, dies entsprechend zum Vorteil des Modernisierungsvorhabens zu nutzen. Wird sich für eine Neuentwicklung entschieden, können alle drei Migrationsstrategien in Betracht gezogen werden.

Die *Big Bang*-Strategie hat im Zusammenhang mit strukturierten Systemen und Schichtenarchitekturen den Vorteil der Hilfestellung durch das Altsystem bei der Entwicklung des Datenbankgateways. Dieses kann unter Umständen sogar als Teil der Schichtenarchitektur in diese integriert werden. Während der Neuentwicklung müssen Alt- und Zielsystem parallel betrieben und sich ändernde Anforderungen in beiden Systemen umgesetzt werden. Es bleiben die bekannten Nachteile des Risikos bei der Big Bang-Umstellung und der Betriebsstillstand bei dem Übertragen der Daten in die neuen Datenbankschemata bestehen. Dieser Nachteil kann durch erhöhte Entwicklungsaufwänden bei Nutzung der *allgemeinen Datenbankmigration* ausgeglichen werden.

Bei der *Chicken Little*-Strategie wird das Risiko durch viele inkrementelle und iterative Entwicklungsschritte kontrollierbar. Es ist jedoch ein zeitaufwändigeres Vorgehen. In diesem Szenario muss ebenso der Parallelbetrieb von Alt- und Zielsystem so lange gewährleistet werden, bis das Zielsystem die benötigten Funktionen des Altsystems vollständig abdeckt. Die Struktur des Bestandssystems hilft im Rahmen dieses Vorgehens insofern weiter, dass die Module als Leitfaden für die Reihenfolge der Neuentwicklung dienen können. Außerdem unterstützt sie die Entwicklung der notwendigen Datenbankgateways, unabhängig davon, welche Datenmigrationsmethode gewählt wird.

Wenn die Datenmigration als Hauptziel des Modernisierungsvorhabens festgelegt ist, kann im Zusammenhang mit der *Butterfly*-Strategie eine passende Softwareentwicklungsmethode gewählt werden, die den risikosenkenden, inkrementellen und iterativen Ansatz der Methode positiv unterstützt. Durch ihren Einsatz wird ein Betriebsausfall komplett durch zusätzliche Entwicklungsaufwände verhindert.

Wenn die Entscheidung zugunsten einer Migration gefällt wird, bietet sich die *Chicken Little*-Strategie besonders an. Durch die beabsichtigte Kommunikation zwischen Alt- und Zielsystem können die Elemente des strukturierten Systems schrittweise ersetzt werden. Ihre Restrukturierung in Schichten sollte bei einer angemessenen Projektplanung, aufgrund der vorliegenden Altsystem-Architektur, kein Hindernis darstellen. Mit der Ablösung des letzten Elements des Altsystems ist dieses vollständig migriert und somit ebenfalls eine komplette Neuentwicklung. Passend zu diesem Vorgehen kann eine Datenbankmigrationsstrategie gewählt werden. Alle zusätzlichen notwendigen Entwicklungen werden durch das strukturierte Bestandssystem unterstützt.

Möglich ist in diesem Zusammenhang auch eine Kombination aus *Butterfly* und *Chicken Little*-Strategie, um die Vorteile des inkrementellen und iterativen Ansatz konsistent für das gesamte Migrationsvorhaben zu nutzen. Die entstehenden Entwicklungsaufwände für den *Data Access Allocator* und das Vorhaben sind insgesamt zeitaufwändiger. Dabei bleiben die Risiken durchgehend kontrollierbar und Betriebsausfälle können komplett vermieden werden.

5.1.3 Microservices

Da es sich bei Microservices um ein strukturiertes Architekturschema handelt, ist eine vorgelagerte Betrachtung des strukturierten Bestandssystems eine Basis für Hilfestellungen bei dem Entwurf der Microservices-Architektur. Die vorgegebenen Strukturen dürfen jedoch nicht pauschal mit denen der Zielarchitektur gleichgesetzt werden. Da die Anforderungen von Microservices an die Modularisierung über einen rein technischen Aufbau hinausgehen, kann angenommen werden, dass ausgehend von der Bestandssystemstruktur zusätzliche Aufwände zur Restrukturierung und Umsetzung der bisherigen Prozesse zu Microservices nötig sind. Die separierten Schnittstellen des Altsystems können den Aufbau der Microservices-Infrastruktur unterstützen, genauso wie deren Kommunikation mit ihrer Umgebung. Eine Datenbankmigraton ist für das Zusammenwirken des Datenbestandes mit dem Microservices-Konzept aller Voraussicht nach notwendig. Für das

gewählte Datenbankkonzept müssen zusätzliche Pläne zur Umsetzung in die entwickelte Infrastruktur gemacht werden, da die Kommunikation nicht mehr auf einen Datenbankservice mit einer zugehörigen Datenbank reduziert werden kann. Eigene Datenbanken für die jeweiligen Services, die zu einem späteren Zeitpunkt in einer zentralen Datenbank zusammengeführt werden, sind keine Seltenheit. Wenn sich aus organisatorischer Sicht für eine Neuentwicklung entschieden wird, ist die Wahl der Migrationsstrategie wesentlich.

Bei Betrachtung der Ablösung mit der *Big Bang*-Strategie stellt sich die Frage nach der Zweckmäßigkeit dieser Methode. Wenn bei kompletter Restrukturierung und Modularisierung der bisherigen Prozesse im gleichen Entwicklungsschritt bereits das Datenbankenkonzept und auch die komplette Infrastruktur aufgesetzt werden muss, erscheint der Einsatz dieser Methode nur realistisch, wenn dies schrittweise pro Prozessbaustein inkrementell aufgebaut werden kann. Um einen Big Bang überhaupt zu ermöglichen, müsste Abstand von den beabsichtigen Datenbankmigrationsprozessen genommen werden.

Daher erscheint die *Chicken Little*-Strategie mit ihrem iterativen und inkrementellen Vorgehen besser für die Entwicklung dieser Zielarchitektur geeignet. Trotzdem kann im Kontext einer Neutenwicklung, die keine Kommunikation zwischen Alt- und Zielsystem vorsieht, der größte Vorteil der Ablösung von Bestandssystemen mit Microservices, nämlich die schrittweise Erweiterung des Systems bis zur vollständigen Umstellung, nicht genutzt werden. Allerdings taucht auch hier die Problemstellung auf, dass sich ändernde Anforderungen bis zur Ablösung zeitgleich in beiden System umgesetzt werden müssen. Zusätzlich kann auch die Chicken Little-Strategie mit den angedachten Datenbankmigrationen keine Modernisierung mit einem beherrschbaren Risiko durchführen.

Somit ist die *Butterfly*-Strategie für die Datenmigration in Kombination mit der Chicken Little oder Big Bang-Strategie das einzige praxistaugliche Vorgehen, wobei die iterative und inkrementelle Methode der Butterfly und der Chicken Little-Strategie nahtloser ineinander greifen. Der zu entwickelnde *Data Access Allocator* kann unter Umständen auch als Basis für die kontinuierliche Migration von Daten, nach der eigentlichen Ablösung, in der Microservices-Infrastruktur dienen. Durch diese notwendige Entwicklung des umfangreichen Data Access Allocators und aufgrund des inkrementellen und iterativen Systementwicklungsprozesses ist dieses Vorgehen sehr langwierig. Dabei bleibt jedoch das Risiko bei den einzelnen Schritten kontrollier- und überwachbar.

Im Falle einer Migration kann die *Chicken Little*-Strategie durch die schrittweise Erweiterung den Microservices-Vorteil der kontinuierlichen Altsystemablösung nutzen. Die Struktur des Bestandssystems und seine klar definierten Schnittstellen unterstützen dieses Vorgehen maßgeblich. Es muss kein Parallelbetrieb und auch keine Parallelentwicklung praktiziert werden. Allerdings sind auch in diesem Fall die vorgesehenen Datenmigrations-Strategien nicht anwendbar.

Daher bietet es sich auch in diesem Fall an, die bereits beschriebene Kombination der Chicken Little und der *Butterfly*-Strategie, unter Nutzung aller Vorteile der inkrementellen und iterativen Entwicklung und Migration, zu verwenden. Dies hat den Vorteil, dass der Nutzen für die Organisation und die Anwender stetig nach Implementierung der grundlegenden Infrastruktur mit der Ablösung jedes Prozesses steigt und das Risiko beherrschbar bleibt. Es handelt sich zwar um eine

verhältnismäßig zeit- und somit kostenintensive Migrationsmethode, die aber letztendlich eine flexible und nachhaltige Microservices-Architektur schaffen kann.

5.1.4 Monolithen

In dieser Ausarbeitung bezieht sich der Begriff des Monolithen auf sogenannte Deployment-Monolithen, die intern eine modulare Struktur aufweisen, jedoch nur als Einheit ausgeliefert werden können. Ausgehend von einem strukturierten Bestandssystem kann dessen Betrachtung eine Orientierung zur internen Modularisierung der Zielarchitektur sein und bei der Entwicklung von Elementen, die eine Ablösung unterstützen können, helfen. Die separierten Systemschnittstellen können das Grundgerüst für die Kommunikation des Monolithen mit seiner Umgebung bilden. Im Rahmen der Modernisierungsmaßnahme kann eine Wiederverwendung einzelner Komponenten in Betracht gezogen werden, wenn sie eine entsprechende Qualität aufweisen. Die Art der Modularisierung bei dieser Zielarchitektur hat keine genauen Vorgaben. Der primäre Risikofaktor ist der *Single Point of Failure* des Deployment der Zielanwendung. Wenn die organisatorische Entscheidung für eine Neuentwicklung gefällt wird, lassen sich grundsätzlich alle Migrationsstrategien anwenden.

Fraglich ist im Zusammenhang mit der *Big Bang*-Strategie, ob die Kombination des Big Bang-Umstellungsrisikos und des *Single Point of Failure* eines Deployment-Monolithen tragbar ist. Eine Reduzierung des Risikos könnte mit einer möglichst produktionsnahen Test- und Integrationsumgebung erwirkt werden. Darüber hinaus muss der interaktionslose Parallelbetrieb und die Umsetzung sich ändernder Anforderungen in Alt- und Zielsystem beachtet werden. Der Betriebsausfall, den die *Database Last*-Migration bewirken würde, kann mit zusätzlichen Entwicklungsaufwänden durch eine *allgemeinen Datenmigration* vermieden werden.

Mit der *Chicken Little*-Strategie kann das Umstellungsrisiko durch das inkrementelle und iterative Vorgehen besser kontrolliert werden. Die Elemente des Monolithen können schrittweise entwickelt werden. Im Fehlerfall schlägt die Anwendung zwar im Ganzen fehl, die Fehlerquelle kann im Kontext jedoch besser eingegrenzt werden. Während der Neuentwicklung muss der interaktionslose und parallele Betrieb von Alt- und Zielanwendung sichergestellt werden. Dies gilt auch für die Entwicklung sich ändernder Anforderungen in beiden Systemen, um die Bedürfnisse der Organisation während des Modernisierungsvorhabens weiterhin stillen zu können. Zum Chicken Little-Vorgehen kann eine beliebige Datenmigrationsmethode anhand ihrer eigenen Vor- und Nachteile bewertet und gewählt werden.

Zur Datenmigration kann die *Butterfly*-Strategie angewendet werden, die durch zusätzliche Entwicklungsaufwände einen Betriebsausfall umgeht. Es kann ein beliebiges Systementwicklungsverfahren zu diesem Vorgehen ausgewählt werden.

Wenn die bestehende Strukturierung des Bestandssystem genutzt werden soll, bietet sich eine Migration und die damit verbundene vorgesehene Interaktion zwischen Bestands- und Zielsystem an. Die Nutzung der *Chicken Little*-Strategie ermöglicht einen schrittweisen Austausch der strukturierten Bestandteile des Altsystems durch interne modulare Elemente des Monolithen. Die definierten

Systemschnittstellen ermöglichen deren Interaktion. Passend zu diesem Vorgehen kann eine Datenmigration bewertet und ausgewählt werden, die das Modernisierungs- und Entwicklungsrisiko des Monolithen steuerbar hält.

Die *Butterfly*-Strategie kann als Ergänzung zum iterativen und inkrementellen Vorgehen der Chicken Little-Methode oder einem anderen passenden Entwicklungsvorgehen zur Datenmigration genutzt werden, um einen Betriebsausfall zu vermeiden.

5.2 Modernisierung semi-strukturierter Systeme

Der Aufbau von semi-strukturierten Systemen kann die nahezu ideale Ausgangslage der strukturierten Systeme nicht erreichen, da Datebankservices und Anwendungslogik nicht separiert sind. Die Schnittstellen zu anderen Systemenï sowie die die Nutzerschnittstellen sind isoliert.

5.2.1 Standardsoftware

Da semi-strukturierte Systeme über keine separierte Datenbankschnittstelle verfügen, können sie die Implementierung zusätzlicher Strukturen für die Ablösungsstrategien nur bedingt unterstützen. Trotzdem gilt auch in diesem Fall, dass der Zustand des Bestandssystems, bei einer Ablösung durch eine Standardsoftware, nicht relevant ist.

Zur Ablösung kann die *Big Bang*-Strategie unter Anwendung der *Database Last*-Migration verwendet werden. Diese geht mit dem bereits bekannten Ausfall des produktiven Systems einher. Den Sonderfall der Entwicklung eines dauerhaften Datenbankgateways ist erst nach einer aufwändigen Sanierung des Bestandssystems und dem Herauslösen der Datenbankschnittstelle möglich. Dies würde den Betriebsausfall vermeiden, aber nicht unerhebliche zusätzliche Implementierungsaufwände verursachen.

Bei der Ablösung von semi-strukturierten Systemen eignet sich ebenfalls die *Butterfly*-Strategie durch ihre Fokussierung auf die Datenmigration am besten. Die Entwicklung des *Data Access Allocators* wird durch die Struktur des Systems jedoch nicht positiv beeinflusst. Nach dessen Entwicklung kann das Bestandssystem direkt abgelöst werden und die Kaufsoftware produktiv eingesetzt werden. Der Nutzen wird für die Organisation und Anwender sofort sichtbar.

5.2.2 Schichtenarchitekturen

Auch bei semi-struktuierten Bestandssystemen kann eine Vorbetrachtung der modularen Elemente helfen, eine Orientierung über die angezielte Schichtenarchitektur zu erlangen und die Restrukturierung und Hierarchisierung zu erleichtern. Abhängig von ihrer Qualität können die Kommunikationsmechanismen der Systemschnittstellen wiederverwendet werden. Die Wiederverwendung der Nutzerschnittstellen beruht auf deren Anwendung aktueller Sehgewohnheiten der Anwender (s. 3.3.1). Die nicht separierbaren Strukturen der Anwendungslogik und Datenbankservices

müssen einer technischen Prozessanalyse unterzogen und abhängig von der gewählten Modernisierungsmethode auch bis zu einem gewissen Grad saniert werden, um einen möglichst hohen Anteil der Strukturen des Bestandssystems für die Ablösung gewinnbringend nutzen zu können. Ob eine Neuentwicklung oder eine Migration die geeignetere Ablösungsmethode ist, wird durch das Verhältnis der strukturierten und unstrukturierten Anteile beeinflusst. Im Falle einer Neuentwicklung ist grundsätzliche die Anwendung jeder Strategie möglich.

Bei der *Big Bang*-Strategie kann aufgrund der nicht idealen Modularisierung des Bestandssystems, ohne Sanierungsarbeiten, keine Unterstützung für die Entwicklung des Datenbankgateways gewonnen werden. Unabhängig davon kann ein nachhaltig geschriebenes Gateway in die angestrebte Schichtenarchitektur integriert werden. Während der Neuentwicklung findet keine Kommunikation zwischen Alt- und Zielsystem statt. Die Systeme müssen während des gesamten Modernisierungsprozesses parallel betrieben und sich ändernde Anforderungen in beiden Systemen umgesetzt werden. Der Nachteil des hohen Risikos bei der Big Bang-Umstellung bleibt bestehen. Der Betriebsausfall während der Datenübertragung unter Anwendung der *Database Last*-Methode kann durch erhöhte Entwicklungsaufwände mithilfe der *allgemeinen Datenmigration* vermieden werden.

Mit der *Chicken Little*-Strategie wird das Systementwicklungsrisiko durch das inkrementelle und iterative Vorgehen gesenkt. Durch den kommunikationslosen Parallelbetrieb von Bestands- und Zielsystem müssen sich zwischenzeitlich ändernde Anforderungen analog in beiden Systemen umgesetzt werden. Die strukturierten Bestandteile des Altsystems können die schrittweise Entwicklung begünstigen. Dies gilt jedoch nicht für die Gatewayentwicklung, aufgrund der verschmolzenen Anwendungslogik und Datenbankservices des semi-strukturierten Systems. Somit kann die Wahl der Datenmigrationsstrategie unabhängig getroffen werden, da ihre zusätzlichen Entwicklungsaufwände nicht positiv beeinflusst werden können. Sie muss jedoch passend zur iterativen und inkrementellen Systementwicklung gewählt werden.

Wenn im Rahmen der Neuentwicklung die Datenmigration das primäre Ziel ist, kann die *Butterfly*-Strategie angewendet werden. Entsprechend ihres inkrementellen und iterativen Vorgehens sollte ein kompatibles Softwareentwicklungsvorgehen gewählt werden, um die Schichtenarchitektur umzuseten. Bei der Entwicklung des *Data Access Allocators* kann mit keiner Unterstützung durch das Bestandssystem gerechnet werden. Durch die erhöhten Entwicklungsaufwände kann mit dieser Strategie ein Betriebsausfall, aufgrund einer Datenübertragung im Ganzen, vermieden werden.

Wenn eine Entscheidung zugunsten einer Migration getroffen wird, kann diese mit Hilfe der *Chicken Little* oder der *Butterfly*-Strategie erarbeitet werden. Die beiden Strategien können auch kombiniert werden. Bei einer Migration und der vorgesehenen Kommunikation zwischen Alt- und Zielsystem kann Chicken Little sein Potenzial des iterativen und inkrementellen Ansatzes bei den modularen Strukturen des Bestandssystems vollständig ausschöpfen. Diese können schrittweise abgelöst und sich ändernde Anforderungen müssen nicht parallel umgesetzt werden. Bei der Ablösung des unstrukturierten Anteils des Systems fallen jedoch zusätzliche Aufwände bei der Aufarbeitung der verwobenen Prozesse und Funktionalitäten an, bevor diese im Zielsystem implementiert werden können. Die Anwendungslogik und Datenbankservices können erst vollständig abgelöst werden,

wenn sie komplett in der Zielarchitektur umgesetzt wurden. Gemeinsam mit dem Umfang der Einzelmigrationen steigt auch deren Risiko. Letztendlich ist die Ablösung durch den Einsatz der Chicken Little-Strategie ebenfalls eine komplette Neuentwicklung. Die Wahl der Datenmigrationsmethode ist abhängig vom Risikomanagement der Organisation.

Passend zu dieser Strategie kann die *Butterfly*-Methode ohne Einschränkungen durch die semi-optimale Struktur des Bestandssystems für eine iterative und inkrementelle Datenmigration verwendet werden, die einen Betriebsausfall verhindert und das Risiko beherrschbar macht. Eine Unterstützung bei der Entwicklung des *Data Access Allocators* kann nicht erzielt werden.

5.2.3 Microservices

Das semi-strukturierte System ist eine weniger ideale Ausgangslage für ein Modernisierungsvorhaben. Da jedoch bei der Restrukturierung der Prozesse zu Microservices keine einfache Transformation vorgenommen werden kann und das gesamte Vorhaben zeitintensiv ist, da die Zielarchitektur eine hohe Grundkomplexität aufweist, hat die Struktur des Bestandssystems keinen weiteren maßgeblichen negativen Einfluss. Bei diesem System sind die Systemschnittstellen separiert und können für den Aufbau der Infrastruktur und der Kommunikation mit der Umgebung als Unterstützung genutzt werden. Die Aufgabenstellung der Datenmigration zugunsten der Microservices-Architektur ist unabhängig von jeglicher Altsystemstrukturierung. Im Rahmen einer angestrebten Neuentwicklung muss auch bei semi-strukturierten Systemen eine genaue Vorabbewertung vorgenommen werden, ob durch dieses Vorgehen entscheidende Vorteile, die den gesamten Modernisierungsprozess positiv beeinflussen können, genutzt werden können. In diesem Fall muss das Verhältnis der modularen und somit einfacher abzulösenden Elemente und der verschmolzenen Gebilde, die zur schrittweisen Migration zunächst eine Sanierung benötigen, beurteilt werden.

Im Falle der Entscheidung zugunsten einer Neuentwicklung muss für die Anwendung der *Big Bang*-Strategie ein den Anforderungen gerecht werdendes Datenmigrationsverfahren ausgewählt werden. Während der Entwicklung muss der parallele Betrieb der beiden Systeme und die analogen Umsetzung sich ändernder Anforderungen eingeplant werden. Das Risiko der Big Bang-Umstellung bleibt bestehen, der Betriebsstillstand während der Datenmigration kann jedoch vermieden werden.

Die *Chicken Little*-Strategie eignet sich aufgrund ihres risikobeherrschenden, inkrementellen und iterativen Ansatzes tendenziell besser für eine Ablösung durch eine Neuentwicklung. Dies gilt jedoch, wie beim Big Bang, nur mit einer angepassten Datemigration, durch die der Betriebsausfall des Altsystems während der Ablösung vermieden werden kann. Bei dieser Strategie muss ebenfalls der Parallelbetrieb sichergestellt und sich ändernde Anforderungen in beiden System umgesetzt werden.

Die *Butterfly*-Methode wird den Anforderungen der Datenmigration im Zusammenhang mit Microservices aufgrund des inkrementellen und iterativen Vorgehens gerecht. Durch sie wird ein Betriebsausfall während einer ganzheitlichen Datenübertragung verhindert. Dadurch eignet sie sich zur Kombination mit der Big Bang oder der Chicken Little-Strategie. Für die Entwicklung des *Data*

Access Allocators kann eine Unterstützung aus den bestehenden Datenbankservices abgeleitet werden, da diese nicht von der Anwendungslogik trennbar sind.

Wenn die Entscheidung zugunsten einer Migration getroffen wird, kann die *Chicken Little*-Strategie zur schrittweisen Ablösung des semi-strukturierten Bestandssystems verwendet werden. Dadurch muss kein Parallelbetrieb und auch keine Umsetzung von Anforderungen in Alt- und Zielsystem beachtet werden. Kritisch kann die Ablösung der nicht separierbaren Strukturen werden, da mit dem Umfang der Einzelmigrationen auch deren Risiko steigt, wenn nicht zuvor Zeit in eine Sanierung der Strukturen investiert wurde. Trotzdem können die Ablösungsvorteile von Microservices für Bestandssysteme bis zu einem gewissen Grad umfänglich genutzt werden. In diesem Szenario werden ebenfalls die vorgesehenen Datenbankmigrationen den Anforderungen nicht gerecht.

Somit bietet sich die Kombination der *Butterfly*-Strategie mit dem iterativen, inkrementellen Vorgehen der Chicken Little-Strategie an, um so eine kontinuierliche Steigerung des Nutzens der Migration für die Organisation und die Anwender schon während der Modernisierungsmaßnahme zu erwirken, das Risiko kontrollierbar zu halten und eine nachhaltige Microservices-Architektur und Infrastruktur zu entwickeln.

5.2.4 Monolithen

Wenn die Ausgangslage ein semi-strukturiertes Bestandssystem ist, kann die Betrachtung der bestehenden Struktur Hilfestellungen in Bezug auf den Entwurf der Zielarchitektur und Zusatzentwicklungen bieten. Eine Wiederverwendung von einzelnen Komponenten kann möglich sein.

Bei einer Neuentwicklung mit der *Big Bang*-Strategie stellt sich unabhängig von der Ausgangstruktur des Bestandssystems ebenfalls die Frage, ob das Risiko des *Single Point of Failure* in Kombination mit einer Big Bang-Umstellung zumutbar ist. Außerdem muss der interaktionslose Parallelbetrieb und die Umsetzung sich ändernder Anforderungen während der Moderniserungsmaßnahme bedacht werden. Der Betriebsausfall im Rahmen einer Datenmigration mit der *Database Last*-Methode kann durch die zusätzliche Entwicklungsaufwände bei Anwendung der *allgemeinen Datenmigration* kompensiert werden.

Wenn die *Chicken Little*-Strategie angewendet wird, kann durch die inkrementelle und iterative Entwicklung ein Fehler bei der Auslieferung besser eingegrenzt werden. Dadurch wird das allgemeine Risiko etwas gemindert. Zudem passt das Entwicklungsverfahren besser als die Big Bang-Strategie zu der vorgesehenen inneren modularen Struktur des Monolithen. Dazu passend kann an den eigenen Vorteilen und Risiken eine Datenmigrationsmethode gewählt werden.

Dies kann beispielsweise auch die *Butterfly*-Strategie sein, deren ebenfalls inkrementelles und iteratives Vorgehen gut mit der Chicken Little-Strategie kombiniert werden kann, um einen Systemausfall zu vermeiden und das Datenmigrationsrisiko zu senken. Diese Rückversicherung wird sich ebenfalls durch zusätzliche Entwicklungsaufwände des *Data Access Allocators* erworben, die durch das semi-strukturierte Bestandssystem nicht oder nur bedingt begünstigt werden.

Vor einer Migration mit der *Chicken Little*-Strategie muss bewertet werden, inwiefern eine Interaktion der Komponenten von Alt- und Zielsystem das Vorhaben begünstigt. Muss eine Vielzahl von Sanierungen durchgeführt werden, um eine Kommunikation zwischen den Systemen zu ermöglichen, kann eine Neuentwicklung, auch in Abhängigkeit zur angewendeten Modernisierungsstrategie, die bessere Wahl sein. Im Falle einer Entscheidung zugunsten der Migration können die Stärken der Chicken Little-Strategie bei der Interaktion zwischen Bestands- und Zielsystem genutzt werden. Sich ändernde Anforderungen müssen nur einmalig umgesetzt werden und das durch den Deployment-Monolithen weiterhin bestehende *Single Point of Failure*-Risiko kann durch das inkrementelle und iterative Entwicklunsgvorgehen besser beherrscht werden. Dazu kann eine beliebige Datenmigrationsmethode gewählt werden.

Im iterativen und inkrementellen Fall bietet sich häufig die Kombination mit der *Butterfly*-Strategie an. So kann das schrittweise Vorgehen von der Systementwicklung bis zur Datenmigration kontinuierlich angewendet werden. Somit bleibt das Modernisierungsvorhaben überwach- und steuerbar und Betiebsausfälle können durch zusätzliche Entwicklungsaufwände eines *Data Access Allocators* vermieden werden.

5.3 Modernisierung unstrukturierter Systeme

Unstrukturierte Systeme sind die schlechteste Ausgangslage für eine Ablösung, wenn der Zustand des Bestandssystems für diese relevant ist. Da das System eine Blackbox ist, können auch die Schnittstellen nicht separat betrachtet werden.

5.3.1 Standardsoftware

Da bei der Ablösung durch eine Standardsoftware der Zustand des Ausgangssystems nicht relevant ist, bietet sich diese Modernisierungsmöglichkeit bei unstrukturierten Systemen an. Im Falle von Zusatzentwicklungen, die eine Ablösung unterstützen sollen, können jedoch keine oder nur aufwändig sanierte Hilfestellungen aus dem bestehen Quellcode gezogen werden.

Bei einer Ablösung im Rahmen der klassischen *Big Bang*-Strategie mit einer *Database Last*-Migration kommt der Nachteil des Stillstand des Betriebs während der Übertragung der Daten zum Tragen. Die Entwicklung eines Gateways würde allerdings einen hohen Sanierungsaufwand erfordern, um die Datenbankschnittstelle aus der Altsystem-Blackbox separieren zu können. Diese Aufwände müssen verglichen werden, um bei Anwendung dieser Strategie die organisatorisch beste Entscheidung für das Vorgehen treffen zu können.

Die technisch bessere Entscheidung wäre auch hier die *Butterfly*-Strategie mit ihrem Hauptaugenmerk auf die Datenbankmigration, obwohl auch in diesem Fall keine oder sehr wenig Unterstützung für die Entwicklung des notwendigen *Data Access Allocatos* aus dem Bestandscode gezogen werden kann. Diese Strategie vermeidet jedoch den Ausfall des produktiven Systems und die

Organisation kann direkt Nutzen aus dem Kauf der Standardsoftware nach dem Einsatz des Data Access Allocators ziehen.

5.3.2 Schichtenarchitekturen

Im Fall eines unstrukturierten Bestandssystems kann die Betrachtung und Analyse nicht oder nur bedingt bei der Entwicklung der Zielarchitektur und auch bei Zusatzentwicklungen im Rahmen der Ablösung helfen. Durch den Zusammenschluss aller Schnittstellen, der Anwendungslogik und der Datenbankservices ist anzunehmen, dass von diesen Strukturen nichts wiederverwertet werden kann und sollte, ohne vorab umfassende und zeitintensive Sanierungen vorzunehmen. Diese können zumindest für die Systemschnittstellen lohnenswert sein, um das Zielsystem besser in die Anwendungslandschaft der Organisation integrieren zu können. Potenziell kann die Ablösung eines solchen Systems Chancen bieten, Änderungen und Verbesserungen für die Infrastruktur und auch andere Systeme zu erwirken. Die Vorgabe, dass das angestrebte Zielsystem eine Schichtenarchitektur aufweisen soll, beeinflusst den Ablauf einer Ablösung eines unstrukturierten Systems nicht, da dessen Entwurf *auf der grünen Wiese* stattfindet. Bezogen auf das einzelne unstrukturierte System erscheinen Strategien für eine interaktionslose Neuentwicklung am erfolgsversprechenden.

Im Falle der Anwendung der *Big Bang*-Strategie können die bekannten Nachteile nicht ausgeglichen werden. Bis zur risikoreichen Big Bang-Übernahme müssen Alt- und Zielsystem gleichzeitig weiter betrieben und sich ändernde Anforderungen parallel umgesetzt werden. Der Betriebsausfall der angedachten *Database Last*-Migration kann unter Anwendung einer *allgemeinen Datenmigration* verhindert werden. Bei der Entwicklung, der in diesem Rahmen benötigten Gateways, des *Coordinators* und der *Mapping Table*, kann es durch die mangelnde Struktur und das Verständnis der bisherigen Datenbankservices zu erhöhten Aufwänden kommen.

Chicken Little fördert das Risikomanagement des Neuentwicklungsvorhabens durch sein iteratives und inkrementelles Vorgehen. Aufgrund der Tatsache, dass wenige bis keine Informationen aus dem Altsystem zu gewinnen sind, sind die einzelnen Schritte ein reines Projektorganisationsthema. Der parallele Betrieb der beiden Systeme muss bis zuletzt sicher gestellt werden und sich ändernde Anforderungen analog umgesetzt werden. Die Datenmigration kann anhand der Bewertung ihrer eigenen Vor- und Nachteile gewählt werden.

Die *Butterfly*-Strategie und ihre reine Fokussierung auf die Datenmigration scheint im Zusammenhang mit einer passend gewählten Systementwicklungsstrategie für die Modernisierung am vielversprechendsten. Sie verhindert einen Betriebsausfall des Bestandssystems bis zur letzten Datenmigration, die im optimalen Fall zeitgleich mit der Inbetriebnahme des Zielsystems abgeschlossen wird.

Sollte sich die Organisation für eine Migration entscheiden, ist die Anwendung der *Chicken Little*-Strategie unter der Voraussetzung, dass wie vorgesehen Interaktionen zwischen Alt- und Neusystem stattfinden sollen, ein extrem umfangreiches Vorhaben. Bevor eine Kommunikation zwischen den Systemen etabliert werden kann, muss das Bestandssystem so saniert und vorbereitet werden, dass die Schnittstellen grundlegend separier- und erreichbar sind. Es stellt sich daher die Frage,

ob es wirtschaftlich, anhand der Risikobewertung und der zu erwartenden Aufwände überhaupt sinnvoll sein kann, dieses Vorgehen zu wählen.

Im Rahmen einer Migration würde somit nur die *Butterfly*-Strategie verbleiben, deren Verlauf identisch zum Ablauf bei einer Neuentwicklung wäre.

5.3.3 Microservices

Unstrukturierte Bestandssysteme bieten anhand ihres Aufbaus und ihrer Infrastruktur die wenigsten Möglichkeiten, bei dem Entwurf und der Entwicklung von Microservices, zu unterstützen. Es kann davon ausgegangen werden, dass bei einem solchen System der zuvor angewendete, iterative und inkrementelle Migrationsvorteil nicht oder nur nach außerordentlichen Sanierungsaufwänden genutzt werden kann. Daher geht die Tendenz einer vorteilhaften Entscheidung in dieser Konstellation in Richtung einer kompletten Neuentwicklung, um eine Interaktion und somit Sanierungsaufwände am Altsystem möglichst zu vermeiden.

Bei Anwendung der *Big Bang*-Strategie kann interaktionslos während des Parallelbetriebs von Alt- und Zielsystem entwickelt werden. Sich ändernde Anforderungen müssen während der Modernisierungsmaßnahme in beiden Systemen umgesetzt werden. Das Risiko der zentralen Systemumstellung kann nicht ausgeglichen werden und die komplexe Datenbankmigration kann nicht durch die klassischen Verfahren abgebildet werden. Es besteht somit die Notwendigkeit der Kombination mit der Butterfly-Strategie oder einer ähnlichen Methode, um das Risiko kontrollierbar zu halten und die Anforderungen abbilden zu können.

Eine bessere Risikokontrolle bietet die *Chicken Little*-Strategie mit ihrem iterativen und inkrementellen Vorgehen. Dabei muss die Gewährleistung des parallelen Betriebs und die Umsetzung sich ändernder Anforderungen während der Neuentwicklung beachtet werden. Mit dieser Strategie besteht ebenfalls die Notwendigkeit der Kombination mit der Butterfly-Strategie oder einer ähnlichen Methode.

In Kombination mit der Big Bang oder Chicken Little-Strategie übernimmt die *Butterfly*-Methode die Datenbankmigration mit Hilfe ihres inkrementellen und iterativen Ansatzes, um die Komplexität des Datenbankkonzepts, deren Kommunikation und Infrastruktur überwach- und kontrollierbar zu halten.

Die Entscheidung zu einer Migration ist aus technischer Sicht untragbar. Die *Chicken Little*-Strategie versucht eine schrittweise Migration durch Interaktion von Alt- und Zielsystem. Um das Bestandssystem jedoch so weit zu sanieren, damit eine Migration überhaupt praxistauglich durchführbar wäre, würde einen ähnlichen Aufwand bedeuten, wie eine komplette Neuentwicklung.

Auch die Kombination mit der *Butterfly*-Strategie macht ein Migrationsvorhaben, bei dem Interaktionen der Systeme nötig sind, nicht praktikabel.

5.3.4 Monolithen

Bei unstrukturierten Bestandssystemen kann davon ausgegangen werden, dass keine förderlichen Erkenntnisse aus der Betrachtung des Systems in Bezug auf die Zielarchitektur und methodenabhängige notwendige zusätzliche Entwicklungen der Modernisierungsmaßnahmen gewonnen werden können. Von einer möglichen Wiederverwendung kann ebenfalls nicht ausgegangen werden.

Eine Neuentwicklung kann, ausgehend von einem unstrukturierten Bestandssystem, eine bessere Lösung sein, da bei ihrer Umsetzung auf Interaktionen zwischen Alt- und Zielsystem verzichtet werden kann. Gegen die Anwendung der *Big Bang*-Strategie spricht jedoch weiterhin die risikoreiche Kombination des *Single Point of Failure* des Deployment-Monolithen und die Umstellung als Big Bang.

Eine bessere Risikokontrolle bietet die *Chicken Little*-Strategie, die durch das inkrementelle und iterative Vorgehen Rückschlüsse auf Fehlerquellen der Neuentwicklung zulässt, auch wenn die Auslieferung des Monolithen gänzlich fehl schlägt. Passend zum Vorgehen des Chicken Little kann eine entsprechende Datenmigrationsmethode gewählt werden.

Mit der *Butterfly*-Strategie kann eine iterative und inkrementelle Datenmigration ohne Betriebsausfälle realisiert werden. Aus dem unstrukturierte Bestandssystem kann jedoch keine Unterstützung für die Entwicklung des benötigten *Data Access Allocators* gezogen werden.

Im Falle eines angezielten Deployment-Monolithen ist, wie auch in den zuvor durchgespielten Szenarien, eine Migration, die aufgrund von Interaktion zwischen Bestands- und Zielsystem vorteilhaft ist, wenn das Bestandssystem unstrukturiert ist, kein praktikables Vorgehen. Die Sanierungsaufwände, die nötig wären, um eine Kommunikation zwischen den Systemen zu ermöglichen, können ähnlich dimensioniert sein, wie eine komplette Neuentwicklung. Daher ist die Durchführung einer Migration mit der *Chicken Little*-Strategie, auch in Kombination mit einer *Butterfly*-Datenmigration als nicht praxistauglich anzusehen.

5.4 Weitere Modernisierungsszenarien

5.4.1 Hybride Systeme

In der Praxis ist die Begegnung mit reinen strukturierten, semi-strukturierten und unstrukturierten Systemen, ebenso wie Mischformen dieser, keine Seltenheit. [Brodie and Stonebraker, 1993] fassen diese unter dem Begriff der hybriden Systeme zusammen.

Die optimalen Ablösungsstrategien für hybride Systeme sind Kombinationen aus den erläuterten Szenarien. Ausschlaggebend ist der Anteil der modularen und unmodularen Strukturen, sowie die angestrebten Zielarchitektur. Auch anteilige Migration und Neuentwicklung sind wahrscheinlich. Jedes reale Szenario benötigt eine detaillierte Voranalyse und Einzelfallbetrachung, um auf dieser Basis eine nachhaltige und umsetzbare Strategie wählen zu können.

5.4.2 Datenbank-Monolithen

Die im Rahmen der Systemarchitekturen erwähnten Datenbank-Monolithen sind in der Praxis ebenfalls häufiger vorkommende Strukturen. Anhand der bisher gewonnen Erkenntnisse gibt es mehrere Möglichkeiten für ihre Ablösung, beziehungsweise dafür, den Nutzen für die Organisation und die Anwender nachhaltig zu verbessern. Wenn die gesamte Struktur der Datenbank verändert werden soll, kann davon ausgegangen werden, dass mit der *Butterfly*-Methode und einer beliebigen Neuentwicklungsstrategie in diesem Zusammenhang die besten Ergebnisse erzielt werden können. Eine Alternative kann es sein, wenn die Anforderungen und Prozesse von dem Datenbank-Monolithen aktuell und in Zukunft vollständig abgebildet werden können, die angeschlossenen Benutzeroberflächen für die Datenabfragen und -eingaben durch Neuentwicklungen zu ersetzen. Eine weitere Möglichkeit ist der Einzug eines Datenbankgateways, der die Datenabfragen vollständig übernimmt und das Aufsetzen einer neuen Zielarchitektur entsprechend der notwendigen, abzubildenden Anforderungen.

5.5 Zusammenfassung

In 5.1 und 5.2 wurden die Erkenntnisse der Aus- und Bewertung von strukturierten, semi-strukturierten und unstrukturierten Systemen vereinfacht zusammengefasst. Sie dienen als Übersicht, sind aber nur im Zusammenhang mit der schriftlichen Ausarbeitung aussagekräftig. Für jedes Szenario wurde anhand der qualitativen Bewertung eine grobe quantitative Einschätzung vorgenommen und vermerkt.

Grundlegende Erkenntnisse sind:

- Je unstrukturierter das abzulösende Bestandssystem ist, desto weniger eignet es sich für eine Migration, da diese auf ihre Vorteile, wie beispielsweise kein Parallelbetrieb von Alt- und Zielsystem, gegenüber der Neuentwicklung nur Nutzen kann, wenn die vorgesehene Interaktion keine zusätzlichen größeren Sanierungsaufwände zu verursachen. Eine interaktionslose Neuentwicklung würde weniger Aufwände erfordern und somit Zeit in Anspruch nehmen und dadurch Kosten verursachen, als eine unvorhersehbare Sanierung mit einer anschließenden Migration. Darüber hinaus ist das Risiko, dass eine Sanierung bis zum benötigten Grad gar nicht möglich ist, nicht unerheblich.

- Im Gegenteil, je strukturierter ein Bestandssystem ist, desto weniger praktikabel ist der Beginn einer Neuentwicklung *auf der grünen Wiese*, da aus den bestehenden Konzepten des Systems, Anhaltspunkte für die Implementierung der abzubildenden Anforderungen gezogen werden können. Eine Migration kann viele Nachteile einer Neuentwicklung ausgleichen und ihr Grundrisiko senken, indem Bestands- und Zielsystem zeitweise miteinander verbunden werden und so eine schrittweise Ablösung, mit kontinuierlich verbessertem Nutzen für die Organisation und die AnwenderInnen, vorgenommen werden kann.

- Ein allgemeines iteratives und inkrementelles Vorgehen ist aller Voraussicht nach ein zeitaufwändigeres Vorhaben, bietet jedoch die Möglichkeit beim Auftritt von Fehlern, diese schrittweise analysieren zu können und bei Bedarf zurückrollen zu können. Dadurch kann ein präziseres Risikomanagement durchgeführt werden.
- Der Aspekt der Datenmigration kann entscheidend für die Wahl der nötigen Migrationsstrategie sein. Um Nachteile bestimmter Methoden auszugleichen, muss immer mit einem erhöhten Entwicklungsaufwand für unterstützende Elemente gerechnet werden. Diese können ihrerseits hochkomplex werden, so dass sie ebenso detailliert betrachtet werden müssen, wie die Systementwicklung.
- Je komplexer die angestrebte Zielarchitektur ist, desto detaillierter muss die Projekt- und Infrastrukturplanung praktiziert werden, um die Komplexität des Vorhabens beherrschen zu können.

Zustand	strukturiertes System				semi-strukturiertes System				unstrukturiertes System			
Ziel	**Gegebenheiten**	**Strategie**	**Szenarien**	**Bewertung**	**Gegebenheiten**	**Strategie**	**Szenarien**	**Bewertung**	**Gegebenheiten**	**Strategie**	**Szenarien**	**Bewertung**
Standard-software	- Zustand des Bestandsystems nicht relevant - Struktur des Systems hilft bei unterstützenden Entwicklungen	Big Bang	- Database Last mit Betriebsausfall, Nutzen erst nach der Datenmigration - Entwicklung eines dauerhaften Datenbankgateways, kein Betriebsausfall, direkter Nutzen	-- +	- Zustand des Bestandsystems nicht relevant - Struktur des Systems hilft nur bedingt bei unterstützenden Entwicklungen	Big Bang	- Database Last mit Betriebsausfall, Nutzen erst nach der Datenmigration - Entwicklung eines dauerhaften Datenbankgateways nach Sanierung der Datenbankschnittstelle, kein Betriebsausfall, direkter Nutzen	-- -	- Zustand des Bestandsystems nicht relevant - Struktur des Systems hilft nicht bei unterstützenden Entwicklungen	Big Bang	- Database Last mit Betriebsausfall, Nutzen erst nach der Datenmigration - Entwicklung eines dauerhaften Datenbankgateways nach aufwändiger Sanierung der Datenbankschnittstelle, kein Betriebsausfall, direkter Nutzen	-- -
		Butterfly	- Entwicklung eines Data Access Allocators, kein Betriebsausfall, direkter Nutzen	++		Butterfly	- Entwicklung eines Data Access Allocators, kein Betriebsausfall, direkter Nutzen	++		Butterfly	- Entwicklung eines Data Access Allocators, kein Betriebsausfall, direkter Nutzen	++
	Gegebenheiten	**Strategie**	**Szenarien**	**Bewertung**	**Gegebenheiten**	**Strategie**	**Szenarien**	**Bewertung**	**Gegebenheiten**	**Strategie**	**Szenarien**	**Bewertung**
Schichten-architektur	- Zustand des Systems ist relevant abhängig von der Modernisierungsstrategie - Struktur des Systems hilft bei unterstützenden Entwicklungen - Möglichkeit Teile des Systems wiederzuverwenden	Big Bang	- NE: Database Last mit Betriebsausfall, parallele Entwicklung, Nutzen erst nach der Datenmigration - NE: allgemeine Datenmigration ohne Betriebsausfall, parallele Entwicklung, Nutzen erst nach dem Big Bang	-- -	- Zustand des Systems ist relevant abhängig von der Modernisierungsstrategie - Struktur des Systems kann bei unterstützenden Entwicklungen helfen - eventuell Möglichkeit Teile des Systems wiederzuverwenden	Big Bang	- NE: Database Last mit Betriebsausfall, parallele Entwicklung, Nutzen erst nach der Datenmigration - NE: allgemeine Datenmigration ohne Betriebsausfall, parallele Entwicklung, Nutzen erst nach dem Big Bang	- o	- Zustand des Systems ist relevant abhängig von der Modernisierungsstrategie - Struktur des Systems kann nicht für unterstützende Entwicklungen genutzt werden - keine Möglichkeit Teile des Systems wiederzuverwenden	Big Bang	- NE: Database Last mit Betriebsausfall, parallele Entwicklung, Nutzen erst nach der Datenmigration - NE: allgemeine Datenmigration ohne Betriebsausfall, parallele Entwicklung, Nutzen erst nach dem Big Bang	- +
		Chicken Little	- NE: frei wählbare Datenmigration, Betriebsausfall vermeidbar, kontrollierbares Risiko, parallele Entwicklung, Zeitpunkt des erkennbaren Nutzens variabel - M: frei wählbare Datenmigration, Betriebsausfall vermeidbar, kontrollierbares Risiko, verbundene Entwicklung, Zeitpunkt des erkennbaren Nutzens variabel	o + bis ++		Chicken Little	- NE: frei wählbare Datenmigration, Betriebsausfall vermeidbar, kontrollierbares Risiko, parallele Entwicklung, Zeitpunkt des erkennbaren Nutzens variabel - M: frei wählbare Datenmigration, Betriebsausfall vermeidbar, kontrollierbares Risiko, verbundene Entwicklung, Zeitpunkt des erkennbaren Nutzens variabel	o o		Chicken Little	- NE: frei wählbare Datenmigration, Betriebsausfall vermeidbar, kontrollierbares Risiko, parallele Entwicklung, Zeitpunkt des erkennbaren Nutzens variabel - M: frei wählbare Datenmigration, Betriebsausfall vermeidbar, kontrollierbares Risiko, verbundene Entwicklung, Zeitpunkt des erkennbaren Nutzens variabel	+ --
		Butterfly	- NE: frei wählbares Systementwicklungsvorgehen, kontrollierbares Risiko, kein Betriebsausfall - M: frei wählbares Systementwicklungsvorgehen, kontrollierbares Risiko, kein Betriebsausfall	++ ++		Butterfly	- NE: frei wählbares Systementwicklungsvorgehen, kontrollierbares Risiko, kein Betriebsausfall - M: frei wählbares Systementwicklungsvorgehen, kontrollierbares Risiko, kein Betriebsausfall	+ +		Butterfly	- NE: frei wählbares Systementwicklungsvorgehen, kontrollierbares Risiko, kein Betriebsausfall - M: frei wählbares Systementwicklungsvorgehen, kontrollierbares Risiko, kein Betriebsausfall	+ --

Abbildung 5.1: Zusammenfassung der Bewertung (Teil 1)

Abbildung 5.2: Zusammenfassung der Bewertung (Teil 2)

Zustand	strukturiertes System				semi-strukturiertes System				unstrukturiertes System			
Ziel	**Gegebenheiten**	**Strategie**	**Szenarien**	**Bewertung**	**Gegebenheiten**	**Strategie**	**Szenarien**	**Bewertung**	**Gegebenheiten**	**Strategie**	**Szenarien**	**Bewertung**
Microservices	- Zustand des Systems ist relevant abhängig von der Modernisierungsstrategie - Struktur des Systems hilft beim Architektur- und Infrastrukturentwurf - wenige bis keine Übernahmemöglichkeiten bestehenden Quellcodes - Datenbankmigration hochkomplex und kann mit den klassischen Migrationsmethoden nicht abgebildet werden	Big Bang	-NE: nur in Kombination mit Butterfly oÄ einsetzbar, parallele Entwicklung, hohes Umstellungsrisiko, keine Interaktion mit Bestandssystem nötig	--	- Zustand des Systems ist relevant abhängig von der Modernisierungsstrategie - Struktur des Systems hilft nur bedingt beim Architektur- und Infrastrukturentwurf - sehr wenige bis keine Übernahmemöglichkeiten bestehenden Quellcodes - Datenbankmigration hochkomplex und kann mit den klassischen Migrationsmethoden nicht abgebildet werden	Big Bang	-NE: nur in Kombination mit Butterfly oÄ einsetzbar, parallele Entwicklung, hohes Umstellungsrisiko, keine Interaktion mit Bestandssystem nötig	o	- Zustand des Systems ist relevant abhängig von der Modernisierungsstrategie - Struktur des Systems hilft nicht beim Architektur- und Infrastrukturentwurf - keine Übernahmemöglichkeiten bestehenden Quellcodes - Datenbankmigration hochkomplex und kann mit den klassischen Migrationsmethoden nicht abgebildet werden	Big Bang	-NE: nur in Kombination mit Butterfly oÄ einsetzbar, parallele Entwicklung, hohes Umstellungsrisiko, keine Interaktion mit Bestandssystem nötig	+
		Chicken Little	-NE: nur in Kombination mit Butterfly oÄ einsetzbar, parallele Entwicklung, kontrollierbares Risiko, keine Interaktion mit Bestandssystem nötig	o		Chicken Little	-NE: nur in Kombination mit Butterfly oÄ einsetzbar, parallele Entwicklung, kontrollierbares Risiko, keine Interaktion mit Bestandssystem nötig	+		Chicken Little	-NE: nur in Kombination mit Butterfly oÄ einsetzbar, parallele Entwicklung, kontrollierbares Risiko, keine Interaktion mit Bestandssystem nötig	++
			-M: nur in Kombination mit Butterfly oÄ einsetzbar, verbundene Entwicklung, kontrollierbares Risiko, Interaktion mit Bestandssystem nötig	++			-M: nur in Kombination mit Butterfly oÄ einsetzbar, verbundene Entwicklung, kontrollierbares Risiko, Interaktion mit Bestandssystem nötig	o			-M: nur in Kombination mit Butterfly oÄ einsetzbar, verbundene Entwicklung, kontrollierbares Risiko, Interaktion mit Bestandssystem nötig	--
		Butterfly	-NE: in Kombination mit Big Bang praxistaugliches Vorgehen für die DB-Migration, teilweise kontrollierbares Risiko, kein Betriebsausfall, keine Interaktion mit Bestandssystem nötig	--		Butterfly	-NE: in Kombination mit Big Bang praxistaugliches Vorgehen für die DB-Migration, teilweise kontrollierbares Risiko, kein Betriebsausfall, keine Interaktion mit Bestandssystem nötig	o		Butterfly	-NE: in Kombination mit Big Bang praxistaugliches Vorgehen für die DB-Migration, teilweise kontrollierbares Risiko, kein Betriebsausfall, keine Interaktion mit Bestandssystem nötig	+
			-NE: in Kombination mit Chicken Little praxistaugliches Vorgehen für die DB-Migration, kontrollierbares Risiko, kein Betriebsausfall, keine Interaktion mit Bestandssystem nötig	o			-NE: in Kombination mit Chicken Little praxistaugliches Vorgehen für die DB-Migration, kontrollierbares Risiko, kein Betriebsausfall, keine Interaktion mit Bestandssystem nötig	+			-NE: in Kombination mit Chicken Little praxistaugliches Vorgehen für die DB-Migration, kontrollierbares Risiko, kein Betriebsausfall, keine Interaktion mit Bestandssystem nötig	++
			-M: in Kombination mit Chicken Little praxistaugliches Vorgehen für die DB-Migration, kontrollierbares Risiko, kein Betriebsausfall, Interaktion mit Bestandssystem nötig	++			-M: in Kombination mit Chicken Little praxistaugliches Vorgehen für die DB-Migration, kontrollierbares Risiko, kein Betriebsausfall, Interaktion mit Bestandssystem nötig	o			-M: in Kombination mit Chicken Little praxistaugliches Vorgehen für die DB-Migration, kontrollierbares Risiko, kein Betriebsausfall, Interaktion mit Bestandssystem nötig	--
	Gegebenheiten	**Strategie**	**Szenarien**	**Bewertung**	**Gegebenheiten**	**Strategie**	**Szenarien**	**Bewertung**	**Gegebenheiten**	**Strategie**	**Szenarien**	**Bewertung**
Monolith	- Zustand des Systems ist relevant abhängig von der Modernisierungsstrategie - Struktur des Systems hilft bei unterstützenden Entwicklungen und beim Architekturentwurf - Möglichkeit Teile des Systems wiederzuverwenden	Big Bang	- NE: Database Last mit Betriebsausfall, parallele Entwicklung, Nutzen erst nach der Datenmigration, SPoF und Big Bang Risiko	--	- Zustand des Systems ist relevant abhängig von der Modernisierungsstrategie - Struktur des Systems kann bei unterstützenden Entwicklungen und beim Architekturentwurf helfen - eventuell Möglichkeit Teile des Systems wiederzuverwenden	Big Bang	- NE: Database Last mit Betriebsausfall, parallele Entwicklung, Nutzen erst nach der Datenmigration, SPoF und Big Bang Risiko	--	- Zustand des Systems ist relevant abhängig von der Modernisierungsstrategie - Struktur des Systems kann nicht bei unterstützenden Entwicklungen und beim Architekturentwurf helfen - wenige bis keine Möglichkeit Teile des Systems wiederzuverwenden	Big Bang	- NE: Database Last mit Betriebsausfall, parallele Entwicklung, Nutzen erst nach der Datenmigration, SPoF und Big Bang Risiko	--
			- NE: allgemeine Datenmigration ohne Betriebsausfall, parallele Entwicklung, Nutzen erst nach dem Big Bang, SPoF und Big Bang Risiko	-			- NE: allgemeine Datenmigration ohne Betriebsausfall, parallele Entwicklung, Nutzen erst nach dem Big Bang, SPoF und Big Bang Risiko	-			- NE: allgemeine Datenmigration ohne Betriebsausfall, parallele Entwicklung, Nutzen erst nach dem Big Bang, SPoF und Big Bang Risiko	-
		Chicken Little	- NE: frei wählbare Datenmigration, Betriebsausfall vermeidbar, kontrollierbares Risiko, parallele Entwicklung, Zeitpunkt des erkennbaren Nutzens variabel, Fehler besser eingrenzbar	o		Chicken Little	- NE: frei wählbare Datenmigration, Betriebsausfall vermeidbar, kontrollierbares Risiko, parallele Entwicklung, Zeitpunkt des erkennbaren Nutzens variabel, Fehler besser eingrenzbar	o		Chicken Little	- NE: frei wählbare Datenmigration, Betriebsausfall vermeidbar, kontrollierbares Risiko, parallele Entwicklung, Zeitpunkt des erkennbaren Nutzens variabel, Fehler besser eingrenzbar	+
			- M: frei wählbare Datenmigration, Betriebsausfall vermeidbar, kontrollierbares Risiko, verbundene Entwicklung, Zeitpunkt des erkennbaren Nutzens variabel, Fehler besser eingrenzbar	+ bis ++			- M: frei wählbare Datenmigration, Betriebsausfall vermeidbar, kontrollierbares Risiko, verbundene Entwicklung, Zeitpunkt des erkennbaren Nutzens variabel, Fehler besser eingrenzbar	o			- M: frei wählbare Datenmigration, Betriebsausfall vermeidbar, kontrollierbares Risiko, verbundene Entwicklung, Zeitpunkt des erkennbaren Nutzens variabel, Fehler besser eingrenzbar	--
		Butterfly	- NE: frei wählbares Systementwicklungsvorgehen, kontrollierbares Risiko, kein Betriebsausfall	++		Butterfly	- NE: frei wählbares Systementwicklungsvorgehen, kontrollierbares Risiko, kein Betriebsausfall	++		Butterfly	- NE: frei wählbares Systementwicklungsvorgehen, kontrollierbares Risiko, kein Betriebsausfall	++
			- M: frei wählbares Systementwicklungsvorgehen, kontrollierbares Risiko, kein Betriebsausfall	++			- M: frei wählbares Systementwicklungsvorgehen, kontrollierbares Risiko, kein Betriebsausfall	+			- M: frei wählbares Systementwicklungsvorgehen, kontrollierbares Risiko, kein Betriebsausfall	--

6 Validierung der Ergebnisse

Um die in Kapitel 5 vorgenommene Bewertung der Ablösungsstrategien zu validieren, werden in diesem Kapitel zwei anonymisierte Ausgangsszenarios aus der Praxis auf der gleichen argumentativen Grundlage von Kapitel 5 untersucht und das jeweilige Ergebnis mit den Erkenntnissen 5.1 und 5.2 verglichen.

6.1 Szenario 1

Die Anwendungslandschaft einer Organisation setzt sich aus zahlreichen kleinen bis mittelgroßen Anwendungen zusammen, für deren Entwicklung ein umfangreiches Framework zur Verfügung gestellt wird. Initial wurden die Applikationen modular als Schichtenarchitekturen geplant und umgesetzt. Einige bilden geschäftskritische Vorgänge, wie das Zurverfügungstellen und Verwalten von Kundenstammdaten ab. Teilweise sind sie das Ergebnis von Modernisierungsmaßnahmen aus *COBOL*-Zeiten, teilweise wurden vollkommen neue Geschäftsprozesse abgebildet. Die jüngsten Anwendungen sind seit 5 und die ältesten seit 15 Jahre produktiv im Einsatz. Die vorherrschenden Technologien sind *Java EE*, *Enterpise Java Beans*, sowie Webservices. Als Buildmanagementtool ist *Maven* im Einsatz. Die verwendete *IBM DB2*-Datenbank ist relational.

Durch die jahrelangen Wartungs- und Erweiterungstätigkeiten mit wechselndem Personal konnte die Modularität und die Hierarchien, die ausschlaggebend für Schichtenarchitekturen sind, nicht erhalten werden. Versteckte zyklische Abhängigkeiten kommen besonders in den älteren Anwendungen häufig vor. Da das Framework nicht alle benötigten Abhängigkeiten zur Abbildung der Anforderungen zur Verfügung stellt und die Möglichkeit eigene Abhängigkeiten einzufügen auch nicht konsequent unterbindet, sind viele Bibliotheken in verschiedenen Versionen mehrfach in Projekten verfügbar. Dokumentationen sind nur in Ausnahmefällen auf aktuellem Stand verfügbar. Das Buildmanagementtool durchläuft bei jeder Auslieferung die verfügbaren Tests. Diese sind jedoch qualitativ nicht aussagekräftig und häufig komplett auskommentiert.

6.1.1 Analyse

Die Migration des Datenbestandes wird hier von vorneherein ausgeklammert, da die Datenhaltung des relationalen Datenbanksystems noch von weiteren Anwendungen genutzt wird. Diese sind von den Modernisierungsvorhaben nicht betroffen. Daher muss der Zustand der Datenbank genauso erhalten werden.

Anhand der beschriebenen Ausgangssituation muss hier von *unstrukturierten* oder *hybriden Bestandssystemen* mit einem hohen Anteil unstrukturierter Komponenten ausgegangen werden. Folglich liegt die schlechteste Ausgangslage für eine Ablösung vor, bei welcher der Zustand des Systems für diese relevant ist. Die *Schichtenarchitektur* als Ziel der Modernisierung wird von der Organisation festgelegt.

Aufgrund der Beschaffenheit der Bestandssysteme können bei Entwicklung der Zielarchitektur und bei zusätzlichen Entwicklungen, die die Ablösung unterstützen könnten, keine Hilfestellungen angeboten werden. Auch können aufgrund des Zusammenschlusses aller Schnitttellen, der Anwendungslogik und der Datenbankservices keine Strukturen wiederverwendet werden. Im Kontext der einzelnen Anwendungen muss bewertet werden, ob sich eine Sanierung der Schnittstellen lohnt, um die bestehende Infrastruktur der Anwendungslandschaft erhalten zu können.
Im Zusammenhang mit unstrukturierten Systemen ist die erste Annahme, dass eine interaktionslose Neuentwicklung erfolgsversprechender ist als Migrationen, die eine Kommunikation zwischen Alt- und Zielsystem erfordern.

Bei einer Neuentwicklung wären die *Big Bang*- und die *Chicken Little*-Strategie die möglichen Wege. Es muss somit eine Einschätzung vorgenommen werden, ob ein vergleichsweise schnelles Entwicklungsvorgehen mit einem hohen Umstellungsrisiko oder ein zeitintensiveres Vorgehen mit inkrementell und iterativ kontrollierbarem Risiko angewendet werden soll. In beiden Fallen muss, bis zur Ablösung der jeweiligen Anwendung, der parallele Betrieb von Alt- und Zielsystem und die analoge Umsetzung sich ändernder Anforderungen berücksichtigt werden.

Wenn sich für eine Migration entschieden wird, kann die *Chicken Little*-Strategie angewendet werden. Um die benötigte Interaktion zwischen Alt- und Zielsystem ermöglichen zu können, muss das Bestandssystem aufwändig saniert werden, so dass die inkrementelle und iterative Ablösung positive Auswirkungen auf das Migrationsvorhaben haben kann. Unter diesem Gesichtpunkt stellt sich also die Frage, ob die Sanierungsaufwände und die anschließende zeitintensivere Migration ein sinnvolles Vorgehen im Gegensatz zu einer Neuentwicklung ist.

6.1.2 Bewertung

Unter Berücksichtigung der Tabelle 5.1 sind die am besten bewerteten Vorgehen auf Basis unstrukturierter Systeme, welche eine *Schichtenarchitektur* als Ziel haben:

- Eine Neuentwicklung mit der *Big Bang*-Strategie in Kombination mit einer Datenmigrationsmethode, die einen Betriebsausfall verhindert, was in diesem Fallbeispiel nicht berücksichtigt werden muss.
- Eine Neuentwicklung mit der *Chicken Little*-Strategie, die auf kontrollierbares Risiko durch das inkrementelle und iterative Vorgehen setzt.
- Eine Neuentwicklung mit der *Butterfly*-Methode, zu der ein entsprechendes Systementwicklungsverfahren gewählt werden kann. Da in diesem Szenario jedoch keine Datenmigration vorgenommen werden soll, ist sie in dieser Bewertung ohne Relevanz.

Laut der Tabelle 5.1 sind die schlechtesten Vorgehen:

- Eine Migration mit der *Chicken Little*-Strategie, aufgrund der notwendigen Interaktion zwischen Alt- und Zielsystem und den zuvor notwendigen Sanierungsaufwänden.

- Eine Migration mit der *Butterfly*-Methode, da für das gewählte Systementwicklungsverfahren ebenfalls die Interaktion der Systeme durch die Aufarbeitung des Bestandssystems ermöglicht werden müsste. Da keine Datenmigration in diesem Szenario vorgenommen werden soll, ist dieses Vorgehen in dieser Bewertung ebenfalls ohne Relevanz.

Die in 5.1 festgehaltenen Bewertungen aus Kapitel 5 decken sich somit mit den Ergebnissen aus der Analyse des beschriebenen Szenarios. Als Vorgehen für die Ablösung sollte die *Big Bang*- oder *Chicken Little*-Strategie, für eine komplette interaktionslose Neuentwicklung, ausgewählt werden.

6.2 Szenario 2

In einer Organisation ist eine Kaufsoftware im Einsatz, welche für diese von hoher Bedeutung ist. Da in dieser Software personenbezogene Gesundheitsdaten verwendet werden, handelt es sich um besonders sensible und schützenswerte Daten. Die im Hintergrund durchgeführten Berechnungen sind umfangreich und komplex, da alle Leistungen der gesetzlichen und privaten Versicherungen im europäischen Raum abgebildet werden können. Aus diesem Grund ist der Anbieter der Software am Markt weitestgehend konkurrenzlos. Die Benutzerschnittstellen der Software sind nicht benutzerfreundlich und das Laden von Daten aus der relationalen Datenbank verursacht lange Wartezeiten. Die Daten werden über ein Datenbankgateway abgefragt, das beim Kauf der Software in kürzester Zeit und nicht nachhaltig innerbetrieblich entwickelt wurde, da keine Entwicklungsunterstützung des Anbieters angeboten wurde. Um die Ladezeiten zu verkürzen, werden die Daten nach dem erstmaligen Laden unverschlüsselt auf einem Computer, der Zugang zum Internet besitzt, zwischengespeichert. Es besteht somit zusätzlich die Notwendigkeit der Ablösung aufgrund von Sicherheitsaspekten. Der Anbieter stellt für seine Kunden die Berechnungsfunktionen zur direkten Einbindung in Individualentwicklungen über eine REST-Schnittstelle zur Verfügung. Die in der Organisation verwendeten Technologien sind *Java EE*, *Spring*, *Maven*, sowie *COBOL*.

6.2.1 Analyse

Die Migration des Datenbestandes wird hier ausgeklammert, da die relationale Datenbank ebenfalls für weitere Anwendungen der Organisation Daten zur Verfügung stellt und sich ihre Datenhaltung aus diesem Grund nicht verändern darf.

Aufgrund dessen, dass es am Markt keine Anwendung gibt, die die Anforderungen der Organisation in diesem Maße abbilden kann, ist der Kauf einer alternativen Standardsoftware als Ablösungsmethode ausgeschlossen.

Da es sich bei der Ausgangssituation um eine Kaufsoftware mit einer ergänzenden Individualentwicklung handelt, deren Qualität ungewiss ist, muss hier von einer *Blackbox*, beziehungsweise einem *unstrukturierten Bestandssystem*, ausgegangen werden. Es liegt also eine nachteilige Ausgangslage für eine Ablösung vor, bei welcher der Zustand des Bestandssystems für diese relevant ist. Als Zielarchitekturen kommen für die Organisation sowohl eine *Schichtenarchitektutr*, als auch ein *Deployment-Monolith* in Frage.

Aus der Kaufsoftware und dem unstrukturierten Datenbankgateway lassen sich keine bis sehr wenige Informationen für die mögliche Struktur der zukünftigen Zielarchitektur sowie für unterstützende Zusatzentwicklungen für die Modernisierung ziehen. Eine Sanierung der Schnittstellen und Abfragen des Gateways können jedoch womöglich mit Sanierungsaufwänden die Entwicklung eines neuen Datenbankservices unterstützen. Anhand der Blackbox-Ausgangslage erscheint eine Neuentwicklung als das einzige mögliche Vorgehen. Das Datenbankgateway könnte durch eine Migration abgelöst werden.

Eine Neuentwicklung kann mit der *Big Bang* oder *Chicken Little*-Strategie durchgeführt werden. In beiden Fällen muss der Parallelbetrieb zwischen Alt- und Zielsystem und die Umsetzung sich ändernder Anforderungen im Zielsystem beachtet werden. Das Big Bang-Verfahren kann vergleichsweise schneller durchgeführt werden, hat jedoch ein hohes Umstellungsrisiko. Im Gegensatz dazu ist eine Neuentwicklung mit Chicken Little zeitintensiver, hält jedoch das Risiko durch den inkrementellen und iterativen Ansatz kontrollierbarer. Wenn es spezifisch um die Umstellung auf eine *Schichtenarchitektur* geht, müssen darüber hinaus, da die Datenmigration außen vor gelassen werden kann, keine weiteren Faktoren bei der Entscheidungsfindung beachtet werden. Wenn ein *Monolith* entwickelt werden soll, summiert sich bei Anwendung der Big Bang-Strategie das Risiko gemeinsam mit dem *Single Point of Failure* des Monolithen, was das Risikomanagement zusätzlich erschwert. Eine Chicken Little-Neuentwicklung muss diesen Nachteil ebenfalls umgehen. Durch den inkrementellen und iterativen Ansatz sind die Fehler im Falle ihres Eintreffens besser eingrenzbar.

Wenn sich für eine teilweise Migration des Datebankgateways entschieden wird, müssen erhöhte Sanierungsaufwände einkalkuliert werden, um die *Chicken Little*-Strategie durchführen zu können. Da es sich nur um eine Komponente handelt, kann dies praktikabel sein und das iterative und inkrementelle Vorgehen kann risikoarm und zeitgleich mit der Neuentwicklung der Ablösung der Standardsoftware stattfinden.

6.2.2 Bewertung

Bei Betrachtung der Tabelle 5.2 sind die besten Vorgehen auf Basis unstrukturierter Systeme, welche einen *Monolithen* als Ziel haben:

- Eine Neuentwicklung auf Basis der *Chicken Little*-Strategie, durch das inkrementelle und iterative Entwicklungsverfahren und die Möglichkeit, durch sie auch die möglichen Fehler besser eingrenzen zu können.

- Eine Neuentwicklung mit der *Butterfly*-Strategie, für die ein passendes Systementwicklungsverfahren ausgewählt werden kann. In diesem Szenario ist jedoch keine Datenmigration vorgesehen, daher entfällt diese Option.

Die schlechtesten Vorgehen für einen *Monolithen* auf Basis unstrukturierter Systeme sind:

- Eine Migration mit der *Chicken Little*-Strategie, aufgrund der Sanierungaufwände, die für die Ermöglichung der Interaktion zwischen Alt- und Zielsystem investiert werden müssten.
- Eine Migration mit der *Butterfly*-Strategie den gleichen, zuvor genannten Gründen. Diese ist aufgrund ihres primären Ziels der Datenmigration in diesem Szenario jedoch ohne Relevanz.
- Eine Neuentwicklung mit der *Big Bang*-Strategie, aufgrund des extrem hohen Risikos, bei der Kombination des *Single Point of Failures* und der Big Bang-Umstellung.

Aus Tabelle 5.1 lassen sich für die Umstellung eines unstrukturierten Bestandssystems auf *Schichtenarchitekturen* folgende gute Vorgehen ableiten:

- Eine Neuentwicklung mit der *Big Bang*-Strategie in Kombination mit einer Datenmigrationsmethode, die einen Betriebsausfall verhindert, was in diesem Fallbeispiel nicht berücksichtigt werden muss.
- Eine Neuentwicklung mit der *Chicken Little*-Strategie, die auf kontrollierbares Risiko durch das inkrementelle und iterative Vorgehen setzt.
- Eine Neuentwicklung mit der *Butterfly*-Methode, zu der ein entsprechendes Systementwicklungsverfahren gewählt werden kann. Da in diesem Szenario jedoch keine Datenmigration vorgenommen werden soll, ist dieses Vorgehen in dieser Bewertung ohne Relevanz.

Laut der Tabelle 5.1 sind bei *Schichtenarchitekturen* mit unstrukturierten Bestandssystemen die schlechtesten Vorgehen:

- Eine Migration mit der *Chicken Little*-Strategie, aufgrund der notwendigen Interaktion zwischen Alt- und Zielsystem und den zuvor notwendigen Sanierungsaufwänden.
- Eine Migration mit der *Butterfly*-Methode, da für das gewählte Systementwicklungsverfahren ebenfalls die Interaktion der Systeme durch die Aufarbeitung des Bestandssystems ermöglicht werden müsste. Da keine Datenmigration in diesem Szenario vorgenommen werden soll, ist dieses Vorgehen in dieser Bewertung ebenfalls ohne Relevanz.

Diese aus Kapitel 5 festgehaltenen Bewertungen decken sich mit den Erkenntnissen, die im Zusammenhang mit dem beschriebenen Szenario gewonnen werden konnten. Aufgrund ihrer größeren Flexibilität für zukünftige Entwicklungen und mehr Entscheidungsmöglichkeiten, sollte als Ziel die Entwicklung einer *Schichtenarchitektur* gegenüber der Entwicklung eines *Monolithen* vorgezogen werden. Die Neuentwicklung der Kaufsoftware kann mit der *Big Bang* oder *Chicken Little*-Strategie durchgeführt werden. Das Datenbankgateway kann ebenfalls mit diesen Methoden neuentwickelt werden oder mit Sanierungsaufwänden und *Chicken Little* migriert werden.

7 Fazit

7.1 Würdigung der Ergebnisse

In dieser Arbeit wurden Modernisierungsszenarien anhand ihrer Ausgangssituationen auf ihre möglichen notwendigen und praktikablen Ablösungsmöglichkeiten untersucht. Dazu wurden zunächst die Optionen zur Modernisierung definiert. Danach wurden die Strategien betrachtet und mit den Optionen verknüpft, um auf dieser Ebene schon erste widersprüchliche Konstellationen filtern zu können. Daraufhin folgte die Aufarbeitung der Modernisierungsoptionen und -strategien anhand konkreter theoretischer Modernisierungsszenarien mit der Ausrichtung auf die Zielarchitekturen. Anschließend wurden zwei anonymisierte Beispiele aus der Praxis analysiert und auf Grundlage der Analyse validiert, ob die vorgenommene Bewertung der theoretischen Modernisierungsszenarien auf reale Szenarien anwendbar ist.

Das Ergebnis ist eine ganzheitliche Betrachtung aller Modernisierungsoptionen und -strategien für verschiedene Modernisierungsvorhaben in Kombination mit den gängigsten Zielarchitekturen. Diese ist übersichtlich in zwei Tabellen zusammengefasst und bildet im Zusammenhang mit der Ausarbeitung eine abstrahierte und trotzdem noch ausreichend konkrete Unterstützung. Sie diesen als technische Argumentationsgrundlage für das Vorgehen realer geplanter Modernisierungsszenarien, ohne die organisatorischen Aspekte außer acht zu lassen.

7.1.1 Optionen und Strategien zur Modernisierung

Die Aufarbeitung der Modernisierungsoptionen und -strategien bereitete größere Schwierigkeiten als zuvor angenommen. In der Literatur werden Optionen und Strategien oft gleichgesetzt und nicht differenziert betrachtet. Es werden detaillierte Migrationsschritte auf Basis des Zustandes des Altsystems aufgelistet, jedoch keine allumfassende Betrachtung vorgenommen, für welche Systeme und aus welchem Grund, diese Migrationsschritte nicht gelingen können. Dieser Umstand machte es deutlich aufwändiger, die Probleme und Risiken den entsprechenden Vorgehen zuzuordnen.

Das Thema der Datenmigration wurde in der vorliegenden Literatur nur beiläufig behandelt, obwohl diese für das gewählte Vorgehen, im Zusammenhang mit geschäftskritischen Prozessen, die primären Risiken birgt. Aus diesem Grund benötigt sie besondere Berücksichtigung. Darüber hinaus wird nirgendwo eine flexible Anwendung der angedachten Datenmigrationsmethoden in Betracht gezogen. Die drei Ablösungsgrundstrategien bieten keinen Interpretationsspielraum in Bezug auf die in ihrem Rahmen anzuwendenden Datenmigrationsmethoden. Sie sind daher, so wie sie bisher in der Literatur referenziert wurden, zur Anwendung zu weit von den flexiblen Problemlösungen

entfernt, die die Praxis erfordert. Dies hat die Aufarbeitung der Kombinationen von Optionen und Strategien zusätzlich erschwert, da diese Ausarbeitung die klare Zielsetzung hatte, als praktisches Werkzeug für reale Problemlösungen verwendet werden zu können.

7.2 Generalisierung

Die Einordnung, der Modernisierungsoptionen und -strategien in den Kontext einer konkreten Zielarchitektur hat die zahlreichen zu bedenkenden Punkte eines Modernisierungsvorhaben aufgedeckt. Die Ausgangssituation, aus welcher die Ablösung eines Bestandssystems erfolgen soll, ist ebenso relevant wie das angestrebte Ziel. Die Wahl der Datenmigration kann ebenso ein ausschlaggebender Faktor sein.

Anhand der vorgenommen qualitativen technischen Bewertung, die in den Ergebnistabellen festgehalten wurde, lassen sich mögliche Vorgehen für reale Modernisierungsvorhaben ableiten. Wenn das Ziel der Modernisierung eine der betrachteten gängigen Zielarchitekturen ist, lassen sich die Tabellen aufgrund ihrer ganzheitlichen Analyse als Argumentationsgrundlage für jedes praktische Szenario anwenden. Zudem können mithilfe der dargestellten Tabellen auch die Konstellationen mehrerer Szenarien bewertet werden.

Letztendlich müssen die Optionen und Strategien jedoch auch anhand organisatorischer Faktoren, wie Risikomanagement, Zeit und Kosten bewertet werden, um so das für das Vorhaben passende Vorgehen auswählen zu können.

7.3 Ausblick

Letztendlich hat die vorgenommene Abgrenzung, die anschließende Zusammenführung und das Durchspielen aller Möglichkeiten gezeigt, dass die in der Praxis häufig abgetane komplette Neuentwicklung, wenn alle Variablen im technischen Kontext betrachtet werden, ein sehr empfehlenswertes Vorgehen sein kann.

Modernisierungsvorhaben haben immer auch einen organisatorischen und personellen Aspekt. Unabhängig davon, ob es bei einem Projektvorhaben um die Ablösung eines Bestandssystems oder die Abbildung eines neuen Prozesses geht, wirken diese Aspekte meist schwerwiegender, als die Erkenntnisse einer isolierten technischen Einschätzung.

Literaturverzeichnis

[Aschoff, 2017] Aschoff, M. (2017). GUI-Redesign nach Continuous-Delivery-Prinzipien - Formschön. *iX Developer*, Jan/Feb.

[Ballüder et al., 2020] Ballüder, K., Pietsch, S.-L., and Langmack, B. (2020). So bringen Sie Legacy-Anwendungen in die Cloud. *Computerwoche, Online-Artikel*. https://www.computerwoche.de/a/so-bringen-sie-legacy-anwendungen-in-die-cloud Abgerufen am: 16.03.2020.

[Barashkov, 2019] Barashkov, A. (2019). Microservices vs. Monolith Architecture. Online-Artikel. https://medium.com/pixelpoint/microservices-vs-monolith-architecture-c7e43455994f Abgerufen am: 15.03.2020.

[Basedow, 2017] Basedow, J. (2017). Abhängigkeiten in Legacy-Systemen verwaltbar machen - Unter Kontrolle. *iX Developer*, Jan/Feb.

[Bisbal et al., 1998] Bisbal, J., Lawless, D., Wu, B., Grimson, J., Wade, V., Richardson, R., and O'Sullivan, D. (1998). An overview of legacy information system migration. pages 529 – 530.

[Bogdan and Spruth, 2013] Bogdan, P. D. M. and Spruth, P. D. W. G. (2013). Vorlesung Enterprise Computing: Einführung in z/OS. E-Book. https://www.informatik.uni-leipzig.de/cs/ebook/Band1/totalBand1.pdf Abgerufen am: 08.03.2020.

[Brautigam, 2019] Brautigam, R. (2019). Reevaluating the Layered Architecture. Website. https://dzone.com/articles/reevaluating-the-layered-architecture Abgerufen am: 02.03.2020.

[Brodie and Stonebraker, 1993] Brodie, M. L. and Stonebraker, M. (1993). DARWIN: On the Incremental Migration of Legacy Information Systems. Projektbericht, College of Engineering, University of California, Berkeley.

[Carnegie Mellon University, 2017] Carnegie Mellon University (2017). Software Architecture. Website. https://www.sei.cmu.edu/research-capabilities/all-work/display.cfm?customel_datapageid_4050=21328 Abgerufen am: 08.03.2020.

[Erdle, 2005] Erdle, C. (2005). Legacy Migrationsstrategien. Hauptseminar: Management von Softwaresystemen, TU Braunschweig.

[Evans and Fowler, 2004] Evans, E. and Fowler, M. (2004). *Domain-driven Design: Tackling Complexity in the Heart of Software*. Addison-Wesley.

[Feathers, 2004] Feathers, M. (2004). *Working Effectively with Legacy Code*. Prentice Hall PTR.

[GI and Hasselbring, 2006] GI and Hasselbring, W. (2006). Software-Architektur. Website. https://gi.de/informatiklexikon/software-architektur Abgerufen am: 08.03.2020.

[GI et al., 2005] GI, Richter, J.-P., Haller, H., and Schrey, P. (2005). Serviceorientierte Architektur. Website. https://gi.de/informatiklexikon/serviceorientierte-architektur Abgerufen am: 04.03.2020.

[Gimnich and Winter, 2005] Gimnich, R. and Winter, A. (2005). Workflows der Software-Migration. Technical report, Universität Koblenz-Landau, Iinstitut für Informatik.

[Hofmann et al., 2019] Hofmann, M., Lackner, M., Lilienthal, C., Röwekamp, L., and Schulz, M. (2019). *Von Monolithen und Microservices*. entwickler.press, Software & Support Media GmbH.

[IBM, 2020] IBM (2020). IBM Z Mainframe Server und Software. Website. https://www.ibm.com/de-de/it-infrastructure/z Abgerufen am: 08.03.2020.

[IDG, 2018] IDG, R. S. (2018). Studie Legacy-Modernisierung. Studie, IDG Buisness Media GmbH.

[Josuttis et al., 2009] Josuttis, N., Krafzig, D., and Tilkov, S. (2009). SOA-Manifest. Website. http://www.soa-manifest.de/ Abgerufen am: 04.03.2020.

[Kaps, 2017a] Kaps, S. (2017a). Migration eines Legacy Systems - Alles neu? *iX Developer*, Jan/Feb.

[Kaps, 2017b] Kaps, S. (2017b). Migrationsstrategien im Vergleich - Orientierungshilfe. *iX Developer*, Jan/Feb.

[Leitenberger, 2012] Leitenberger, B. (2012). Die Entwicklung der Programmiersprachen. Website. https://www.bernd-leitenberger.de/entwicklung-der-programmiersprachen.shtml Abgerufen am: 08.03.2020.

[Lilienthal, 2017a] Lilienthal, C. (2017a). *Langlebige Software-Architekturen: Technische Schulden analysieren, begrenzen und abbauen*, volume 2. dpunkt.verlag.

[Lilienthal, 2017b] Lilienthal, C. (2017b). Technische Schulden in Legacy-Code finden und bewerten - Erneuern oder ersetzen? *iX Developer*, Jan/Feb.

[Marston, 2012] Marston, T. (2012). What is the 3-tier architecture? Website. http://www.tonymarston.net/php-mysql/3-tier-architecture.html Abgerufen am: 24.02.2020.

[Merriam-Webster, 2020] Merriam-Webster (2020). Legacy. https://www.merriam-webster.com/dictionary/legacy Abgerufen am: 13.03.2020.

[Oracle Corporation, 2020] Oracle Corporation (2020). Oracle PL/SQL. https://www.oracle.com/database/technologies/application-development-PL/SQL.html Abgerufen am: 13.03.2020.

[Prüger, 2017] Prüger, B. (2017). Der Mainframe und seine Vorteile. Online-Artikel. https://www.channelpartner.de/a/der-mainframe-und-seine-vorteile,3050315 Abgerufen am: 08.03.2020.

[Raymond, 2003] Raymond, E. S. (2003). Basics of the unix philosophy. Website. http://www.catb.org/~esr/writings/taoup/html/ch01s06.html Abgerufen am: 20.02.2020.

[Rinne and Springer, 2017] Rinne, T. and Springer, S. (2017). Modernisierung von Legacy-Webapplikationen - Ran an den Altbau. *iX Developer*, Jan/Feb.

[Roden, 2017a] Roden, G. (2017a). Anwendungen strukturiert aufbauen - Meisterlich konstruiert. *iX Developer*, Jan/Feb.

[Roden, 2017b] Roden, G. (2017b). Software vorausschauend entwickeln - Gebrauchsprosa. *iX Developer*, Jan/Feb.

[SAP Germany GmbH, 2020] SAP Germany GmbH (2020). SAP. Website. https://www.sap.com/germany/index.html Abgerufen am: 12.03.2020.

[Schill and Springer, 2007] Schill, A. and Springer, T. (2007). *Verteilte Systeme: Grundlagen und Basistechnologien*. eXamen.press. Springer Berlin Heidelberg.

[Schmidt and Bär, 2014] Schmidt, R. and Bär, F. (2014). Ein Service-System orientiertes Rahmenwerk für System-Integration, -Migration und Konsolidierung. *Springer Fachmedien Wiesbaden, Online-Publikation*.

[Wegener, 2013] Wegener, I. (2013). *Komplexitätstheorie: Grenzen der Effizienz von Algorithmen*. Springer-Lehrbuch. Springer Berlin Heidelberg.

[Wolff, 2016] Wolff, E. (2016). *Microservices - Flexible Software Architecture*. Addison-Wesley Professional.

[Wolff, 2017] Wolff, E. (2017). Microservices - gerade in Legacy-Szenarien geeignet - Nie wieder Altlasten. *iX Developer*, Jan/Feb.

Ehrenwörtliche Erklärung

Ich versichere hiermit ehrenwörtlich, dass ich meine vorliegende Abschlussarbeit selbstständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel – insbesondere keine im Quellenverzeichnis nicht benannten Internet-Quellen – benutzt habe.

Die Arbeit wurde vorher nicht in einem anderen Prüfungsverfahren eingereicht und die eingereichte schriftliche Fassung entspricht derjenigen auf dem elektronischen Speichermedium.

Wörtlich oder dem Sinn nach aus anderen Werken entnommene Stellen sind unter Angabe der Quellen kenntlich gemacht.

Ort, Datum

Unterschrift (Vor- und Nachname)

Abstract

Julia Kordick

Migration von Bestandssystemen

Historisch gewachsene Legacy-Anwendungen sind in Unternehmen allgegenwärtig. Hohe Wartungskosten und Personalmangel sind nur zwei von vielen Argumenten, die das "am Leben erhalten" eines solchen Bestandssystems in Frage stellen. Möglichkeiten sind die Ablösung durch Standardsoftware, Neuentwicklungen oder aber Migrationen. Doch Ablösungen erfordern Strategien, um das Vorhaben gezielt durchführen zu können. In dieser Master-Thesis werden Ablösungsmöglichkeiten und -strategien, sowie gängige Zielarchitekturen in einen Kontext gesetzt. Das Ergebnis ist eine qualitative Bewertung mit technischer Fokussierung, um unter Einbeziehung aller Faktoren die richtige Entscheidung für Modernisierungsvorhaben treffen zu können.

Migration of Legacy Systems

Historically grown legacy systems are omnipresent in organizations. High maintenance costs and staff shortage are just two of many reasons to stop keeping these systems alive. Options for replacements are redevelopment, migration or detachment by standard software. These replacements need strategies to make modernization projects successful. In this master thesis replacement options, strategies and common tagret architectures are put into context. The result is an evaluation with technical focus to support coming to the right decision for modernization projects.

Abgabedatum: 31.03.2020