HAW HAMBURG

BACHELORTHESIS

Daniel Stark Galván

Microservices, eine langlebige Architektur ohne technische Schulden?

FAKULTÄT TECHNIK UND INFORMATIK
Department Informatik

Faculty of Computer Science and Engineering
Department Computer Science

HOCHSCHULE FÜR ANGEWANDTE WISSENSCHAFTEN HAMBURG
Hamburg University of Applied Sciences

Daniel Stark Galván

Microservices, eine langlebige Architektur ohne technische Schulden?

Bachelorarbeit eingereicht im Rahmen der Bachelorprüfung
im Studiengang *Bachelor of Science Informatik Technischer Systeme*
am Department Informatik
der Fakultät Technik und Informatik
der Hochschule für Angewandte Wissenschaften Hamburg

Betreuender Prüfer: Prof. Dr. Stefan Sarstedt
Zweitgutachter: Prof. Dr. Ulrike Steffens

Eingereicht am: 25.08.2022

Daniel Stark Galván

Thema der Arbeit

Microservices, eine langlebige Architektur ohne technische Schulden?

Stichworte

Microservice, langlebige Architektur, evolutionäre Architektur, technische Schulden, Softwareproduktschulden, Fitness Funktion

Kurzzusammenfassung

Microservices dominieren die Architekturlandschaft von Informationssystemen. Immer häufiger verwenden Unternehmen dieses Architekturmuster, um ihre Produktentwicklungszeit zu verkürzen. Die Betrachtung der langfristigen Nachteile von Microservices ist ein unerforschtes Gebiet. Daher ist es notwendig zu betrachten, welche langfristigen Folgen in der Form von technischen Schulden entstehen und ob eine Microservices-Architektur langlebig und anpassbar ist. Das Ziel in der vorliegenden Arbeit ist die Erfassung der langfristigen Folgen sowie deren Vermittlung und Visualisierung. Dazu wird die folgende Forschungsfrage gestellt: Sind Microservices eine langlebige Architektur ohne technische Schulden? Die Bewertung wurde anhand der kritischen Gegenüberstellung eines zu erfüllenden Rahmens durchgeführt. Spezifisch wurden die Einflüsse von Microservices auf die Stabilität, die Wartungskosten und die Anpassungsfähigkeit des Systems betrachtet. Die gewonnen Erkenntnisse zeigten den unzureichenden Vergleich anhand von technischen Schulden, die erhöhte Schwierigkeit architektonische Schulden in Microservices-Systeme zu identifizieren und die Notwendigkeit eines übergeordneten Konstruktes für Schulden. Darauf basierend werden Methodiken für die Behebung und Identifikation von architektonischen Schulden aufgeführt.

Daniel Stark Galván

Title of Thesis

Microservices, a sustainable Architecture without technical debt?

Keywords

Microservice, sustainable Architecture, evolutionary Architecture, technical debt, software product debt, fitness function

Abstract

Microservices dominate the architectural landscape of information systems. Companies are increasingly using this architecture pattern to optimize their Time-to-Market. The consideration of long-term disadvantages of microservices is unexplored territory. Therefore, it is necessary to consider what long-term consequences arise in the form of technical debt and whether a microservices architecture is durable and adaptable. The aim of this work is to record these long-term consequences and to convey them by visualization. To this end, the following research question is asked: Are microservices a sustainable architecture without technical debt? The evaluation was carried out based on a critical differentiation of the properties from Microservices and the ones to be fulfilled. Specifically, the influences of microservices on the stability, maintenance and adaptability of the system were reviewed. The gained results showed the insufficient comparison based on technical debt, the increased difficulty of identifying architectural debt in microservices systems and the need for a higher-level construct for debts. Based on this, methodologies for the elimination and identification of architectural debts are presented.

Inhaltsverzeichnis

Abbildungsverzeichnis

Tabellenverzeichnis

Listings

Abkürzungsverzeichnis

TTM	Time-to-Market
ATD	Architectural technical debt („architektonische technische Schuld “)
TD	Technical debt (technische Schulden)
DB	Datenbank (Database)
MRe	„Muss“-Regeln
FF	Fitness-Funktion
CD	Continuous Deployment
ADR	architectural decision record
KPI	Key Performance Indicator
FK	Foreign Key (Fremdschlüssel)
DDD	Domain Driven Design
QAW	Quality Attribute Workshop

VS	Verteiltes System
SoC	Separation of Concerns
UI	User Interface
SPA	Single-Page-App

Glossar

Time-to-Market	Messeinheit in Zeit (Stunde/Tage/Monate) bis ein Produkt marktreif ist und angeboten werden kann.
„Muss“-Regeln	Richtlinien oder Entscheidungen, welche befolgt werden müssen. Das Ignorieren solcher Richtlinien stellt ein hohes Risiko dar.
KPI	Kennzahl zur Messung des Erfolges oder der Produktivität.
QAW	Methode, um die kritischen architektonischen Merkmale zu identifizieren.
SOLID	Akronym von fünf Designprinzipien.

1 Einleitung

Die Geschichte der Menschheit wird stark vom Drang der Optimierung geprägt, folgendermaßen versuchen die Menschen seit Anfang der Steinzeit bessere Werkzeuge zu entwerfen und deren Abläufe effizienter und kostengünstiger zu gestalten. Beispiele für die stetige Innovation sind das Rad oder die archimedische Schraube[1] bis hin zur Digitalisierung von Prozessen und der Industrie 4.0[2]. Die Digitalisierung verspricht einen Produktivitätswachstum, weniger Kosten und höhere Effizienz.

Die digitale Welt wird von Produkten beherrscht, welche den Kunden Anpassungen und Innovationen in kürzester Zeit anbieten können. Je schneller ein Produkt platziert wird, desto schneller kann Profit geschöpft werden; diese kurze Time-to-Market[3] (TTM) Doktrin ist ein wesentlicher Bestandteil der gegenwärtigen Wirtschaft (Swamidass, 2000). Allerdings steigt mit der Erweiterung der digitalen Angebote sowie durch deren Verteilung die Komplexität und die Größe der IT-Lösungen. Demzufolge treffen zwei gegenseitige Ströme aufeinander: Schnelligkeit bei der Lieferung von Produkten und Komplexität der IT-Landschaften. Wie können Systeme schnell angepasst werden, welche komplexer werden? Wenn die Folgen einer Änderung nicht überschaut werden können, kann diese Änderung dann umgesetzt werden?

Aus Angst IT-Systeme durch Anpassungen zu verschlechtern und das Kerngeschäft zu gefährden, können sich IT-Systeme in einem Starrzustand befinden, wodurch ein kurzes Time-to-Market unerreichbar ist. Solche Systeme werden als Legacy Systems deklariert. Dylan Beattie traf eine passende Aussage zum Legacy Code, welches auf Systeme extrapoliert werden kann

[1] https://dewiki.de/Lexikon/Archimedische_Schraube

[2] https://www.plattform-i40.de/IP/Navigation/DE/Industrie40/WasIndustrie40/was-ist-industrie-40.html

[3] https://www.tcgen.com/time-to-market/

„*Code that's too scary to upadte and too profitable to delete*". Der unwirtschaftliche Zustand soll durch eine erhöhte Anzahl von technischen Schulden (TD) und Abhängigkeiten begünstig werden. Microservices sollen die Eindämmung von TD begünstigen und technische Möglichkeiten bieten, um schnelle Anpassungsfähige Systeme zu erschaffen.

Die kritische Beurteilung der Anpassungsfähigkeiten von Microservices Architekturen, die am häufigsten resultierenden technischen Schulden (TD) solcher Architekturen, deren Erkennungsstrategie und die Zukunftsfähigkeit, verkörpern das übergeordnete Ziel dieser wissenschaftlichen Arbeit. Hierfür werden im zweiten Kapitel die Grundlagen für die Beurteilung vermittelt, um im dritten Kapitel das Konzept einer angemessenen Architektur anhand einer langlebigen und evolutionären Architektur zu erarbeiten. Im vierten Kapitel werden die Grenzen vom Konstrukt TD dargestellt und erweitert. Die gewonnenen Erkenntnisse werden im fünften Kapitel der Microservcie-Architektur gegenübergestellt. Abschließend beschäftigt sich die Arbeit mit der Rekapitulation der Zukunftsfähigkeit von Microservices-Architekturen im sechsten Kapitel und einem Fazit im siebten Kapitel.

1.1 Problemstellung und Zielsetzung

Microservices-Systeme versprechen durch das „Divide and Conquer"-Prinzip[4] eine hohe Anpassungsfähigkeit, eine geringere Erosion der Architektur und eine resiliente Struktur, um zeitnah auf zukünftige Trends zu reagieren (Wolff, 2018). Zusätzlich soll die hohe Modularisierung von Microservices-Architekturen zu unabhängige Release-Zyklen führen, welche kürzere TTM-Zyklen ermöglichen.

Allerdings verlagert sich die Komplexität einzelner Module des Systems auf die Kommunikationsebene des gesamten Systems. Somit ist die Gesamtarchitektur nicht mehr leicht überschaubar. Der Fokus herkömmlicher Architektur-Tools befand sich bei der Herstellung auf monolithische Architekturen. Ist dieser Ansatz mit der Dezentralisierung von Microservices kompatibel? Wie können TD in der abstrakten Makroarchitekturschicht erkannt werden? Wie können diese objektiv erfasst, visualisiert und ausgewertet werden? Zusätzlich zu den

[4] https://de.wikipedia.org/wiki/Teile-und-herrsche-Verfahren

fehlenden Methodiken Schulden darzustellen, entsteht die Schwierigkeit diese in Arten zu kategorisieren. Unterschiede in den Anforderungen der Systeme oder Microservices führen zu unterschiedlichen Gewichtungen der Arten der TD.

Im Verlauf dieser Arbeit werden Vorgehensweisen entwickelt, die kritisch das Konzept der Anpassungsfähigkeit, Wartbarkeit und Wirtschaftlichkeit in Softwarearchitekturen, im Rahmen von Microservices-Systemen, beurteilen. Ziel ist es TD zu erkennen, kategorisieren und gewichten sowie eine kritische Beurteilung der Langlebigkeit von Microservices durchzuführen. Die gewonnenen theoretischen Erkenntnisse werden anhand von Praxisbeispielen revidiert.

2 Grundlagen

Zunächst wird in den folgenden Abschnitten ein Einblick in die Konzepte der technischen Schuld (Abschnitt 2.1) und der Softwarearchitektur (Abschnitt 2.2) mit den Eigenschaften langlebige Architektur (Abschnitt 2.2.1) und evolutionärer Architektur (Abschnitt 2.2.2) gegeben. Anschließend folgt die Beleuchtung von Microservice-Systemen (Abschnitt 2.3). Abschließend werden abstrakte Begriffe für den weiteren Verlauf der Arbeit konkretisiert und definiert (Abschnitt 2.4).

2.1 Technische Schulden

1992 prägte[5] Ward Cunningham die Metapher der „technischen Schulden“[6], welche das Entstehen suboptimaler technischer Lösungen im Bereich der Softwareentwicklung rahmt und aus Sicht von finanziellen Verbindlichkeiten betrachtet. C. Lilienthal fasst das Konzept TD wie

[5] http://c2.com/doc/oopsla92.html

[6] https://wiki.c2.com/?WardExplainsDebtMetaphor.

folgt zusammen: „*Technische Schulden 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, 2019, p. 4). Die Entstehung von TD erfolgt im gesamten Softwareentwicklungsprozess (*Figure 1*): Test und Qualitätssicherung; Dokumentation und Kommunikation; Organisation; Architektur und Programmierung (N. S. R. Alves, L. F. Ribeiro, V. Caires, T. S. Mendes and R. O. Spínola, 2014).

Figure 1: Softwareentwicklungsprozess

TD werden in drei Bestandteile zerteilt (Saulo S.de Toledo, Antonio Martini, Dag I.K.Sjøberg, 2021) Schuld (engl.: debt), Zinsen (engl.: interest) und Lösungskosten (engl.: principal).

- Schuld: Eine bewusste oder unbewusste getroffene suboptimale technische Lösung, welche zukünftige Kosten generiert. Ein bekanntes Beispiel einer TD ist der Verzicht auf automatisierte Softwaretests[7]. Im Vorfeld wird ein Zeitgewinn erwirtschaftet, nämlich die notwendige Entwicklungszeit für die Tests. Infolgedessen werden mehr Features im gleichen Zeitraum umgesetzt.
- Zinsen: Die zusätzlichen Kosten, welche im Softwareentwicklungsprozess entstehen, wenn die Schuld vorhanden ist. Die zuvor getroffene Entscheidung führt zu höheren Wartungskosten, zumal bei Softwareänderungen die Funktionalitäten der Software nicht automatisiert gewährleistet werden können. Stattdessen wird ein kostspieliger und manueller Sicherungsprozess benötig oder neue, in Zukunft zu behebende Fehler werden ins System eingebaut. Die Feature-Auslieferungsgeschwindigkeit leidet unter

[7] https://www.softwaretestinghelp.com/technical-debt-and-qa/

dieser Schuld. Die Zinsen sind der Aufwand bei jeder Änderung die Software manuell zu testen und die neuentstandenen Fehler zu beheben.

- Lösungskosten: Die Kosten, welche bei der Entwicklung einer optimalen Lösung und dem Ausbau der Schuld entstehen. Fortführend mit dem Beispiel, betragen die Lösungskosten die Zeit für die Entwicklung automatisierte Tests und die Behebung aller noch nicht entdeckten Fehler.

Der Schweregrad von TD wird anhand der Verhältnisse zwischen den Kostenquellen definiert. In *Figure 2* werden drei unterschiedliche TD in deren Grundsteine zerteilt, zusätzlich werden deren Kosten in geschätzten Komplexitätspunkten bewertet.

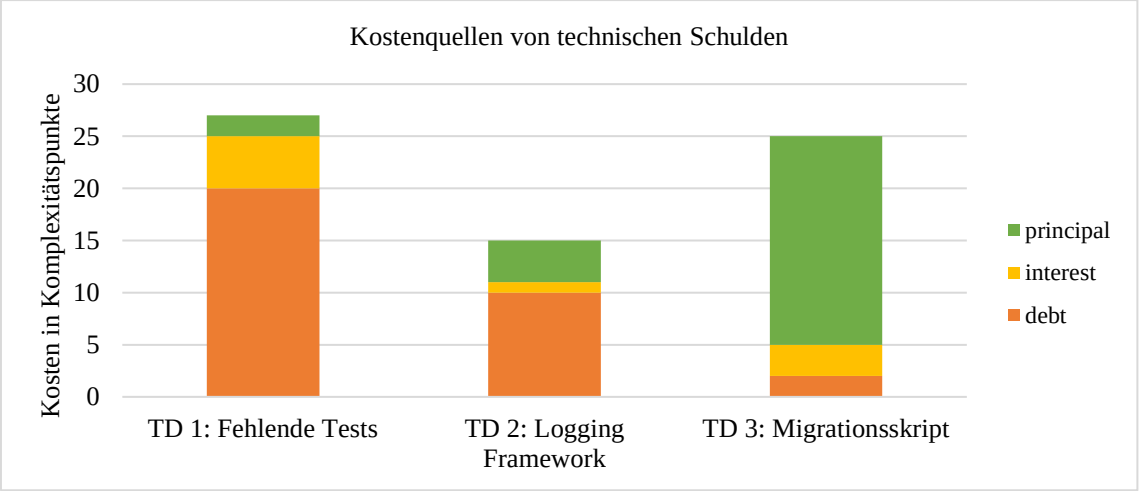

Figure 2: Kostenquellen von TD

TD1 enthält eine hohe Schuld und generiert bedeutsame Zinsen, zugleich ist die optimale Lösung leicht umsetzbar. **TD1** spiegelt das Beispiel der fehlenden Testautomatisierung wider. Derartige Schuldenverhältnisse sollten vermieden werden. **TD2** repräsentiert eine mittlere technische Schuld mit niedrigen Zinsen und vertretbaren Lösungskosten. Die Eigenentwicklung eines Logging-Frameworks entspricht einer solchen Schuld; unwissend wurde eine Lösung entwickelt, welche durch das Hinzufügen der Abhängigkeit zu einem Logging-Framework hätte vermieden werden können. Resultierende Anpassungen am eigenen Logging-Framework werden als unnötige Kosten (Zinsen) erachtet. **TD3** stellt eine niedrige Schuld mit vertretbaren Zinsen dar, aber mit einer teuren optimalen Lösung. Dementsprechende Konstellationen an Kosten könnten ein einmaliges Datenmigrationsskript oder Programm sein. Das

Programm liest z. B. Daten aus einer Datei ein und schreibt diese in eine Datenbank (DB). Die Unterstützung unterschiedlicher Dateitypen wie *.pdf*, *.csv* oder. *.json*; die Unterstützung unterschiedliche Trennzeichen wie , / - _ oder |; eine grafische Unterstützung oder den Support von allen gängigen DBs, sind Anforderungen, welche die Flexibilität und Qualität der Lösung steigern, aber nicht benötigt werden. Nach der Migration der *.txt* Datei in die MySQL DB wird das Skript nicht mehr benötigt. Die Inkaufnahme der TD hilft dabei mit vertretbaren Zinsen eine schnelle Lösung zu erschaffen . Im Idealfall wird das einmalige Skript nach der Migration gelöscht und es entstehen keine Zinsen.

Die Kosten für bewusste getroffene TDs können in Kauf genommen werden, um Meilensteine schneller zu erreichen oder wenn die Lösungskosten kostspielig sind (Lilienthal, 2019) (Wolff, 2021).

Ergänzend zu den bewussten und versehentlichen Entscheidungen, welche TD charakterisieren, erweitert Martin Fowler[8] in seinem „Technical Debt Quadrant“ (*Figure 3*) die Eigenschaften von TD in umsichtig (prudent) und rücksichtlos (reckless).

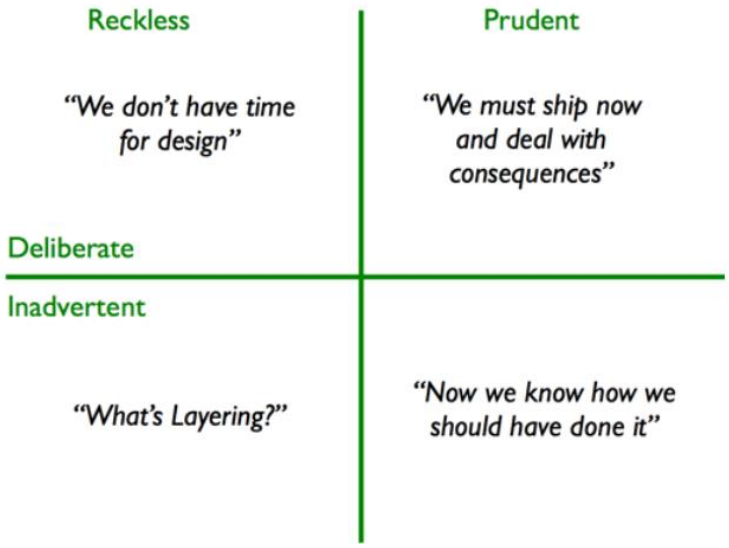

Figure 3: TechnicalDebtQuadrant

Quelle: https://martinfowler.com/bliki/TechnicalDebtQuadrant.html

[8] https://martinfowler.com/bliki/TechnicalDebtQuadrant.html

Das Management der Schuld ist notwendig, um eine Überhand der Schulden und somit der Zinsen zu vermeiden. Ein Überblick, welche Folgen durch Missmanagement der Schulden entstehen, wird im Abschnitt 2.2.1 gegeben.

2.2 Softwarearchitektur

Der Begriff „*Softwarearchitektur*“ wurde erstmals in der *NATO Software Engineering Conference* von 1969 in Rom[9] verwendet (NATO SCIENCE COMMITTEE, 1969). Trotz der Langlebigkeit des Begriffes besteht kein Konsens über eine eindeutige Definition, sondern ein abstraktes Konzept und eine Vielzahl von Definitionen, welche vereinzelte Aspekte der Architektur hervorhebt.

Im Rahmen der vorliegenden Arbeit wird der Begriff Softwarearchitektur durch folgende drei Aussagen geprägt:

*Definition 1: „*Softwarearchitektur ist die Summe aller wichtigen Entscheidungen, die im Verlauf der weiteren Entwicklung nur schwer zu ändern sind“ (Kruchten, 2004).

Definition 2: „Die grundsätzliche Organisation eines Systems, verkörpert durch dessen Komponenten, deren Beziehungen zueinander und zur Umgebung sowie Prinzipien, die für seinen Entwurf und seine Evolution gelten“ (Starke, 2020, p. 16). Die Definition entspricht einer Übersetzung durch Gernot Starke der Definition im IEEE-Standards 1471[10].

Definition 3: „Softwarearchitektur ist die Summer der schwierigen Entscheidungen hinsichtlich des Entwurfs von Software. Es sind jene Entscheidungen, deren Risiko besonders hoch ist. Entweder, weil sie die Qualität der Software maßgeblich beeinflussen oder weil sie sich nur unter hohem Zeit- oder Geldaufwand korrigieren lassen“ (Hirschmeier, 2021, p. 2). Die Definition entspricht einer Übersetzung durch Sebastian Hirschmeier einer Definition von Eoin Woods.

[9] http://homepages.cs.ncl.ac.uk/brian.randell/NATO/

[10] http://www.iso-architecture.org/ieee-1471/defining-architecture.html

Das Abstraktionsniveau der jeweiligen Definitionen weist einen klaren Unterschied auf. Der Fokus der ersten Definition liegt auf den bewussten oder unbewussten getroffenen Entscheidungen, welche die Entwickler und Architekten für das System erarbeitet haben. Die zweite Definition konkretisiert eine Reihe von Entscheidungen: Abhängigkeiten, Integration im System, Patterns, der Schnitt der Komponenten und weiteres. Hinzufügend betrachtet die Aussage die Struktur des Softwaresystem. Abschließend findet bei der dritten Definition der Versuch statt, eine Brücke zwischen den vorherigen aufzubauen. In der vorliegenden Arbeit wird der Schwerpunkt auf die getroffenen Entscheidungen sowie deren Findungsprozess gesetzt. Zusätzlich ist die Betrachtung der Prinzipien, Rahmenbedingungen und Strukturen auf Mikro- und Makroebene ein Schwerpunkt. Erarbeitete Softwarearchitekturen oder Prinzipien im Verlauf dieser Arbeit sind als Leitlinien zu verstehen und empfehlenden Wegweiser, welche anhand entsprechender Rahmenbedingungen nicht für alle Softwarearchitekturen geeignet sein können. Explizite „Muss"-Regeln werden jeweilig gekennzeichnet (MRe).

2.2.1 Langlebige Softwarearchitektur

Erhebliche Investitionskosten fallen für die Neuentwicklung von Individualsoftware an. Dies bestärkt die Hemmschwelle alte Software zu entsorgen und eine Neuentwicklung anzustoßen. Das Resultat der Hemmschwelle ist ein langer Software-Lebenszyklus. Die Lebensdauer von Systemen kann 40-50 Jahre erreichen; Beispiel sind alte COBOL-Programme, welche weiterhin stark im Finanzsektor vertreten sind[11]. Die immense Anzahl an Änderungsanforderungen im Zeitverlauf von 40 Jahre erhöht die Wahrscheinlichkeit eine hohe Anzahl an TD zu akkumulieren. Das Diagramm in *Figure 3* repräsentiert die unkontrollierte zeitabhängige Steigung an TD eines Softwaresystems bis zum Erreichen des roten kritischen Bereiches, in welchen jede Änderung hohe bis unplanbare Kosten verursacht und einen Entwicklungs-Starrzustand auslöst.

[11] https://www.it-finanzmagazin.de/cobol-als-wichtigste-programmiersprache-111380/

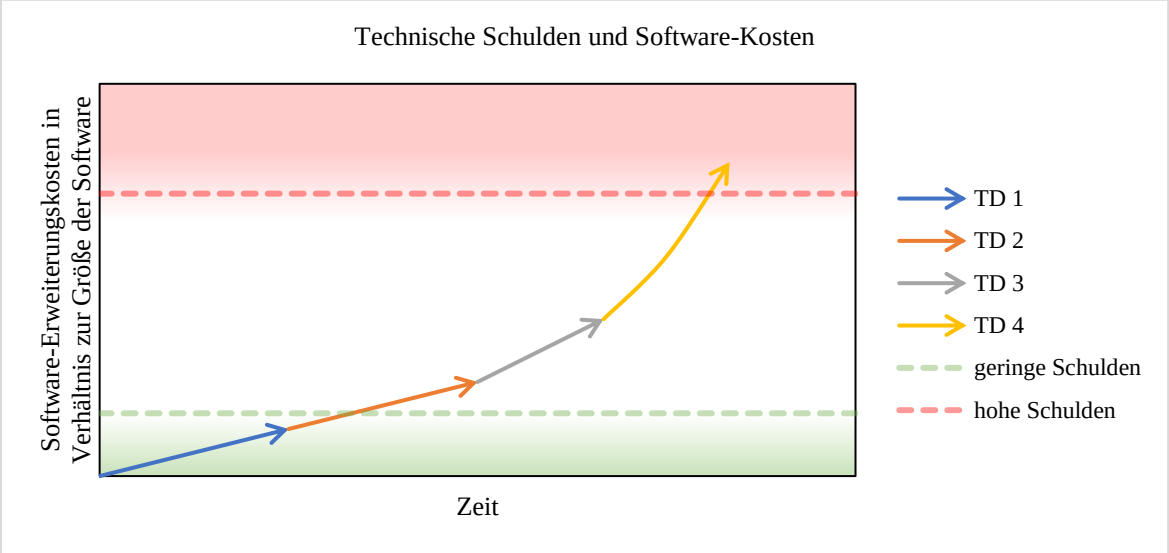

Figure 4: Technische Schulden und Software-Kosten

Das Hauptziel langlebiger Softwarearchitekturen sind kontinuierliche Wartungs- und Erweiterungskosten (Lilienthal, 2019). Die stetige Erosion der Softwarearchitektur benötigt einen iterativen TD-Managementzyklus (MRe). Ziel des Lebenszyklus ist die Überwachung von TDs und deren Abbau.

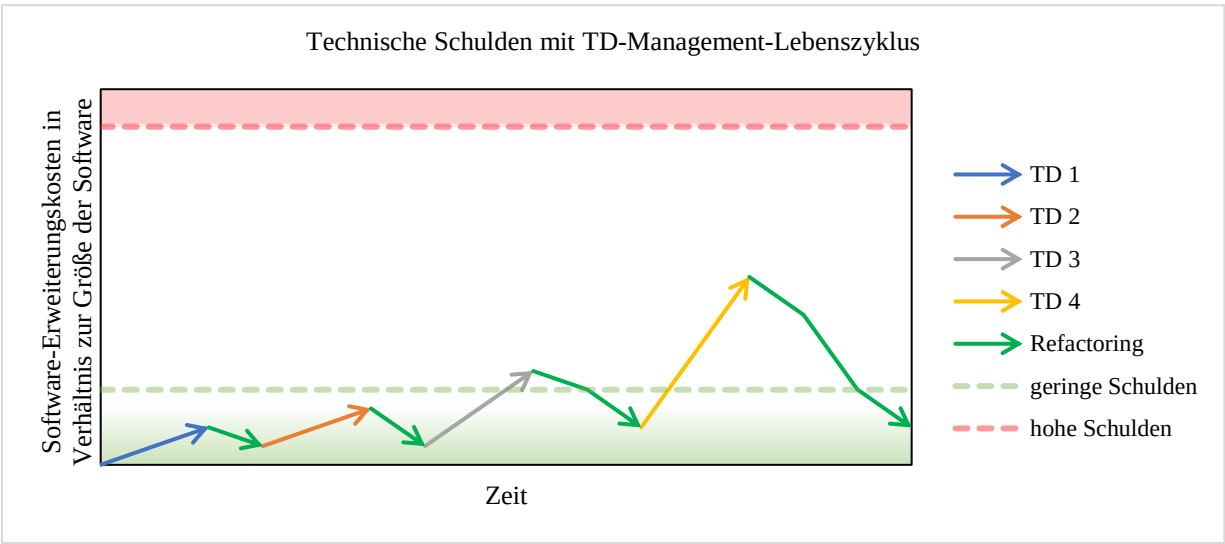

Figure 5: TD mit TD-Management-Lebenszyklus

Die zeitliche Entwicklung der TDs im Diagramm *Figure 5* zeigt ein Software-System mit einem TD-Management-Lebenszyklus. Die entstandenen Schulden werden erkannt und anhand von geplanten Refactorings abgebaut. Das System tendiert zu einer hohen Qualität und befindet sich überwiegend im grünen Bereich, welches vertretbare Wartungs- und Erweiterungskosten aufzeigt. Eine Akkumulation der Schulden bis in den roten Bereich wird vermieden. Das System kann länger „überleben".

Kognitive Psychologie

„*Das Ziel langlebiger Architektur ist, […] lange Änderungszeiten zu vermeiden. Das Softwareentwicklungsteam soll sich bei seiner Arbeit im Sourcecode und in dessen Strukturen schnell zurechtfinden können.*" (Lilienthal, 2019, p. 69)

Die Kognitive Psychologie bietet ein Verständnis über die Art und Weise, wie die Menschen aus ihrer Umgebung Wissen und Erkenntnisse sammeln. Die Berücksichtigung vorhandener strukturbildender Prozesse der Kognitive Psychologie, unterstützen bei der erfolgreichen Erstellung einer langlebigen Softwarearchitektur (Lilienthal, 2019). Das schnelle Zurechtfinden in der Software wird durch drei Prozesse gestützt: *Chunking*, der *Aufbau von Schemata* und die *Bildung von Hierarchien.* (Lilienthal, 2019)

- Chunking[12]: Iteratives Gruppieren und Auswählen von zusammenhängenden Wissenseinheiten (*Chunks*) in höherwertigen Abstraktionen, welche beim *Recoding*[13] verdichtet werden und als neue Einheit abgespeichert wird.
- Aufbau von Schemata: „*Unter einem Schema werden Wissenseinheiten verstanden, die aus einer Kombination von abstraktem und konkretem Wissen bestehen. Schemata enthalten auf der abstrakten Ebene typische Eigenschaften der von ihnen schematisch abgebildeten Zusammenhänge und legen für diese Eigenschaft wertebereiche fest. Auf der konkreten Ebene beinhaltet ein Schema eine Reihe von Exemplaren, die prototypische Ausprägungen des Schemas darstellen."* (Lilienthal, 2019, p. 78)

[12] https://www.britannica.com/topic/mnemonic#ref1083377

[13] https://www.britannica.com/biography/George-A-Miller#ref1200617

- Bildung von Hierarchien: Die Bildung hierarchischer Strukturen aus Elementen mit unterschiedlichen Relationen, fördert das Aufnehmen, Wiedergeben und Zurechtfinden von Wissen. Die Anordnung der Elemente erfolgt unter der Verwendung unterschiedlicher Relationen: Teil-Ganzes-Beziehung, kategoriale Unterordnung oder Verwendungsbeziehung (Lilienthal, 2019) (Anderson, 1980).

Komplexität

„*Die Komplexität eines Softwaresystems speist sich aus zwei verschiedenen Quellen: dem Anwendungsproblem, für das das Softwaresystem gebaut wurde, und der Lösung aus Programmtext Datenbank usw. […] Man spricht hier von der probleminhärenten und der lösungsabhängigen Komplexität.*“ (Lilienthal, 2019, p. 9)

Die Betrachtung der Komplexität in zwei Kategorien, *Probleminhärent* und *Lösungsabhängig*, sowie in zwei Stufen, *essenziell* und *akzidentell*, ermöglicht die Analyse überflüssiger Komplexität. Ein ideales System tendiert zu einer Komplexität, wo *Probleminhärent = Lösungsabhängig* ist und keine akzidentelle Komplexität vertreten ist.

	Essenziell (unvermeidlich)	**Akzidentell (überflüssig)**
Probleminhärent	Komplexität der Fachdomäne.	Missverständnisse über die Fachdomäne.
Lösungsabhängig	Komplexität der Technologie und der Architektur	Missverständnisse über die Technologie. Überflüssige Lösungsanteile.

Tabelle 1: Komplexität

Die Verwendung eines TD-Management-Lebenszyklus, die Vermeidung akzidenteller lösungsabhängiger Komplexität und die Berücksichtigung der kognitiven Psychologie stützen die Entstehung einer langlebigen Architektur mit reduzierten Wartungs- und Erweiterungskosten.

2.2.2 Evolutionäre Architektur

Evolutionäre Architektur bedient sich der Metapher der Evolutionstheorie von Charles Darwin (Darwin, 2008). Architekturen sind, wie die Arten in Darwin‘s Werk, konstanten Änderungen,

technischer sowie fachlicher Natur, ausgesetzt. Am besten angepasste Architekturen überleben häufiger (Survival of the Fittest). Evolutionäre Architektur wird wie folgt definiert: „*An evolutionary architecture supports guided, incremental change across multiple dimension*“ (Neal Ford, Rebecca Parsons, Patrick Kua, 2017, p. 15) und besteht aus drei Hauptaspekten (Neal Ford, Rebecca Parsons, Patrick Kua, 2017, p. 14):

Inkrementelle Änderungen: Diese Eigenschaft betont die Kritikalität von Änderungen über einen längeren Zeitraum und die Implikation einer kontinuierlichen wechselnden Umgebung.

Angemessene Kopplung: Fokussierung auf die Identifikation, welche Bereiche der Architektur aneinandergekoppelt sein sollten, um eine Min-Max-Optimierung zwischen Kosten und Nutzen zu erreichen.

Fitness Funktion[14] *(FF):* Objektive integrale Bewertungsmethode der Evolution einer oder mehrerer der wichtigsten Architektureigenschaft(en).

Zusätzlich zu den vorherigen drei Hauptaspekten von evolutionären Architekturen, ist die Erwähnung von „*multiple dimension*“ bedeutsam. Unterschiedliche Betrachtungspunkte werden als Dimensionen definiert. In Folge besitz eine Architektur mehre Dimensionen, zum Beispiel eine technische Dimension, welche Frameworks oder Abhängigkeiten begutachtet. Anderseits kann eine Architektur über eine datenzentrierte Dimension verfügen, welche als Augenmerk die DB Schemata, Tabellen Layouts oder Datenzugriffsoptimierung besitzt (Neal Ford, Rebecca Parsons, Patrick Kua, 2017). Jede Dimension kann eine oder mehrere FFen besitzen und entspricht eine architektonische Eigenschaft. Architekturen ohne FFen bergen folgende Risiken (Neal Ford, Rebecca Parsons, Patrick Kua, 2017):

- Das Treffen von Entscheidungen, welche in letzter Instanz zu einer Software führt, welche in ihrer Umwelt nicht besteht.
- Die Umsetzung von Entscheidungen, welche Ressourcen kosten und keinen unternehmerischen Mehrwert schöpfen.

[14] https://www.cs.bham.ac.uk/~mmk/Teaching/AI/l5.html

- Die Verhinderung der leichten Weiterentwicklung des Systems bei zukünftigen Umweltänderungen.

Die gezielte Weiterentwicklung der Architektur, gesteuert durch Fitness-Funktionen, hat eine am besten weiterentwickelte Architektur zur Folge. Architekturen werden bei jeder Änderung anhand der Fitness-Funktionen gemessen, entstandene negative Abweichungen sind identifizierbar und können adressiert werden (Parsons, 2021). Darauffolgend können unterschiedliche konkurrierenden Ziele und Anforderungen abgewogen werden (Neal Ford, Rebecca Parsons, Patrick Kua, 2017). Mit dem Verlauf der Zeit und der Volatilität der Umgebung verschieben sich die Rahmenbedingungen des Systems und die FFen veralten. Passend zur evolutionären Architektur, müssen die FFen in einem iterativen Review-Prozess bewertet und erneuert werden (Neal Ford, Rebecca Parsons, Patrick Kua, 2017).

Diese Arbeit lehnt sich an die Kategorisierung von FFen, welche im Werk „*Building Evolutionary Architectures*“ definiert werden (Neal Ford, Rebecca Parsons, Patrick Kua, 2017, pp. 18-21):

- Atomic: Überprüfung eines einzelnen Kontextes und Dimension (Beispiel: Unittest).
- Hoistic: Verifizierung mehrerer interagierenden Kontexte und unter Betrachtung mehrere Dimensionen (Beispiel: End-to-End Tests, welche Performance und Security misst.).
- Triggered: Event gesteuerte Ausführung (Beispiel: Tests in einer CD-Pipeline.).
- Continual: Konstante Prüfung gewünschter Dimensionen (Beispiel: Monitoring der Transaktionsgeschwindigkeit des Systems).
- Static: Funktion mit einem festen erwarteten Ergebnis (Beispiel: Unittest bestanden/gescheitert).
- Dynamic: Das erwartete Ergebnis der Funktion wird anhand des Kontextes bestimmt. Als Beispiel dient das Monitoring der Transaktionsgeschwindigkeit. Abhängig von der Skalierung können unterschiedliche Zeiten akzeptabel sein.
- Automated: Automatisierte Prüfung (Beispiel: Unittest).
- Manual: Manuelle durchzuführende Prüfung (Beispiel: Einhaltung gesetzlicher Vorgaben).

- Temporal: FF mit einer zeitabhängigen Komponente (Beispiel: monatliche Überprüfung externer Abhängigkeiten).
- Intentional Over Emergent: „*[…] some fitness functions will emerge during development of the system. Architects never know all important parts of the architecture at the beginning […] and thus must identify fitness functions as the system evolves.*“
- Domain-specific: Kapselung der fachlichen Treiber als FF. Überprüfung der wesentlichen Eigenschaften der Architektur, um eine Abweichung zu den wesentlichen fachlichen Anforderungen zu vermeiden. Beispielhaft wäre ein Security-Check, durch die eigene Ausführung von DDoS-Angriffen im Bereich der kritischen Infrastruktur.

2.3 Microservices-Architektur

In den Abschnitten „*Langlebige Architektur“* (Abschn. 2.2.1) und *„Evolutionäre Architektur“* (Abschn. 2.2.2) werden Eigenschaften von Softwarearchitekturen beschrieben, welche Konzepte oder Bedingungen das System besitzt. Im Gegensatz beschreibt die Microservice-System-Architektur einen Ansatz, wie ein System technisch umgesetzt wird. Hierbei können gewisse Architektureigenschaften, wie Langlebigkeit, Resilienz oder Weiterentwicklungsfähigkeit, erreicht werden. Microservices-Systeme stehen im Mittelpunkt dieser Arbeit; trotz der Wichtigkeit von Microservices findet sich keine endgültige allgemeine Definition. Im weiteren Verlauf der Arbeit bilden die unteren Aussagen den Grundstein von Microservices.

Definition 1: *„In short, the microservice architectural style is an approach to developing a single application as a suite of small services, each running in its own process and communicating with lightweight mechanisms, often an http resource API. These services are built around business capabilities and independently deployable by fully automated deployment machinery. There is a bare minimum of centralized management of these services, which may be written in different programming languages and use different data storage technologies.“* (James Lewis and Martin Fowler, 2014)

Definition 2: *„Microservices sind unabhängige deploybare Module.“* (Wolff, 2018, p. 3)

Definition 3: *„Microservice sind unabhängig deploybare Services, die rund um eine Businessdomäne modelliert wurden. Sie kommunizieren untereinander über das Netzwerk […]. Damit*

basiert eine Microservice-Architektur auf vielen zusammenarbeitenden Microservices.“ (Newman, 2021, p. 1)

Alle Aussagen treffen den Konsens der Modularität und Unabhängigkeit von Microservices. In *Figure 6* wird anhand des Beispiels von *Uber*[15] eine kompakte Microservices-Architektur beleuchtet.

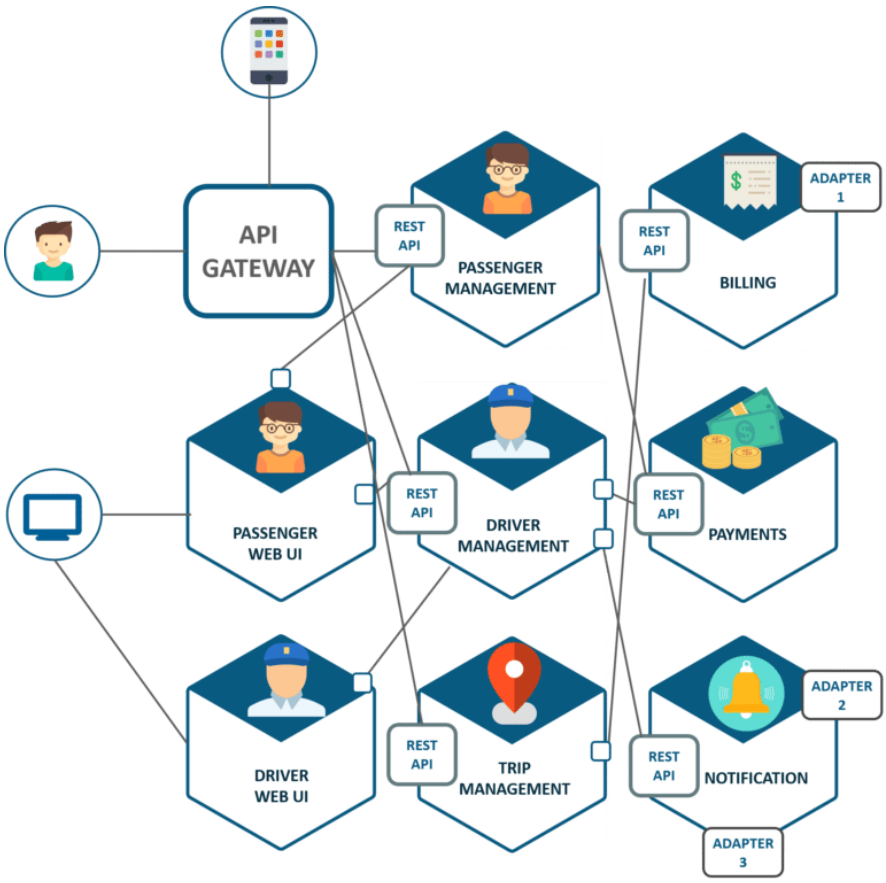

Figure 6: Microservices-Architektur von Uber

Quelle: https://dzone.com/articles/microservice-architecture-learn-build-and-deploy-a

[15] https://www.uber.com/de/de/

Die Autoren fügen der kurzen Beschreibung von Microservices eine Liste an Guidelines, bestmöglichen Maßnahmen und kritische Eigenschaften an, welche einen starken Einfluss auf den Erfolg einer Microservices-Architektur ausüben.[16]

- Microservices sind unabhängige Module, die Zugriffe über APIs bereitstellen. Zugriffe auf die Interna eines anderen Moduls sind verboten und Microservices basieren auf den Prinzipien der Modularisierung von Systemen und SOLID[17]. (MRe)
- Jedes einzelne Modul muss als eigenständiger separater ausführbarer Prozess implementiert werden, welches durch Containerisierung[18] oder virtuelle Maschinen unterstützt wird. (MRe)
- Das System beinhaltet zwei separate Architekturebenen (MRe):
    - Die Makroebene definiert die Richtlinien der Architektur, die einheitlich für alle Microservices gelten; das System wird als Ganzes betrachtet. Die Einschränkung der Kommunikationsprotokolle ist ein Beispiel für eine Entscheidung auf Makroebene (Rest, SOAP[19], RabbitMQ[20], KAFKA[21], etc.).
    - Die Mikroebene beleuchtet die einzelnen Architekturentscheidungen, welche separat für jedes Modul getroffen werden können. Beispielhaft ist die Entscheidung der verwendeten Datenbank (MySQL, MongoDB, Oracle, etc.).
- Der Zuwachs an Netzwerkkommutation zwischen den Services führt zu einer Erhöhung der End-to-End Latenz und der Daten Translation. Leichtgewichtige Kommunikation zwischen den Modulen ist erforderlich. (MRe)
- Die Standardisierung von Metadaten ist notwendig, um die erneute Abfrage bei jedem einzelnen Microservice zu verhindern. Ein Beispiel ist die Authentifikation des

[16] (James Lewis and Martin Fowler, 2014) (Newman, 2015) (Richardson, 2018) (Wolff, 2018) (Wolff, 2018) (INNOQ, kein Datum)

[17] http://butunclebob.com/ArticleS.UncleBob.PrinciplesOfOod

[18] https://www.docker.com/resources/what-container/

[19] https://www.redhat.com/de/topics/integration/whats-the-difference-between-soap-rest

[20] https://www.rabbitmq.com/

[21] https://kafka.apache.org/

Benutzers; ein Anmelde-Popup pro Microservice ist nicht tragbar und schadet der Akzeptanz des Systems. (MRe)

- Der Betrieb sollte übergreifend standardisiert sein, was Konfiguration, Deployment, Log-Analyse, Tracing, Monitoring und Alarme einschließt.
- Standardisierung in der Makroebene sollte auf Interface-Niveau stattfinden. Die Definition von REST als Kommunikationsprotokoll könnte erfolgen, aber die Einschränkung auf die verwendete Bibliothek sollte vermieden werden (Spark[22], Django[23], RestSharp[24], etc.).
- Die Unabhängigkeit der Module erfolgt nicht ausschließlich auf Source-Code-Ebene, sondern muss auch im gesamten Delivery-Prozess berücksichtigt werden. Einzelne separate Continuous Deployment (CD) Pipelines müssen eingerichtet werden. (MRe)
- Das Hauptkriterium der Resilienz für verteilte Systemen (Maarten van Steen, Andrew S. Tanenbaum, 2018) ist für die Microservices-Architektur ebenfalls notwendig. Einzelne Module müssen den Ausfall oder Kommunikationsprobleme anderer Modulen abfangen.
- Die Unabhängigkeit der automatisierten Tests zwischen Microservices muss gewährleistet sein, weil die Ausführung Bestandteil der individuellen CD-Pipeline ist.
- Das Potenzial von kurzen Release-Zyklen, von einigen Stunden bis zu wenigen Tagen, ist ein Hauptmerkmal von Microservices-Architekturen. Die Schöpfung dieses Potenzials erfordert agile Vorgehensmodelle. Lange Planungen wie im Wasserfallmodell, würden den Release um Monate verschieben.
- Domain Driven Design (DDD) zerteilt die Entwicklung der Funktionalitäten des gesamten Systems in fachliche Kontexte mit Hilfe von Strategic Design[25]. Diese Methodik ermöglicht das Schneiden grobgranularer Module, welche den Ansatz der Modularität von Microservice widerspiegeln (Vernon, 2017) (Wolff, 2018).

[22] http://sparkjava.com/

[23] https://www.djangoproject.com/

[24] https://restsharp.dev/

[25] https://www.innoq.com/de/articles/2021/01/domain-driven-design-bounded-context/

3 Angemessene Architektur

Die Entwicklung einer „guten“ Softwarearchitektur und deren Beibehaltung ist das Ziel eines jeden/r Softwarearchitekten/-innen. Die Problematik entsteht bei der Definition von „gut“. Wie kann eine abstrakte Eigenschaft erfasst und bewertet werden, wenn die Umgebung konstanten Änderungen unterliegt und jedes Einsatzszenario einzigartig ist?

Im Abschnitt 2.2 wurden Softwarearchitekturen als Leitlinien definiert. Gründe für die vage Definition sind die unterschiedlichen Umgebungen und Rahmenbedingungen, in denen die Architektur entsteht. Beispielhafte Umgebungen wären unter anderem Web-Systeme, Desktop-Systeme, Embedded-Systeme, Anbindungen an Legacy Systeme sowie regionale oder landesweite Gesetzgebungen. Alle Systemarten besitzen unterschiedliche Rahmenbedingungen, welche zu anderen Paradigmen führen. Martin Fowler definiert eine gute Architektur wie folgt: „*But I resolve my concern by emphasizing that good architecture is something that supports its own evolution, and is deeply intertwined with programming.*“ (Fowler, 2019). Eine andere Sichtweise auf die Arbeit von Architekten/-innen und „*gute“* Architektur bietet das Buch „Software Architecture: The Hard Parts“, welches das Erreichen einer guter Architektur als falschen Ansatzpunkt definiert und sich auf die Vermeidung der gravierendsten trade-offs konzentriert: *„[…], the real job of an architect lies in their ability to objectively determine and assess the set of trade-offs on either side of a consequential decision to resolve it as well as possible*“ und „*Often the best design an architect can create is the least worst collection of trade-off […]*“ (Neal Ford & Mark Richards, 2021, p. 2). Demzufolge versuchen alle Architekturen eine hohe Kohärenz der einzelnen Module zu erlangen, die Anzahl von TD zu reduzieren und ein stabiles System zu ermöglichen, das die Anforderungen erfüllt und sich, dank einer angemessenen Kopplung, an einer dauerhaften wechselhaften Umgebung anpasst.

Die dargestellten Eigenschaften der langlebigen Architektur (2.2.1) und evolutionären Architektur (2.2.2) verfolgen die Erreichbarkeit der Ziele, die im letzten Satz des vorherigen Absatzes erwähnt wurden. Langlebige Architekturen fokussieren sich auf die Reduzierung der Anzahl von TD und der Steigung der Kohärenz des Systems, welches die Stabilität und die Erweiterbarkeit des Systems positiv beeinflussen (Lilienthal, 2019). Zusätzlich betrachtet eine

evolutionäre Architektur die angemessene Kopplung zwischen den Systembestandteile, das Schützen der Kerneigenschaften eines Systems (Erfüllung der Anforderungen) durch FF sowie die Anpassungsfähigkeit des Systems.

Constantines Law: „*Eine Struktur ist stabil, wenn die Kohäsion stark und die Kopplung schwach ist.*“ (Newman, 2021, p. 17)

Im weiteren Verlauf des Kapitels werden Konzepte und Methodiken präsentiert, um Annäherungen an eine bestmögliche langlebige und evolutionäre Architektur umzusetzen.

3.1 Modularisierung

Das erste Ziel einer angemessenen Architektur ist die Verständlichkeit des Systems zu erhöhen. Dabei stützt sich eine langlebige Architektur auf die Erkenntnisse der kognitiven Psychologie und stellt die Modularisierung in den Vordergrund. Kleinere, in sich logisch abgeschlossene Konstrukte, helfen bei der Bildung von Hierarchien und Wissenseinheiten (Lilienthal, 2019). Der positive Einfluss der Modularisierung auf die Verständlichkeit, die Reduzierung der Wartungskosten und ein kürzerer TTM, sind Grundsteine der Informatik, welche unter anderem D. L. Parnas in seinen wissenschaftlichen Artikeln analysierte (Parnas, 1972) (Parnas, 1972). Daraus folgt, dass eine schlechte Modularisierung, im Vergleich zu einer gelungenen, den Softwareentwicklungsprozess verlangsamt und somit als TD betrachtet wird.

Die evolutionäre Architektur erweitert den Fokus der Modularisierung um die Interaktion zwischen den Modulen, indem die Kopplung in Betrachtung gezogen wird. Die gewonnenen Erkenntnisse zeigen, wie wichtig der „richtige“ Schnitt von Modulen ist. Zu große Module sind in sich nicht mehr logisch abgeschlossen, während zu kleinen Modulen höhere Komplexität zwischen den Bestandteilen generieren und die Softwarekosten erhöhen.

Um die Grenzen eines Moduls zu erkennen, wird in dieser Arbeit auf folgende Definition zurückgegriffen: „*Here we define a component as a building block of the application that has a well-defined role and responsibility in the system and a well-defined set of operations.*“ (Neal Ford & Mark Richards, 2021, p. 71). In der Definition wird das Wort Komponente verwendet; in diesem Rahmen wird die Aussage auf Module extrapoliert und stellt in sich einen

abgeschlossenen logischen Block dar. Module können selbst aus mehreren Modulen bestehen und bilden eine Analogie zu den Chunks in der kognitiven Psychologie.

Die Findung der „richtigen“ Grenzen eines Moduls, auf architektonischer Ebene, wird durch die Betrachtung diverser nicht funktionaler Dimensionen beeinflusst. Muss einer der Einflussfaktoren erfüllt werden, ist eine Zerteilung des Moduls ratsam.

- Wirtschaftlichkeit: Die Separierung in kleinere Module können die Komplexität des gesamten Systems steigern. Zusätzlich erfordert jede neue Komponente mehr organisatorischen Aufwand. Überwiegen die finanziellen Nachteile den Vorteilen, sollte keine Zerteilung stattfinden.
- Ausfallsicherheit (engl.: Availability): Die Fähigkeit der Erreichbarkeit einzelner Bestandteile des Systems, nach dem einzelne oder mehrere andere Bereiche, durch gravierende Fehler, nicht mehr in der Lage sind zu reagieren. Mittels der Zerlegung des Systems in einzelne einsetzbare Einheiten, kann ein kaskadierendes Versagen auf die einzelne Einheit beschränkt werden (Neal Ford & Mark Richards, 2021, pp. 58-59).
- Skalierung (engl.: Scalability): Die Eigenschaft die Funktionalität aufrecht zu erhalten, beim konstanten steigenden Verbrauch von Ressourcen (Neal Ford & Mark Richards, 2021, p. 56). Die Lastverteilung auf einzelne Module, ermöglicht die redundante Verfügbarkeit mehrere Instanzen der am stärksten belasteten Module. Demzufolge reduzieren sich die operativen Kosten durch die Bereitstellung kleinerer Module, anstatt die Bereitstellung eines großen Konglomerats mit der gesamten Funktionalität der Software.
- Geringere TTM (engl.: Agility): Die kürzeren Zyklen der Gestaltung marktreifer Features kann in drei Kategorien untergliedert werden:
    - Deploybarkeit (engl.: Deployability): Die Auffassung der Herausforderung vom Deployment der Software, wie dessen Frequenz und deren Risikohaftigkeit (Neal Ford & Mark Richards, 2021, p. 55). Kleinere Module können bei einer hohen Automatisierung schneller deployt werden und stellen, durch die minimalere Anzahl an Änderungen, ein kleineres Risiko da.
    - Testbarkeit (engl.: Testability): Die Schlichtheit, um das Softwareprodukt zu testen und die Vollständigkeit der Tests (Neal Ford & Mark Richards, 2021,

p. 54)[oe]. Kleinere Module beinhalten weniger Funktionalitäten, weshalb die Maße sowie die Ausprägungen der Tests geringer ausfallen.

- Wartbarkeit (engl.: Maintainability): Die Betrachtung der Komplexität neue Feature hinzuzufügen, zu ändern oder entfernen sowie Wartungsarbeiten für die Instandhaltung der Infrastruktur, Framework upgrades, third-party-tools upgrades oder die Einspielung von Patches (Neal Ford & Mark Richards, 2021, pp. 50-54). Es besteht eine Korrelation mit dem Einflussfaktor „*Wirtschaftlichkeit*".

- Ach-Ja-Effekt: Werden Funktionalitäten eines Moduls bei Analysen oder Gesprächen oft vergessen, ist dies ein Indikator einer unpassenden Trennung. Die Komponente beinhaltet mehrere Funktionalitäten, die logisch nicht zu einer Wissenseinheit zusammengeführt werden können und sollte daher zergliedert werden.

Die Betrachtung aller Einflussfaktoren ist notwendig, um eine angemessene Kopplung und Modularisierung zu finden. Eine generalisierende Gewichtung der einzelnen Faktoren ist wegen den generischen Lösungen und Einsätzen von Software nicht möglich, wobei die Wirtschaftlichkeit verheerende Folgen für das Unternehmen haben kann.

Data desintegrators vs data integrators

Der Kernbestandteil der Informationssysteme sind die zu verarbeitenden Daten. Softwareprodukte verarbeiten und manipulieren Daten (Informationen), um wirtschaftliche Prozesse zu unterstützen. Die Geschäftsprozesse können sich durch das ganze Unternehmen ziehen, ebenso die notwendigen Daten. Die stärksten Kopplungen zwischen Modulen werden durch die verwendeten Daten erzeugt. Als Beispiel dienen die Informationen des Kunden. Diese sind für die Erbringung der Leistung notwendig oder für die Abrechnung. Besitzt der Kunde einen Vertrag, besteht die Notwendigkeit diese miteinander zu verbinden. Der Versand benötigt die Adresse des Kunden. Das Beispiel kann beliebig erweitert werden. Infolgedessen ist die Modularisierung und die Trennung der Daten der schwierigste Teil bei der Erzeugung von Modulen und deren Kopplungen. Welche Effekte und Herausforderungen im Bereich von Microservices-Systeme entstehen, wird im Kapitel 4 beleuchtet. In diesem wird eine Übersicht der Wechselwirkungen für *Data Desintegrators* und *Data Integrators* gegeben. *Data Desintegrators*

(Tabelle 2) und *Integrators (Tabelle 2)* beschäftigen sich mit den Erwägungsgründe, warum Daten separiert oder vereint werden sollten (Neal Ford & Mark Richards, 2021, p. 132).

Desintegrator

Change Control	Die Fragenstellung, wie viele Module von einer Änderung an einer Datenstruktur betroffen sind.
Connection Management	Die Eigenschaft der Bereitstellung ausreichender Verbindungen für den Datenzugriff durch Softwareprogramme.
Change Scalability	Die Skalierung der Datenquelle, um die Performanceanforderungen der zugreifenden Dienste zu erfüllen.
Change Fault Tolerance	Die Betrachtung des kaskadierenden Versagens von Services beim Ausfall der beobachtende Datenquelle.
Architectual Quanta	Methodik, um stark gekoppelte deploybare Module zu identifizieren. (Kapitel xxxxx)
Database type optimization	Die Verteilung in unterschiedlichen Datenquellenarten für die jeweils optimale Ablage der Daten.

Tabelle 2: Data Desintegratoren

Integrator

Data relationship	Die Beziehungen und Verweise zwischen den Daten. Relationale DB verwenden Views, FKs oder Trigger, um Daten aneinander zu binden und die Integrität dessen zu gewährleisten.

Database transaction	Die Sicherstellung einer kohärenten Verarbeitung in einer logischen Klammer mehrerer Daten. Alle Daten müssen persistiert werden oder keine. Die Erfüllung von ACID[26].

Tabelle 3: Data Integrator

Zusammenfassung

Die Modularisierung ist eines der wichtigsten Konzepte der Informatik und bildet den Grundstein jeder guten Architektur. Die Häufung von Verletzungen der Modularität ist ein starker Indikator für einen mangelnden Architekturprozess.

3.2 TD-Managementlebenszyklus

Im Abschnitt 2.1 wurden die Konsequenzen von TD dargestellt, welche die Reduzierung der Wartbarkeit und eine Erhöhung der Erweiterungs- und Anpassungskosten der Software ergeben. Eine langlebige Architektur setzt einen TD-Managementzyklus voraus. In *Figure 7* wird ein vereinfachter Zyklus, bestehend aus den Schritten *Erkennung, Dokumentation, Einplanung, Anpassung* und *Monitoring*, dargestellt. Der Zyklus stützt sich auf eine iterative Durchführung; aufgrund der neuauftretenden Erkenntnisse und Erweiterungen des Systems, müssen Informationen neubewertet oder ergänzt werden. Komplexere Software-Verbesserungsmethodiken, wie aim42[27] bieten eine große Anzahl an Werkzeugen, um Software zu analysieren und evaluieren. Im ersten Schritt ist die Klarstellung des groben Konzeptes eines TD-Managementzyklus zielführend.

[26] https://www.ibm.com/docs/en/cics-ts/5.4?topic=processing-acid-properties-transactions

[27] https://www.aim42.org/

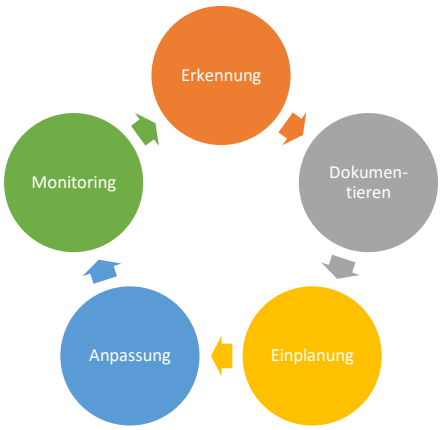

Figure 7: TD-Managementzyklus

Erkennung und Dokumentation

Als Einstiegsschritt empfiehlt sich die Erkennung und Sammlung von TD, konkrete Methodiken werden in den Abschnitten 3.2.1, 3.2.2 und 3.3 besprochen. Folgend zur Erkennung müssen TD dokumentiert werden. Stützende Tools, wie zum Beispiel das statische Code-Analyse-Tool SonarQube[28] , protokollieren zeitgleich die Funde. Werden Tools und Methodiken verwendet, welche keine automatisierte Dokumentation unterstützen, ist eine manuelle Dokumentation notwendig. Im Abschnitt „Architektonische Quant“ (5.5.2) wird eine solche manuelle Methodik verwendet.

Die Dokumentation kann in unterschiedlichen Tools erfolgen. Zusätzlich zum Board von SonarQube, können Excellisten, Tickets in Kanban-Boards oder ADRs in einem Wiki, TD darstellen. Wichtig bei der Vielfalt sind etablierte und abgestimmte Prozesse zwischen den betroffenen Akteuren. Ein Team kann sich auf Tickets im Kanban-Board und das SonarQube-Board einigen. Im Gegensatz dazu legen sich Softwarearchitekten/-innen, welche für die Betrachtung des gesamten Systems zuständig sind, auf ADRs[29] (Architectural Decision Records) in einem Wiki fest. ADRS sind kurze Dokumente, welche die

[28] https://www.sonarqube.org/

[29] https://adr.github.io/

Entscheidungsfindung und die Konsequenzen von architektonischen Entscheidungen dokumentieren. Ein fundamentaler Einfluss bei der Wahl der Dokumentationsmethodik ist der Typ der TD (Abschnitt 3.2.1 und 3.2.3).

Einplanung

Nach Identifizierung der TDs müssen diese gewichtet und deren Lösungen oder Mitigationen eingeplant werden. Der Prozess der Einplanung findet zusätzlich zu den Entwicklungsaufgaben im gleichen Planungsschritt statt. Vorteilhaft ist die Pflege der zu lösenden TD in den „Aufgaben-Katalog" des nächsten Produktmeilensteines[30] aufzunehmen. Entsprechend müssen Aufwände für die Bewältigung von TDs eingeplant und zur Verfügung gestellt werden. Die Einschätzung der Schweregrade von TD können anhand von Risikomatrizen (*Figure 8*) oder Opportunitätsanalyse[31] erfolgen.

Figure 8: Risikomanagementmatrix

Quelle: Markus Harrer, socreatory-Training iSAQB IMPROVE[32]

[30] In einem agilen Vorgehen können die TDs im Backlog hinterlegt werden und bei der Sprint- oder Iterationsplanung des nächsten Zyklus hinzugefügt werden.

[31] https://eltjopoort.nl/blog/2018/07/03/opportunity-cost-in-the-technical-debt-business-case/

[32] Angepasstes Diagramm aus Einführung in die Softwarearchitektur 3: Architekturbewertung von Hendrik Lösch

Anpassung

Das Ziel dieses Schrittes ist der Abbau oder die Minderung von TDs. Bei der Umsetzung der Maßnahmen können weitere TDs entstehen, welche parallel dokumentiert werden. Sind die entstandenen TDs gravierender als das betrachtete zu lösende Problem, kann die Umsetzung gestoppt und die Dokumentation um die gewonnenen Erkenntnisse ergänzt werden. Ein Eingriff aus dem Schritt der *Anpassung* in den anderen Schritten ist möglich.

Monitoring

Analog zum Software- und Architekturmanagement-Prozess wird im TD-Managementzyklus der Stand von TDs überwacht. Wurde eine TD abbezahlt oder ist nicht mehr relevant, zum Beispiel durch das Erreichen des End-of-Life eines Systems, wird die Schuld aus der Dokumentation entfernt. Im negativen Fall können sich die Rahmenbedingungen verändern, welches zu einer Neubewertung oder Neuentstehung von TDs führen kann. Die Erfolge der Maßnahmen können anhand von FFn objektiv gemessen werden und bieten die Möglichkeit Fehlentscheidungen zu identifizieren.

Zusammenfassung

Die Entstehung einer starken Ähnlichkeit zwischen dem TD-Managementzyklus und des Vorgehens im Risikomanagement beleuchtet die Wichtigkeit von TDs. Stefan Toth hebt diesen Aspekt der Softwarearchitektur mit dem folgenden Satz hervor „[…] *und Architektur ist auch so eine Risikodisziplin finde ich, also es geht darum wo sind denn Bedrohungen für unsere Produktvision* […] “ (Toth, 2022). TD müssen als Risiken fürs Unternehmen verstanden werden, wobei Mitigationen oder Beseitigungen erfolgen können (*MRe*).

3.2.1 Ontologie von Technischen Schulden

Die Abarbeitung von TD erfordert zwingend die Erkennung von TD, infolgedessen wird eine stützende Methodik benötigt. Eine erste Annäherung, um TDs zu identifizieren, ist die Erstellung einer Ontologie und deren Kategorisierung. Dadurch können spezifische Merkmale und

Bereiche beobachtet werden. Der weitere Verlauf der Arbeit stützt sich auf den gesammelten Erkenntnissen aus „Towards an Ontology of Terms on Technical Debt“ (N. S. R. Alves, L. F. Ribeiro, V. Caires, T. S. Mendes and R. O. Spínola, 2014). Mitglieder des TD-Researchteams erstellten die in *Figure 9* repräsentierte Ontologie mit der daraus resultierenden Kategorisierung von TD und Indikatoren aus *Tabelle 4.* TD können mehreren Typen gleichzeitig zugeordnet werden: Zum Beispiel kann ein unübersichtliches Programmteil anhand der Komplexität zugleich als „Code debt“ aufgenommen werden, wie als „Documentation debt“ wegen der fehlenden Dokumentation, um das Programm zu verstehen. Zusätzlich können Typen untereinander exkludierend sein.

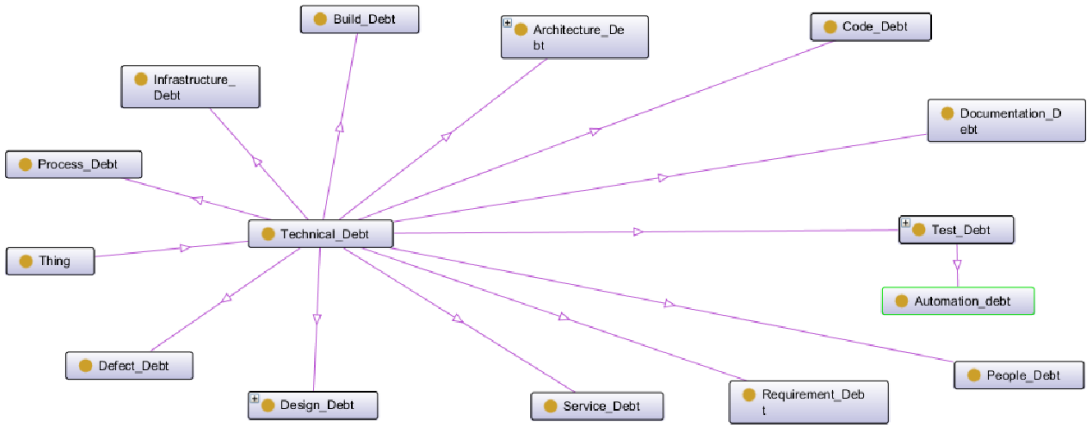

Figure 9: Ontologie technische Schulden (N. S. R. Alves, L. F. Ribeiro, V. Caires, T. S. Mendes and R. O. Spínola, 2014)

Tabelle 4: Kategorisierung von TD und deren Indikatoren.

TD-ID	***TD Typen***	***Indikator***
1	Architecture Debt	ACN/PWDR Betweeness Centrality Issues in software architecture Structural Analysis Structural Dependencies Violation of Modularity

2	Build Debt	„Dead Flags" "Zombie Targets" Dependency Visibility
3	Code Debt	ASA Issues Code Metrics Code outside of standards Duplicated code Multithread correctness (ASA) Slow Algorithm
4	Defect Debt	Uncorrected known defects
5	Design Debt	
6	Documentation Debt	Documentation does not exist Incomplete Design Specification Incomplete Documentation Insufficient comments in code Outdated Documentation Test Documentation
7	Infrastructure Debt	
8	People Debt	
9	Process Debt	
10	Requirement Debt	Requirement Backlog List
11	Service Debt	Selection/Replacement of web service
12	Test Automation Debt	
13	Test Debt	Incomplete Tests Low coverage

Die Entstehung eines Widerspruches zwischen den TechnicalDebtQuadrant (*Figure 3*) von Martin Fowler's und den entstandenen Typen der Ontologie ist nicht vorhanden. Im Gegenteil, es entsteht eine Ergänzung zwischen den jeweiligen Konzepten. Der TechnicalDebtQaudrant differenziert die Motivation der Erstehung der TD, wobei die Ontologie den Bereich der Entstehung definiert. Demzufolge wird das Weshalb? und Wo? einer TD definiert.

3.2.2 Source-Code Analyse

Die Überwachung von Indikatoren von TD kann durch die Analyse von Source-Codes gestützt werden. Die rechtzeitige Erkennung von TD werden durch schnelle Feedbacks begünstig und führen zu deren Vermeidung. Darauffolgend stützen sich langlebige Architekturen auf Code Analysen, wobei das stumpfe Verfolgen von Metriken in Betrachtung von Goodhart's Law „*When a measure becomes a target, it ceases to be a good measure*." schädlich für die Architektur ist.

Die Source-Code Analyse erfolgt in zwei Kategorien: *statisch* oder *dynamisch*. Ersteres wird als „*das Testen und Evaluieren einer Applikation durch die Untersuchung von Source-Code ohne die Ausführung der Applikation*" definiert, wobei dynamische Analysen „*das Testen und Evaluieren einer Applikation zur Laufzeit*" darstellen (CISM, kein Datum).

Die Notwendigkeit beider Vorgehensweisen wird im wissenschaftlichen Artikel „*Comparing four approaches for technical debt identification*" beleuchtet. Hier erlangen die Autoren drei wichtige Erkenntnisse (Zazworka, N., Vetro', A., Izurieta, C., 2013).

- „*Different TD techniques point to different classes and therefore different problems*."
- „*There are few overlaps between the results reported by these techniques*."
- „*Based on quality Goals, decide which Tools and indicators are suitable*."

Die Verwendung mehrerer Techniken und Werkzeuge führt so zu einer höheren Menge erkannter TD, welche Unterschiede in deren Kategorisierung aufweisen. Einen Überblick darüber, welche Methodiken und Tools verwenden werden können, liefern die Unterpunkte „*Statische Analyse*" und „*Dynamische Analyse*". Dabei handelt es sich nicht um eine umfassende Marktanalyse der zur Verfügung stehenden Tools und deren Bewertung, sondern um einen voreingenommenen Einblick in Möglichkeiten. Die Auswahl der dargestellten Werkzeuge erfolgte anhand der bereits vorhandenen Kenntnisse, den Zugriff auf das Tool durch Studien- oder Demolizenzen oder durch eine einfache schnelle Einarbeitung und enthält somit einem Bias.

Statische Analyse

Statische Analysen zeigen ihre Stärke bei der Erkennung von Modularitätsverstößen, Code-Smells[33] und der Verletzungen von Code-Richtlinien[34].

Modularitätsverstöße können manuell oder automatisiert ermittelt werden.

Eine manuelle Analyse erfolgt üblicherweise durch eine/n Architekten/-innen, welche/r die Liste der neu erstandenen Abhängigkeiten zwischen Modulen vor einem Release kontrolliert. Als Beispiel dient die Überprüfung der MANIFEST.MF Datei im Kontext von Java und OSGI[35] Bundles. OSGI bietet die Möglichkeit deklarativ Module (Bundles) und die Zugriffe untereinander zu definieren. Abweichende Abhängigkeiten von der Soll-Architektur können identifiziert werden. Architekten haben die Möglichkeit, bewusst die Soll-Architektur zu modifizieren oder die Änderung abzulehnen und einer TD entgegenzuwirken.

Die automatisierte Analyse kann durch Tools wie ArchUnit[36] oder ArchUnitNet[37] durchgeführt werden. Architektonische Regeln können als Unittest hinterlegt werden. Zum Beispiel wird bei einer Schichtenarchitektur folgende Zugriffsregeln hinterlegt: controller → services → persistence. Verstößt eine Abhängigkeit (controller → persistence) gegen diese Regel, schlägt der Test in *Listing 1* fehl, wie in *Figure 10* zu erkennen ist.

```
@AnalyzeClasses
public class ArchitekturLayerTest {

        @ArchTest
        public static final ArchRule layeredArchitectureCheck =
          layeredArchitecture()
              .consideringAllDependencies()
              .layer("controller").definedBy("..controller..")
              .layer("services").definedBy("..services..")
              .layer("persistence").definedBy("..persistence..")
              .whereLayer("controller")
```

[33] https://t2informatik.de/wissen-kompakt/code-smell/

[34] https://www.triology.de/blog/code-conventions

[35] https://www.osgi.org/

[36] https://www.archunit.org/

[37] https://archunitnet.readthedocs.io/en/latest/

```
                .mayNotBeAccessedByAnyLayer()
            .whereLayer("services")
                .mayOnlyBeAccessedByLayers("controller")
            .whereLayer("persistence")
                .mayOnlyBeAccessedByLayers("services");

}
```

Listing 1: Sichten Verstoß Architektur Test

Quelle: https://github.com/DanielStarkGal/ba_crm

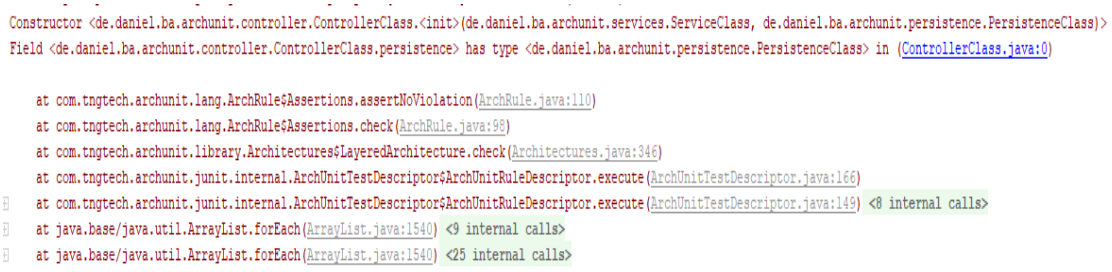

Figure 10: Fehlerhafter Architekturtest

Code Smells sind Indikatoren für nicht gut strukturierte Programmteile, welche die akzidentelle lösungsabhängige Komplexität steigern. Martin Fowler definiert Code Smell wie folgt: „*A code smell is a surface indication that usually corresponds to a deeper problem in the system.* […] *Firstly a smell is by definition something that's quick to spot […] The second is that smells don't always indicate a problem.*" (Fowler, 2006). Als stützende Feststellungsmethodik werden Metriken erhoben und ausgewertet. Die Überprüfung vom Source-Code erfolgt in drei Kontexten: „*Identity Harmony*", „*Collaboration Harmony*" und „*Classification Harmony*" (Michele Lanza, Radu Marinescu, 2006). Einen Überblick schafft *Tabelle 5*.

Identity	***Collaboration***	***Classification***
Wie definiere ich mich selbst?	Wie interagiere ich mit anderen?	Wie definiere ich mich hinsichtlich meiner Vorfahren und Nachkommen?
God Class Brain Class	Intensive Kopplung	Verwendung von Vererbung als Code reuse Pattern.

		Refused Parent Bequest → Die nicht Verwendung protected Variable der Elternklasse.

Tabelle 5: Code Smells Disharmonies mit Beispielen.

Kritisch bei der Erhebung von Code Smells ist die Daten Basis (gute Metriken), die Verwendung mehrere Kontexte sowie die unterschiedlichen Erkennungsstrategien (Zazworka, N., Vetro', A., Izurieta, C., 2013).

Die Analyse der Missklänge im Source-Code kann durch Tools wie Sonarqube, CodeCharta[38] oder CodeQL[39] erfolgen. In *Figure 11* wird eine solche Analyse durch das Tool SonarQube dargestellt.

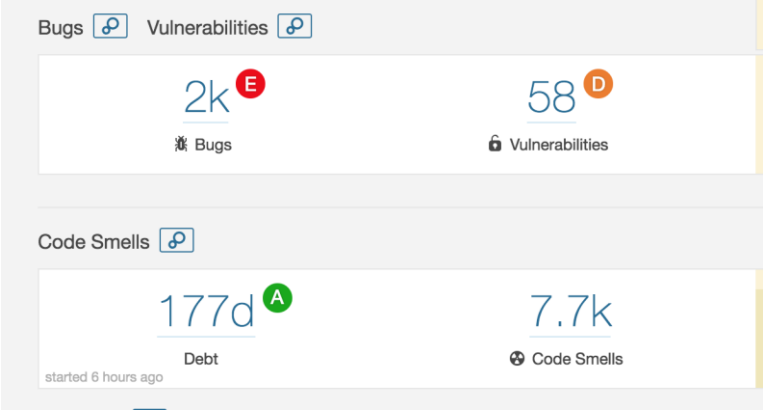

Figure 11: SonarQube Analyse

Quelle: https://medium.com/nicobar-com/sonarcube-nicobar-way-to-look-at-things-does-your-code-smell-bf28aa62cc7c

Dynamische Analyse

Die Überprüfung zur Laufzeit des Source-Codes, bietet die Möglichkeit produktionsnahe Szenarien zu generieren und Annahmen über die Inbetriebnahme des Standes zu treffen. Dabei

[38] https://www.maibornwolff.de/codecharta

[39] https://codeql.github.com/

hilft die Testpyramide (*Figure 12*) die Grundlagen der Testabdeckung darzustellen. Eine hohe Anzahl an Tests korreliert mit der Vermeidung von unerwarteten Fehlern und drücken somit die Wartungskosten der Software.

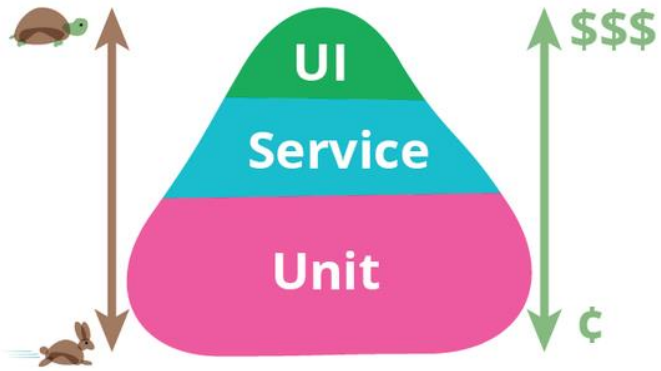

Figure 12: Testpyramide

Quelle: https://martinfowler.com/bliki/TestPyramid.html

Zusammenfassung

Die Verwendung von Source-Code-Analysen Methodiken stützen die Identifikation von TD, wodurch eine langlebige Architektur floriert. Nichtdestotrotz reichen Metriken nicht aus, um eine Architektur zu bewerten, sondern nähren den Boden für den Wachstum einer angemessenen Architektur. Wird nur der Source-Code analysiert, werden auch nur dort Missstände identifiziert. Die Erfüllung von Metriken sollten nicht als das zu erreichende Ziel betrachtet werden, sondern als stützender Mechanismus betrachtet werden.

3.3 Qualitätsfunktionen

Mit voranschreitender Dauer evolviert die Software und erreicht einen bestimmten Reifegrad. Diese Produkte enthalten oft eine hohe Anzahl an Problemen, die sich in folgenden Aussagen widerspiegeln könnten: „Das System ist kaum noch wartbar, wir benötigen ein neues System“, „Kleinere Änderungswünsche benötigen viele Ressourcen und werden erst in ferner Zukunft geliefert“ oder „Das ist alles unübersichtlich, wie soll man da durchblicken!“. Gemeinsamkeit

aller dieser Aussagen ist die mangelnde Qualität des Systems und somit die Nicht-Erfüllung der Qualitätsanforderungen.

Qualitätsfunktionen sollen eine objektive Messung der Qualitätsanforderungen liefern, dabei auf Mängel oder Fehler hinweisen und folglich eine angemessene Architektur, zur Zufriedenheit aller Stakeholders des Systems, begünstigen. Zusätzlich bieten Qualitätsfunktionen die Möglichkeit, die Auswirkungen von Änderungen auf die Qualität und deren Merkmale zu erkennen und schützt die wichtigsten Architektureigenschaften. Die Parallelen zu Fitness Funktionen, nach der Definition der evolutionären Architektur, sind erkenntlich.

Qualitätsanforderungen können anhand von zwei Eigenschaften definiert werden: „*Jene Qualitätsmerkmale die unseren Stakeholdern besonders wichtig sind, werden zu Qualitätsanforderungen*“ (Hirschmeier, 2021, p. 22) und „*Qualitätsanforderungen müssen explizite Entwurfsziele sein.*“ (Starke, 2020, p. 40)

Eine angemessene Architektur sollte folgende Qualitätsanforderungen erfüllen und durch Qualitätsfunktionen schützen:

- Angemessene Wartungskosten
- Erfüllung der Fachlichkeit
- Erhebung und Einhaltung der Qualitätsziele
- Anpassungsfähigkeit

Der Schutz einzelner Qualitätsmerkmale wurde in den vorherigen Kapiteln angesprochen, wie zum Beispiel die „Angemessenen Wartungskosten“. Die Wartungskosten werden stark von der Kohärenz eines Systems geprägt, welches entsprechend eine gute Modularisierung aufweist und wenige TD beinhaltet. FF für die Wartungskosten sind ein TD-Managementlebenszyklus, eine angemessene Abwägung der Modularisierungsentscheidungen auf Source-Code sowie architektonischer Ebene und die Erhebung von Metriken durch Source-Code Analyse. Des Weiteren kann die Verständlichkeit des Systems durch Mustersprachen verbessert werden (Lilienthal, 2019, pp. 105-106). Eine Übersicht ist in *Tabelle 6* dargestellt; ein Beispiel einer statischen Überprüfung der Einhaltung der Mustersprache in *Listing 2*.

Was sind Mustersprachen?	***Vorteile***	***Umsetzung***
Architekturstile → Entwurfsmuster auf der gesamter Codebasis.	Erleichtern den Aufbau und Einsatz von Schemata.	ArchUnit, SonarQube oder jQAssistant
Definition auf Klassenebene • Musterelemente • Regeln → Legen Verantwortlichkeit fest	Wirkt gegen Modularitäts- und Zyklenverstöße durch die Elementregeln → Welche Zuständigkeit hat eine Klasse.	Alle Klassen mit der Annotation @Service müssen den Namen Service enthalten und deren API methoden müssen DTO-Objekte zurückgeben.

Tabelle 6: Mustersprachen

```
@AnalyzeClasses
public class MustersprachеTest {

    @ArchTest
    public static final ArchRule all_Services_should_be_called_Service =
      classes().that()
            .areAnnotatedWith(Service.class).should()
            .haveSimpleNameContaining("Service");

    @ArchTest
    public static ArchRule all_API_methods_in_Service_layer_should_return_DTO =
      methods().that().areDeclaredInClassesThat().resideInAPackage("..services..")
            .and().arePublic().should().haveRawReturnType(DTO.class)
            .because("We use DTOs for decoupling the Service-Layer.");
}
```

Listing 2: Mustersprache Verstoß im Service-Layer Test

Neben der Einhaltung der Kohärenz muss die Erfüllung der Fachlichkeit gewährleistet werden. Die Definition von *funktionalen* und *nicht funktionalen* Anforderungen sowie deren Schutzmechanismen sind in *Tabelle 7* zusammengefasst.

Funktionale	***Nicht Funktionale***	***Qualitätsfunktionen***
Welche Probleme durch das System gelöst werden sollen.	Betrachtung der Qualitätsanforderungen und Randbedingungen.	Erhebung der Anforderungen durch Requirement Engineering

		• DDD[40] • Use Cases
		Automatisierte Test • Unit • Integration • UI (Clicktest) • Last/Performance Test
		Sind Qualitätsschöpfungsmethoden etabliert?
		Definition des Kontexts und der Randbedingungen.

Tabelle 7: Qualitätsfunktionen für die Fachlichkeit

Nach der Dokumentation von Anforderungen benötigen die Erhebung und Einhaltung von Qualitätszielen schützende Maßnahmen. Diese Qualitätsschöpfungsmethoden sind notwendig für die Einrichtung von Qualitätsfunktionen in anderen Bereichen (Kohärenz, Fachlichkeit). Bei der Erhebung hilft die Einführung von Quality Attribute Workshops (QAW). Beispiele solcher Workshops-Methodiken und Einhaltungsstrategien sind in der *Tabelle 8* repräsentiert.

Erhebung	***Einhaltung***
Quality Attribute Workshops, zum Beispiel Qualitätsnetze.	Risk-Storming
Qualitätsradar	
Erkenntnisse dokumentieren und in Qualitätsszenarien erfassen.	Checklisten für die Präsens von Qualitätsszenarien, sowie deren Vollständigkeit.

Tabelle 8: Erhebung und Einhaltung von Qualitätsziele

Das Ergebnis eines QAW ist in *Figure 13* dokumentiert. In diesem Diagramm wurden die Schätzungen unterschiedlicher Stakeholder zusammengefasst und eine gemeinsame Definition

[40] Domain Driven Design

der zu erreichenden Qualitätsziele des Systems (Gelbe Linie) erarbeitet. Zur Vereinfachung wurden nur drei Stakeholdergruppen selektiert. Die Nutzer des Systems, die strategische Leitung des Unternehmens (Führung) und die IT vertreten durch DevOps[41]. Die Definition der Qualitätsmerkmale stammte aus der ISO/IEC 25010[42].

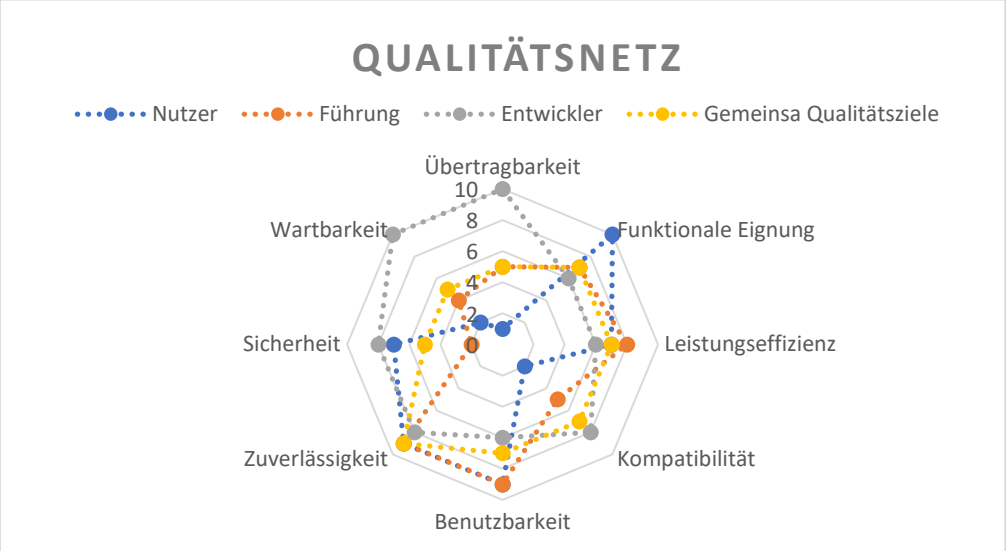

Figure 13: Beispiel eines Qualitätsnetzes

Nach der Einigung der Stakeholder kann anhand eines Risk-Stormings[43] die Überprüfung von Qualitätsverstößen stattfinden. Zuerst werden System-beschreibende Diagramme vorgestellt, um danach einzelne Risiken zu identifiziert, welche an den auftretenden Stellen angeheftet werden. Die Verwendung unterschiedlicher farblicher Post-Its ermöglichen die Einteilung der Risiken in unterschiedliche Schweregrade. Abschließend werden die Risiken besprochen, zusammengefasst und aufgenommen.

[41] https://www.atlassian.com/de/devops

[42] https://iso25000.com/index.php/en/iso-25000-standards/iso-25010

[43] https://riskstorming.com/

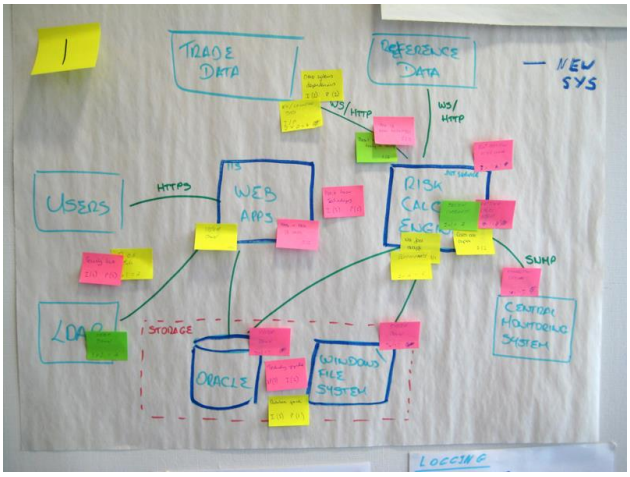

Figure 14: Risk-Storming Workshop

Quelle: https://riskstorming.com/

Zuletzt wird der Schutz der Qualitätsanforderung der Anpassungsfähigkeit beleuchtet. Die Anpassungsfähigkeit spielgelt wieder, wie ein System auf neue Anforderungen aus der Umgebung reagieren kann. Dabei spielt die Reaktionszeit bis zur Verarbeitung der Anforderung, sowie die Robustheit des Systems auf die wechselhafte Umgebung, eine wichtige Rolle. Mögliche Messungen können anhand der DORA-Metriken[44] stattfinden, welche sich auf vier Metriken konzentriert: *Deployment Frequency; Lead Time for Changes; Change Failure Rate; Time to Restore Services*. Zusätzlich bietet der Quotient zwischen den neu erstellten Anforderungen (Tickets) durch die gelösten Anforderungen (Tickets) eine weitere hilfreichere Metrik.

Eine interessante Annährung auf komplexe unsichere Anforderungen zu reagieren, ist die Verwendung von *Hypothesis-Driven Development (HDD)*, wobei die wissenschaftliche Methodik der Hypothesen und empirische Forschung auf die Entwicklung von Softwareprodukten verwendet wird. Folgen von HDD wurden von Sebastian Klepper und Bern Bruegge erforscht: „*In particular, problem solving is enhanced by allowing to recognize misconceptions earlier and discover the right solutions faster. This leads to better problem/solution fit, overall quality of*

[44] https://cloud.google.com/blog/products/devops-sre/using-the-four-keys-to-measure-your-devops-performance

software, and usability since less time is spent on correcting mistakes.“ (Sebastian Klepper, Bernd Bruegge, 2018, p. 7). Somit kann auf Änderungen der Umgebung schneller reagiert werden.

Zusammenfassung

Die ständige Messung und Beurteilung der Qualität des Systems ist kritisch für die Einhaltung der wichtigsten Architektureigenschaften und somit für die Lebensdauer des Softwareproduktes. Dabei muss der Grund der Messung den Beteiligten bewusst sein. Die Bereiche der Wartbarkeit, Erfüllung der Fachlichkeit, Erhebung der Qualitätsziele und der Anpassungsfähigkeit spielen eine zentrale Rolle in der Qualität eines Informationssystems. Willkürliche Metriken steuern das System nicht zur gewünschten Qualität; es ist erforderlich eine Einigkeit der zu gewünschten erreichenden Qualitätsziele zu erlangen. Die Einhaltung der Qualitätsfunktionen führt zu einem besseren Verständnis der Auswirkung einer Änderung.

3.4 Zusammenfassung

Eine gute Architektur bekräftigt die Stabilität, die Anpassungsfähigkeit und die geringen Wartungskosten eines Systems, um die Zufriedenheit der Stakeholder zu stärken. Die gezeigten Methodiken wirken sich auf die unterschiedlichen Ziele positiv aus. Ein Überblick der Beziehungen zwischen den Methodiken und den möglichen Einflüssen ist in *Tabelle 9* repräsentiert. Aus den gewonnenen Erkenntnissen ist ein Leitfaden des Einsatzes der Methodiken ableitbar, resultierend in die „Checkliste guter Architektur“, wobei eine wiederholte Überprüfung der Liste empfehlenswert ist.

Stabilität	**Geringe Wartungskosten**	**Anpassungsfähigkeit**
Modularisierung	Modularisierung	Modularisierung
Ausfallsicherheit	TD-Managementzyklus	TD-Managementzyklus, Abbau von absichtliche umsichtige TD
Fehlerreduzierung	Angemessene Kopplung	Angemessene Kopplung
Qualitätsfunktionen	Qualitätsfunktionen	

Tabelle 9: Beziehung zwischen Methodiken und Systemziele

Checkliste gute Architektur

- ☐ Softwarearchitektur-Prozess
  - ☐ Architektur Dokumentationen (Bsp. arc42)
  - ☐ Technologiebeurteilung
  - ☐ Architekturüberprüfung
  - ☐ Architekturziele
- ☐ Modularisierung
  - ☐ Definition von Modulen
  - ☐ Definition von Modulen in Modulen
  - ☐ Definition der Interaktionsregeln zwischen Modulen (API)
  - ☐ Dokumentation der Zerteilungsgründe von Modulen/Komponenten
  - ☐ SOC, Information Hidding, SOLID
- ☐ TD-Managementzyklus
  - ☐ TD Erkennungsvorgehen etabliert (Bsp. aim42)
  - ☐ Verantwortliche für den Zyklus benannt
  - ☐ Code Analyse
    - ☐ Abhängigkeitsverletzungen
    - ☐ Code Smells Überwachung
    - ☐ Tests vorhanden
    - ☐ Automatisierte Tests
    - ☐ Tools sind up-to-date
    - ☐ Sicherstellung der Einhaltung der Mustersprachen
- ☐ Qualitätsfunktionen
  - ☐ Mustersprachen definiert
  - ☐ Use Cases/ Test-Cases/ Anforderungs-Katalog/ Handbücher/ DDD
  - ☐ Qualität
    - ☐ Qualitätsziele Vorhanden
    - ☐ Wichtigste Qualitätsmerkmale definiert
    - ☐ Randbedingungen erfasst
    - ☐ Stakeholder definiert
    - ☐ Kontinuierliche Überprüfung der Qualitätsziele
  - ☐ Messung der Anpassungsfähigkeit
    - ☐ Gelöste Tickets vs neue Tickets
    - ☐ Deploymentzyklen
    - ☐ Fehlerrate
    - ☐ Fallback Zeit
    - ☐ Umsetzungszeiten

4 Grenzen von Technische Schulden

Die Metapher der TD, um technische Missstände zu veranschaulichen, besitzt Grenzen. Die Effektivität des literarischen Hilfsmittels ist kontextgebunden. Probleme außerhalb der Technik lassen sich schwierig mit der Metapher darstellen und nicht jeder Missstand lässt sich durch optimalere technische Prozesse lösen. Das Bewusstsein der Grenzen des Einflussbereiches der Technik ermöglicht die Erkennung der Schulden, welche technik-getrieben sind. Somit können die Ursachen nicht-technisch-getriebener Schulden verfolgt werden und an die entstehenden Stellen adressiert werden. Es findet eine Ursachenbekämpfung statt, anstatt einer Symptombekämpfung. Abweichungen des gewünschten Ablaufes von Geschäftsprozessen entstehen im gesamten Unternehmen. Drei Beispiele von Schulden stellen die beiden folgenden Zitate und Conways Law dar.

„*We view architecture stories as separate from technical debt stories. Technical debt stories usually capture things a developer needs to do in a later iteration to „clean up the code“, whereas an architecture story […]*“ (Neal Ford & Mark Richards, 2021, p. 84)

Conways Law: „*Organisationen, die Systeme entwerfen, sind gezwungen, Designs zu produzieren, die Kopien der Kommunikationsstrukturen dieser Organisationen sind.“* Dies bedeutet, wenn zum Beispiel Teams organisatorisch nach technischer Zuständigkeit getrennt sind (UI, Backend und DB), wird ein System mit genau der gleichen Trennung entstehen. Dies kann als Schulden aufgegriffen werden. Eine Änderung kann mehrere Iterationen benötigen, da erst das DB-Team die Felder in der DB zur Verfügung stellen muss, damit in der nächsten Iteration das Backend-Team diese verwenden kann. Schließlich kann das UI-Team in der dritten Iteration die Information anzeigen.

„*Überrascht es Sie, zu erfahren, […], dass Sie Modellierungsschulden für ein DDD-Projekt zurückzahlen müssen?“* (Vernon, 2017, p. 125)

Anders gesagt entstehen durch die zeitliche Begrenzung von Modellierungsarbeiten Modellierungsschulden. Der Umfang der perfekten Durchführung jeglicher Modellierungsaufgabe kann im Vorfeld nicht geschätzt werden. Der Informationsschöpfende Prozess wird iterativ Erkenntnisse über die Übereinstimmung des Modells und der fachlichen Anforderungen sammeln.

Dieser Prozess muss beim Ablauf der eingeplanten Zeit unterbrochen werden, infolgedessen wird nie ein perfektes Modell gefunden und die Unreinheiten führen zu Modellierungsschulden. (Vernon, 2017)

Die entstehende Analogie zwischen TD und Modellierungsschulden führt zur Frage, wie viele Schuldenbereiche zusätzlich zu TD vorhanden sind. Aktuell gibt es keine Recherche, welche analog zur Ontologie von TD eine Ontologie aller Schulden von Softwareprodukten sammelt. Für den weiteren Diskurs der Arbeit wird in diesem Kapitel ein Rahmen für alle Schulden erstellt.

Für die Erstellung des Rahmens wurden drei wichtige Komponenten für den Erfolg von Softwareprojekten bzw. Produkten identifiziert: Organisation, Technik und Fachlichkeit (Wolff, 2018, p. 56). *Tabelle 10* liefert eine Erläuterung, welche Tätigkeiten in den entsprechenden Bereichen fallen, wobei *Figure 15* eine Zuordnung zwischen den TD der Ontologie aus Kapitel 3.2.1 und den Bereichen wiedergibt und die Grenzen von TD verdeutlicht.

Technik	**Organisation**	**Fachlichkeit**
Implementierung	Organigramme (Teamzerteilung)	Fachlichen Schnitt des Softwareproduktes (Bsp. DDD)
Softwarearchitektur	Kommunikationsfluss	Anforderungen
Betrieb der Software	Conways Law	Produkt Entwicklung
	Strategie Entwicklung	

Tabelle 10: Komponenten eines Softwareprodukts

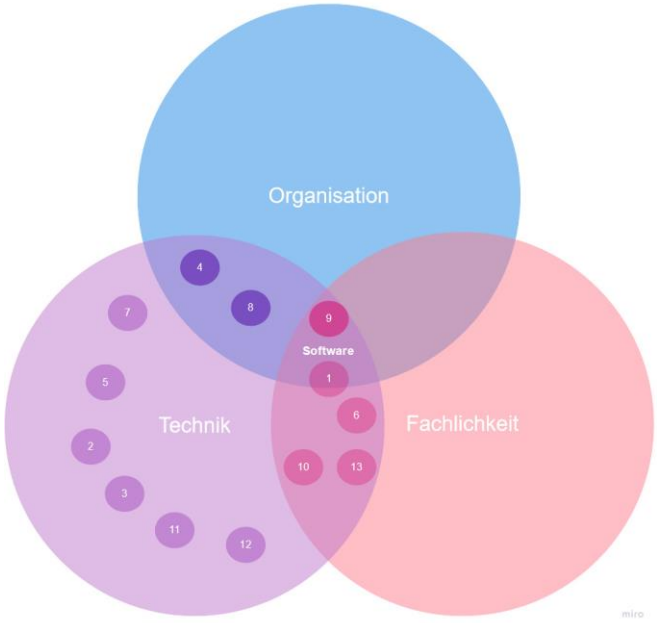

Figure 15: Zuordnung TD zu Softwareprojekt Bereichen

Eine weitere Verdeutlichung der Grenzen der Technik liefert James Lewis in seinen Konferenzbeitrag in der NDC in Kopenhagen. James Lewis definiert neun Eigenschaften, welche zu einem erfolgreich verteilten System führen (Lewis, 2019). Die Eigenschaften sind in *Figure 16* aufgelistet und wurden durch farbliche Vierecke mit den ersten Buchstaben des Bereiches zu den entsprechenden drei Bereichen zugeordnet.

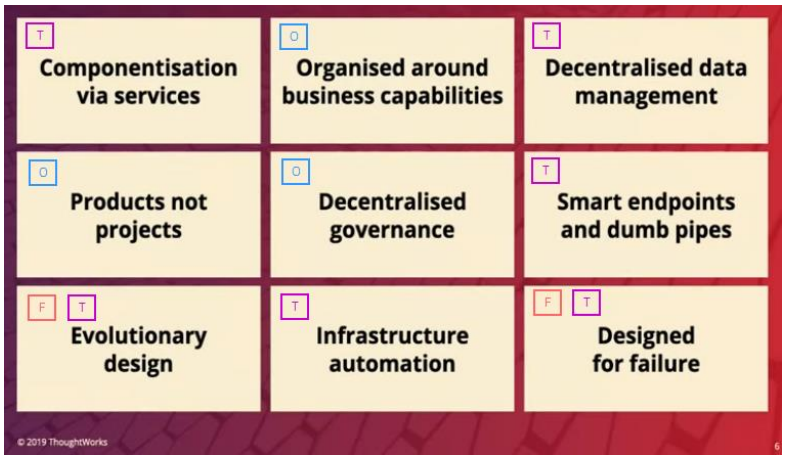

Figure 16: Erfolgreiche Eigenschaften für das Design von verteilten Systemen (VS)

Quelle: https://www.youtube.com/watch?v=uAwJEFLJunk

Nach der Zergliederung der Schulden in drei Bereiche wird eine Nomenklatur für die Sammlung der Schulden benötigt. Die Überladung des Begriffes TD würde zu Missverständnissen führen, weil eine konstante kontextgebundene Differenzierung stattfinden müsste. Der neue Begriff muss die folgenden Motivationen wiedergeben:

- Veranschaulichung eines negativen Deltas zu der gewünschten Produktvision.
- Behebung der Deltaursachen ist nicht immer technisch möglich.
- Erkennung aller Generierungspunkte für Schulden.

„*Software Product Debt: Eine suboptimale Lösung, welches ein negatives Delta zur gewünschten Produktvision generiert und die Automatisierung, Ausführung, Erstellung und Anpassung des Geschäftsprozess beeinträchtigt.*“

Im weiteren Verlauf der wissenschaftlichen Arbeit definiert *Software Product Debt (SPD)* alle Schulden, welche in den drei Komponenten eines Softwareprojektes entstehen können, wobei sich TD nur auf den technischen Teil fokussiert.

Zusammengefasst ergeben sich drei Erkenntnisse:

- Die entwickelten IT-Produkte sind ein Enabler und Teil der Geschäftsprozesse.
- Geschäftsprozesse sind mit Schulden belastet, welche außerhalb der Technik angesiedelt sein können.
- SPD ist eine einfache Metapher, um wirtschaftliche Folgen darzustellen.

5 Angemessene Microservices-Architektur

Die Microservice-Architektur hat sich als eine der dominierenden Architekturen von Informationssystemen durchgesetzt. In der Architektur-Trend-Analyse von InfoQ aus 2019 werden Microservice als Late Majority eingestuft (*Figure 17*), was eine breite Verwendung von Microservices-Architekturen in der Industrie darstellt. Erfüllen Microservices, trotz deren flächendeckender Verwendungen, alle Requisiten für eine angemessene Architektur?

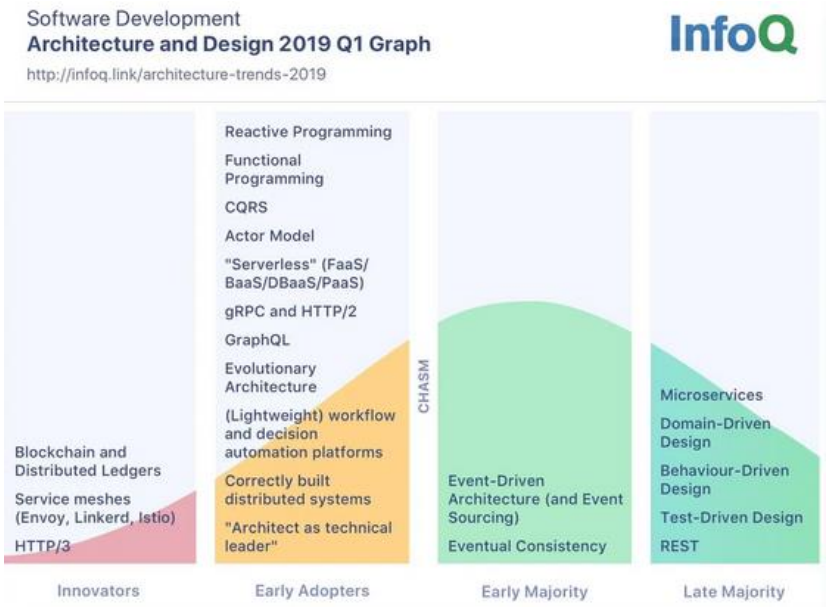

Figure 17: Architektur-Trend-Analyse Q1 2019

Eine erste Annährung, um Microservices-Systeme zu beurteilen, bietet die erarbeitete Checkliste aus dem dritten Kapitel. Zusätzlich zur Abarbeitung der Checkliste, muss die Beleuchtung der richtigen Entscheidungen und deren Auswirkung auf die Eigenschaften einer angemessenen Architektur (Stabilität, Wirtschaftlichkeit und Anpassungsfähigkeit) durchgeführt werden.

Die technischen Gegebenheiten von Microservices begünstigen die Modularisierung und die Anpassungsfähigkeit von Systemen. Somit werden sechs Punkte durch das richtige Einsetzen von Microservices erfüllt (*Tabelle 11*). Die Betrachtung der Checkliste in *Tabelle 11* würde zu

einem negativen Ergebnis für die Verwendung von Microservices führen. Die Auswahl einer Architektur führt nicht zur Erfüllung aller Kriterien, sondern erleichtert oder erschwert diese. Für eine tiefgreifende Analyse, ob Microservices eine gute Architektur ermöglichen, wird die Umsetzung aller vier Bereiche der Checkliste wie Softwarearchitektur-Prozesse, Modularisierung, TD-Managementzyklus und Qualitätsfunktionen, begutachtet.

☐ Modularisierung ☑ Definition von Modulen ☐ Definition von Modulen in Modulen ☑ Definition der Interaktionsregeln zwischen Modulen (API) ☐ Dokumentation der Zerteilungsgründe von Modulen/Komponenten ☑ SOC, Information Hidding, SOLID	☐ Messung der Anpassungsfähigkeit ☐ Gelöste Tickets vs neue Tickets ☑ Deploymentzyklen ☐ Fehlerrate ☑ Fallback Zeit ☑ Umsetzungszeiten

Tabelle 11: Erfüllung der Liste durch Microservices

5.1 Softwarearchitektur-Prozess

Diese Arbeit beschäftigt sich mit der kritischen Beurteilung des Architekturansatzes in Bezug auf Microservices-Architektur. Eine Anleitung, um einen Softwarearchitektur-Prozess und deren Varianten zu etablieren, ist nicht Ziel dieser Arbeit. Somit wird eine mögliche Implementierung eines Prozesses für Microservices dargestellt. Weitgehende Einblicke bieten (Fowler, 2019), (Starke, 2020), (Toth, 2019) und (Mark Richards & Neal Ford, 2020).

Die Architektur von Microservices-Systeme weist eine grundlegende Herausforderung auf, nämlich die Zerteilung in eine Makro- und Mikro-Architektur. Dabei kann zwischen den Ebenen ergänzend eine Bounded-Context-Architektur hinzugefügt werden (*Figure 18*). Die Verwendung einer Bounded-Context-Architekturebene wird im weiteren Verlauf für als Lösungsvorschlag mancher Probleme verwendet.

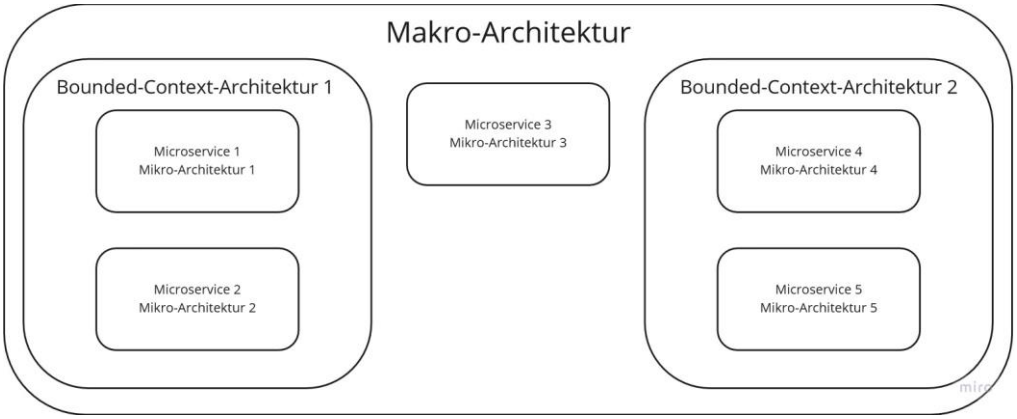

Figure 18: Microservice-Architektur-Ebenen

Diese Zerteilung sowie die steigende Komplexität des Systems führen dazu, dass eine einzelne Gruppe von Enterprise-Architekten nicht mehr am effektivsten agieren kann. Wird das Harmel-Law[45] und die Verantwortlichkeitsverlagerung von Microservices in selbstverantwortliche autonome Teams berücksichtigt, ist demzufolge die Erfüllung der Rolle des Architekten in jedem einzelnen Team ratsam. Dabei können drei Modelle (Toth, 2019) (Toth, 2015) für Mikroservices-Architekturen verwendet werden (*Tabelle 12*).

Unterstützende/r Architekten/-innen	**Architekturagenten/-innen**	**Team als Softwarearchitekten/-innen**
1 bis n Entwickler/-innen übernehmen die Architektenrolle in Teilzeit. Hauptfokus liegt auf Mentoring.	Architekturthemen werden Spezialisten/-innen zugewiesen, welche bei der Entscheidungsfindung unterstützen.	Selbstorganisierte Teams klären Architekturfragen und -entscheidungen.

Tabelle 12: Softwarearchitekten Modelle

Die Etablierung eines der drei Modelle führt zur Einführung vom Softwarearchitektur-Prozess bis zur Bounded-Context-Ebene. Die Abstimmung der Makroarchitektur erweist sich als schwieriger. Eine Variante, Zuständigkeiten für die Makroarchitekturen zu platzieren, ist die

[45] https://www.infoq.com/news/2022/01/software-architecture-advice/

Bildung von Architekturgremien oder Chapter und Gilden[46] , bestehend aus Enterprise-Architekten/-innen und Architekturverantwortlichen der jeweiligen Teams. Gleichzeitig können mehrere Arbeitsgruppen etabliert werden und thematisch zergliedert werden.

Mit der Einführung des Softwarearchitektur-Prozesses werden alle Kriterien des ersten Bereiches erfüllt. Nachteilig ist der zusätzliche organisatorische Aufwand bei der Koordination von Themen der Makroebene.

- ☑ Softwarearchitektur-Prozess
  - ☑ Architektur Dokumentationen (Bsp. arc42)
  - ☑ Technologiebeurteilung
  - ☑ Architekturüberprüfung
  - ☑ Architekturziele

5.2 Modularisierung

Microservices-Systeme stützen sich auf das Prinzip von *Divide and Conquer*, was eine Auseinandersetzung mit der Findung von Modulen und Komponenten sowie deren Schnittstellen erzwingt. Der vorhandene Softwarearchitektur-Prozess vereinfacht die Dokumentation des Entscheidungsprozesses der Modularisierung, welches schlussendlich zur Erfüllung der Checkliste im Bereich Modularisierung führt.

- ☑ Modularisierung
  - ☑ Definition von Modulen
  - ☑ Definition von Modulen in Modulen
  - ☑ Definition der Interaktionsregeln zwischen Modulen (API)
  - ☑ Dokumentation der Zerteilungsgründe von Modulen/Komponenten
  - ☑ SOC, Information Hidding, SOLID

Obwohl die Liste erfüllt ist, bestehen weiterhin Ungereimtheiten. Wie im Kapitel 3.1 beschrieben, ist die Findung des Schnittes der komplexere Aspekt von Modularisierung und kann aus technischer und fachlicher Sicht betrachtet werden. Empfehlenswert für die fachliche Zuordnung von Zuständigkeiten bei Microservices ist DDD. DDD kann in drei Bestandteile

[46] https://www.productplan.com/glossary/tribe-model-management/

zergliedert werden: strategisches Design, taktisches Design und Ubiquitous Languages (dt.: allgegenwärtige Sprache) (Vernon, 2017, pp. 7-9).

Aus technischer Sicht beeinflussen die nicht-funktionalen Dimensionen und die Data Des/-Integrators (Unterkapitel 3.1) die Zerlegung von Modulen. Dabei spielt bei Microservices-Systeme die Granularität eine entscheidende Rolle (Neal Ford & Mark Richards, 2021, pp. 185-187). Somit müssen die nicht-funktionalen Dimensionen erweitert werden, um die richtigen Grenzen eines Microservices zu finden. Zur Verständlichkeit und Verdeutlichung der Wichtigkeit der Separierung, der Dimensionen, der Granularität und Modularität, dient das folgende Zitat:

„In our usage, modularity concerns braking up systems into separate parts […], whereas granularity deals with size of those separate parts. […] most issues and challenges within distributed systems are typically not related to modularity, but rather granularity.“ (Neal Ford & Mark Richards, 2021, p. 187)

Die zusätzlichen Dimensionen der Granularität sind (Neal Ford & Mark Richards, 2021, pp. 188-196):

- Service scope and function → Führt der Service zu viele nicht miteinander verwandte Aktionen aus (Kohäsion Verletzung)?
- Code volatility → Unterschiedliche Änderungsfrequenz von Subbereichen des Microservices.
- Scalability and throughput → Unterschiedliche Skalierungs- und Durchsatzbedarfe von Subbereichen des Microservices.
- Fault tolerance → Die Eigenschaft der Aufrechterhaltung einer Applikation oder Funktionalität bei gravierenden Ausfällen.
- Security → Benötigen Bereiche im Microservice andere Sicherheitsstandards?
- Extensibility → Die Erweiterbarkeitswahrscheinlichkeit eines Services.

Die Ähnlichkeit der Definitionen von Modularität und Granularität spiegelt sich in den jeweils zu betrachtenden Dimensionen wider, welche Parallelen und Überschneidungen aufweisen.

Diese neuen zu betrachtenden Dimensionen erhöhen die Komplexität der Lösungen und erfordern eine Koordination zwischen den Microservices. Die Verlagerung der Komplexität des Ablaufes aus einen Softwareprojekt[47] auf die Kommunikationsebene zwischen separate Services, ermöglich eine neue Vielfalt an ATD. Der wissenschaftlichen Artikel „*Identifying architectural technical debt, principal, and interest in microservices: A multiple-case study*“ erfasst mehrere gravierende ATD (Saulo S.de Toledo, Antonio Martini, Dag I.K.Sjøberg, 2021), welche hauptsächlich auf Kommunikationsprobleme in einem Microservices-System zurückzuführen sind. Die entstehenden Nachteile erfordern zusätzliche Methodiken, um ATD zu überwachen und erkennen (Kapitel 5.5) und erweitern die Checkliste in den Bereichen Modularisierung und TD-Managementzyklus.

- ☑ Modularisierung
  - ☑ Fachliche Zergliederungsmethodik
  - ☑ Definition von Modulen
  - ☑ Definition von Modulen in Modulen
  - ☑ Definition der Interaktionsregeln zwischen Modulen (API)
  - ☑ Dokumentation der Zerteilungsgründe von Modulen/Komponenten
  - ☑ SOC, Information Hidding, SOLID
  - ☑ Definition von Microservices
  - ☑ Trade-off-Analyse zur Microservices Zerteilung/Zusammenführung
    - ☑ Koordination definiert

5.3 Erkennung von Technischen Schulden bei Microservices-Systeme

Wie die Voraussetzung des Softwarearchitektur-Prozesses keine einzigartige Eigenschaft von Microservices-Systeme ist, wird ebenfalls ein TD-Managementlebenszyklus benötigt. Im Unterkapitel 3.2 und dessen Abschnitte wurden mehrere Techniken, Methodiken und Tools dargestellt, um solch einen Zyklus zu etablieren und auszuführen. Diese Erkenntnisse sind auf der kompletten Mikroebene in einem Microservices-System anwendbar. Die wahre Herausforderung entsteht bei der Betrachtung der Makroebene. Viele der Methodiken stützen sich auf Source-Code Analysen, welche begrenzt über die Grenzen von Microservices verwendet

[47] Hier als zusammenhängende Source-Code-Basis in einem technischen Projekt/Solution verwendet.

werden können. Ein Beispiel liefert die Überprüfung von Modularisierungsverstößen. Der Test in *Figure 10* und *Listing 1* ist weiterhin anwendbar, aber die Überprüfung der Zugriffe zwischen den fachlichen Modulen, welche in unterschiedlichen Microservices verankert sind, ist nicht möglich.

Diese Erkenntnis deckt sich mit der Sammlung aus Unterkapitel 5.2 und verschärft die Notwendigkeit der Erkennung von ATD auf Makroebene. Demzufolge wird die Checkliste für den Bereich TD-Managementzyklus nicht vollständig erfüllt.

- [x] TD-Managementzyklus
  - [x] TD Erkennungsvorgehen etabliert (Bsp. aim42)
  - [x] Verantwortliche benannt
  - [x] Code Analyse
    - [ ] Abhängigkeitsverletzungen
    - [x] Code Smells Überwachung
    - [x] Tests vorhanden
    - [x] Automatisierte Tests
    - [x] Tools sind up to date
    - [ ] Sicherstellung der Einhaltung der Mustersprachen
  - [ ] ATD Erkennungsstrategie
    - [ ] Strategie up to date

5.4 Fitness Funktionen

Fitness Funktionen und Qualitätsfunktionen verfolgen die gleichen Ziele und basieren auf den gleichen Definitionen. Beide Konzepte können im Kontext dieser Arbeit als Synonyme verwendet werden. In Anlehnung einer evolutionären Architektur wird der Begriff Fitness Funktionen verwendet.

Die Steuerung von Fitness Funktionen erfolgt uneingeschränkt von der ausgewählten Architektur. Die technischen Vorraussetzungen von Microservices begünstigen viele FF. So fordern größere Microservices-Systeme einen hohen Reifegrad des Entwicklungsprozesses. CD-Pipelines sind deklariert und führen diverse Stufen von Tests aus (Unit/Integration/Performance/Architecture/Bugs Findings). Zusätzlich erleichtert die Isolierung von Microservices die Messung der Anpassungsfähigkeit und übt einen positiven Einfluss auf HDD aus.

Microservices-Architekturen stellen keine zusätzliche Herausforderung für Fachliche FF dar. Als Beispiel dient folgender realer anonymisierter Fall:

Szenario

Die Firma *Verkauf alles GmbH* besitzt Filialen und Verkaufspartner weltweit. Die Firma hat sich auf individuell konfigurierbare Produkte spezialisiert. Online oder vor Ort können die Produkte per Konfiguratoren angepasst und bestellt werden, um schlussendlich vom Kunden abgeholt zu werden. Zusätzlich zur Anwendung „*Konfigurator*“ entwickelt die Firma sämtliche Systeme hausintern, wie zum Beispiel das CRM-System, das Vertriebsportal oder IoT-Systeme zwischen diversen Produktgruppen. Der Datenzugriff findet übergreifend durch mehrere Anwendungen statt und das Systems ist als Monolith implementiert. Das ganze System wurde als Legacy System deklariert. Besonders die „*Konfigurator*“-Anwendung ist nicht mehr wartbar. Eine Erneuerung des Systems soll stattfinden.

Anwendung

Die Anwendung „*Konfigurator*“ ermöglicht die individuelle Gestaltung von Produkten anhand der Anpassung der Bestandteile. Die Anpassungen und Zusammensetzung der Produkte müssen definierte Regeln einhalten, welche als boolische Überprüfungen im Source-Code hinterlegt sind. Mehrere Anpassungen können in einem vordefinierten Konfigurationspaket vereint sein, welches mit Rabatten versehen sein kann. Konfigurationspakete können das Hinzufügen bestimmter Anpassungen verbieten, ohne enthaltene Anpassungen zu erfordern. Zugleich können einzelne Anpassungen ebenso andere ausschließen.

Die Anwendung bietet beim Konfigurationsprozess dem Anwender Vorschläge an, welche die Regeln erfüllen oder Hinweise auf Regelverstöße anzeigen. Technisch sind die Daten zwischen der relationalen DB des Monolithen und des Source-Codes verteilt. Neue Regeln werden von einer Fachabteilung definiert, welche diese zum Teil selbst über eine Oberfläche pflegt oder von der IT einbauen lässt.

Anforderung

Der Auftraggeber ist mit dem aktuellen Stand der Software nicht zufrieden. Die Wartungskosten sind in den letzten Jahren stark gestiegen, wobei Änderungen lange Zyklen benötigen und

neue Anforderungen sehr schwer zu überblicken sind. Im Qualitäts-Workshop wurden die Stabilität und die Einfachheit des Systems als priorisierte Ziele erfasst.

FF

Die Unterstützung von Microservices, um ein stabiles System zu erlangen, wurde anhand mehrerer Beispiele revidiert, aber die Erreichung eines weichen Ziels „*Das System soll einfach sein.*“ stellt eine Herausforderung dar.

Nach einem weiteren Workshop konnte das weiche Ziel in drei Qualitätsanforderungen zerteilt werden.

Neue Regeln dürfen maximal vier Wochen bis zur Einführung benötigen.

Die Einarbeitung neuer Mitarbeitenden in die Anwendung darf maximal drei Wochen betragen.

Das Kunden Feedback zur Komplexität der Benutzung der Anwendung darf vier von fünf nicht unterschreiten.

Diese drei FF stellen fachliche Überprüfungen dar, welche unabhängig von der Architektur gemessen werden können. Die Migration der Anwendung zu einem Microservice mit einer Graphdatenbank erleichterte das Erreichen der FF.

Die Autonomie und technische Vielfalt von Microservices erschwert die Einhaltung von Mustersprachen. Jeder Microservice kann beliebig die passende Programmiersprache und interne Architektur wählen, was die Einrichtung einer Mustersprache für alle Microservices fast unmöglich gestaltet und ergibt folgende Erfüllung der Checklisten.

- ☐ Qualitätsfunktionen
  - ☐ Mustersprachen definiert
  - ☑ Use Cases/ Test-Cases/ Anforderungs-Katalog/ Handbücher/ DDD
  - ☑ Qualität
    - ☑ Qualitätsziele Vorhanden
    - ☑ Wichtigste Qualitätsmerkmale definiert
    - ☑ Randbedingungen erfasst
    - ☑ Stakeholder definiert
    - ☑ Kontinuierliche Überprüfung der Qualitätsziele
  - ☑ Messung der Anpassungsfähigkeit

- ☑ Gelöste Tickets vs neue Tickets
- ☑ Deploymentzyklen
- ☑ Fehlerrate
- ☑ Fallback Zeit
- ☑ Umsetzungszeiten

5.5 ATD

ATD stellen suboptimale Lösungen der schwierigsten Entscheidungen hinsichtlich des Entwurfs von Software dar. Jede dieser Entscheidungen bergen ein besonders hohes Risiko. Charakteristisch für ATD ist deren breite Verteilung im Source-Code und eine erschwerte Identifikation (mostly invisible) (Kruchten, P., Nord, R. L., and Ozkaya, I., 2012). Demzufolge entstehen hohe Zinsen und Lösungskosten. Microservices erhöhen die Anzahl von architektonischen Entscheidungen. Die höhere Anzahl an eingesetzten Technologien im Vergleich zu Monolithen, die Kommunikationsstrategien zwischen jeden einzelnen Microservice und die hohe Umsetzungsrate von Features (Antonio Martini, Jan Bosch, Michel Chaudron, 2014). Zudem basieren die meisten Erkennungsmethodiken auf Source-Code-Analysen, welche durch die Verteilung des Source-Codes erschwert wird. Um der Komplexität bei der Erkennung von ATD entgegenzuwirken, werden in diesem Unterkapitel helfende Mechaniken und zu vermeidende Umsetzungen beleuchtet, um abschließend eine Checkliste aus Praktiken und Indikatoren zu erstellen.

5.5.1 Core Features

ATD bieten eine wirtschaftliche Betrachtung der vorhandenen Missstände. Die höhere Verteilung von ATD im gesamten System, sowie die geringe Unterstützung durch automatisierte Erkennungsmechanismen, erhöhen die Identifikationskosten von ATD. Das Erreichen einer systemübergreifenden Architektur mit wenigen ATD kann infolgedessen kostspielig sein. Die Priorisierung der zu betrachtenden Bereiche ist eine Kostenreduzierungsstrategie, welche die

Stabilisierung der Core Features verfolgt. Zwei mögliche Methoden bieten die strategische Bewertung nach DDD[48] und Wardley Maps[49] an.

Strategische Bewertung nach DDD

DDD zerteilt das System in Bounded Contexts, welche bei idealer Modellierung jeweils auf eine Domäne oder Subdomäne gemappt werden (Vernon, 2017, p. 43). Nach strategischer Wichtigkeit wird die Domäne einer Hauptart zugeordnet: Core (dt.: Kern), Supporting (dt.: unterstützend) und Generic (dt.: generisch) (*Figure 19*).

Core Domäne: Das Feld, in dem sich das Unternehmen hervorhebt und Wettbewerbsvorteile erlangt. Hier sollte die stärkste Investition stattfinden, personell sowie finanziell (Vernon, 2017, pp. 44-45). Die Domäne sollte wenige ATDs aufweisen, um eine gute Anpassungsfähigkeit zu erlangen und somit weiterhin wettbewerbsvorteilhafte Innovationen schnell an den Markt bringen.

Supporting Domäne: Domäne, welche für den Erfolg des Unternehmens notwendig ist, aber nicht als Core Domäne kategorisiert worden ist. Ohne die Domäne können Core Domänen keinen Erfolg erlangen (Vernon, 2017, p. 45). Die Domäne sollte stabil sein und eine gewisse Anpassbarkeit vorweisen. Eine angemessene Anzahl von ATDs ist empfehlenswert.

Generic Domäne: Produkte, welche keine Besonderheit für das Unternehmen darstellen und extern eingekauft werden können (Vernon, 2017, p. 45). Hier sollte die Investition minimal sein.

[48] https://docs.microsoft.com/de-de/azure/architecture/microservices/model/tactical-ddd

[49] https://medium.com/wardleymaps

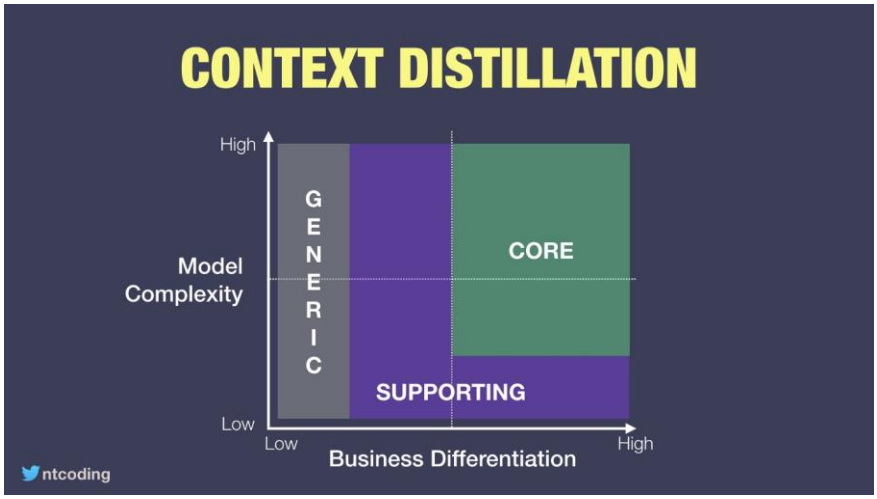

Figure 19: Domänenarten

Quelle: https://medium.com/nick-tune-tech-strategy-blog/core-domain-patterns-941f89446af5

Wardley Maps

„*Wardley Maps sind grafische Hilfsmittel, die Produkte und Produktportfolios, aber auch Wertschöpfungsketten sowie Organisationsstrukturen visualisieren.“* (Spitz, 2021) Die Komponenten werden anhand zwei-dimensionaler Koordinatensystemen arrangiert. Die X-Achse repräsentiert den Reifegrad des Produktes (Evolutionsstufen), wobei die Platzierung auf der Y-Achse, die Sichtbarkeit und Relevanz (Value Chain) aus der Sicht der Stakeholder erfasst. Insgesamt sind vier Evolutionsstufen definiert (Wardley, 2018):

- Genesis (dt.: Entstehung): Komponenten in der Entwicklungsphase, welche Unsicherheiten oder konstanten Änderungen ausgesetzt sind, mit Fokussierung auf Erkundung.
- Custom built (dt.: Eigenbau): Vertretung der ungewöhnlichen und zu lernenden Produkte, welche individuell angefertigt werden. Es entsteht ein Fokus auf Lernen und Entwickeln.
- Product (dt.: Produkt): Etablierte Produkte am Markt mit hoher Nachfrage und Stabilität. Der Fokus liegt auf Verfeinerung und Verbesserung.
- Commodity (dt.: Gebrauchsgut): Stark standarisierte allgegenwärtige Produkte, welche als selbstverständlich gelten. Der Fokus zentriert sich auf die Steigerung der Effizienz der Produktion.

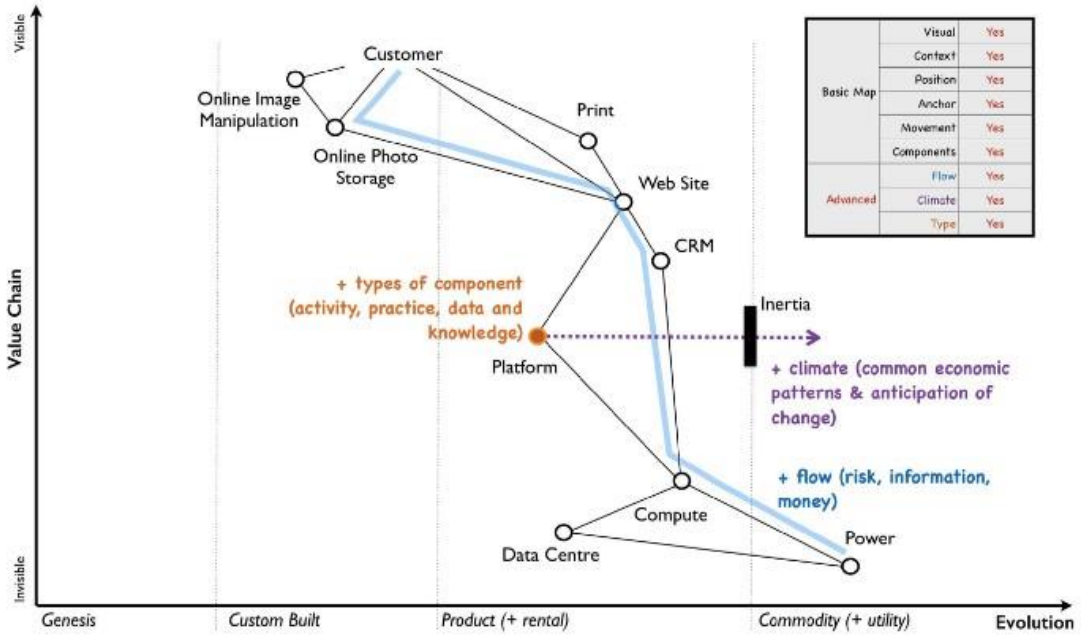

Figure 20: Wardley Map Online Foto Service

Quelle: https://medium.com/wardleymaps/finding-a-path-cdb1249078c0

Figure 20 repräsentiert die Wertschöpfungskette eines Online Foto Services im Jahr 2005 (Wardley, 2018). Das Diagramm betrachtet die Produkte aus Sicht des Customers (de.: Kunden) und gibt einen einfachen Einblick in die Erstellung von Wardley Maps. Zusätzlich zu den bekannten Eigenschaften eines Wardley Maps, können weitere strategische Elemente hinterlegt werden, welche als „*Advanced elements*“ definiert sind: types of component (Aktivitäten, Praktiken, Daten und Wissen); climate (dt.: Klima, eher Trend gemeint); flow (dt.: Fluss) von Geld, Information oder Risiko; inertia (dt.: physikalische Trägheit) (Wardley, 2018).

5.5.2 Architektonisches Quant

Ziel der Trade-off-Analyse, gestützt durch die Erkenntnisse vom architektonischen Quant, ist die Identifizierung von ungewünschten Kopplungen und die Evaluation von gewünschten, d. h. die Findung angemessener Kopplungen (siehe 2.2.2). Dies hat die Vermeidung von ATD und die Stärkung der Doktrin des Time-to-Market zur Folge. Projiziert auf ein Microservices-System werden hauptsächlich die Eigenschaften „independent development“ und „resilience“

gestützt. Als architektonisches Quant wird ein „*unabhängiges deploybare Artefakt mit statischen Kopplungen"* definiert (Neal Ford, Rebecca Parsons, Patrick Kua, 2017) (Neal Ford & Mark Richards, 2021). Genauer ist die folgende Definition aus dem Buch „*Architecture The Hard Parts*", welche um fünf Implikationen erweitert wird.

„*[…] an architectural quantum is defined as an independently deployable artifact with high functional cohesion, high static coupling, and synchronous dynamic coupling.*" (Neal Ford & Mark Richards, 2021, p. 144)

- Jedes architektonische Quant ist in seiner Umgebung eingeständig lauffähig.
- Ein Quant ist erreichbar, bzw. kann kommunizieren.
- Ein fachlicher Prozess kann mehrerer Quanten durchlaufen. Wobei kein Quant für den Gesamterfolg des Workflows verantwortlich sein muss.
- Ein Quant kann mehrere fachliche Prozesse bedienen.

Statische Kopplung

Die statische Kopplung bezieht sich auf die starken Wechselwirkungen zwischen den Elementen in einem Quant, wodurch diese Elemente technisch nicht voneinander trennbar sind. Teil der statischen Kopplung sind alle notwendigen Abhängigkeiten für das Betreiben des Quants, abgebildet in Verträgen. Beispiele dafür sind:

- Abhängigkeiten, welche im Dependency Manager deklariert sind. (NuGet, Maven, Gradle)
- Das Betriebssystem, das für die Ausführung benötigt wird.
- Docker Images im Dockerfile[50] oder Docker Compose[51].
- Verwendete Data Store.
- UI

[50] https://docs.docker.com/engine/reference/builder/

[51] https://docs.docker.com/compose/compose-file/

- Jede Integrationsebene kann zu einer statischen Kopplung führen.

Abschließend liefert die statische Kopplung eine Antwort auf die Frage „*How ist he service wired together?*“ und definiert somit die Grenzen eines Quants (Neal Ford & Mark Richards, 2021).

Dynamische Kopplung

Die Betrachtung der dynamischen Kopplung hilft die Grenzen des Quants zu erweitern und liefert den Grundstein für eine Tradeoff-Analyse über die vorhandene Kopplung zwischen Quanten. Kommunizieren zwei Quanten untereinander synchron, verschmelzen beide zu einem Quant. Durch die Synchronität entsteht eine starke Abhängigkeit zur Laufzeit. Ist der *Quant1* nicht erreichbar, ist *Quant2* nicht komplett lauffähig - es entstehen, unabhängig vom eigenen Zustand, Fehler. Zusätzlich bildet sich eine Deployment-Abhängigkeit zwischen den Quanten. Wird eine neue Version von *Quant1* deployt, kann *Quant2*, solange keine Version von *Quant1* erreichbar ist, nicht seinen Teil des Workflows vervollständigen. Dies kann durch Rolling-Deployment[52] gelindert werden, trotzdem besteht eine Verletzung der Eigenschaft „independent development/deployment“, wodurch beide Quanten in einen verschmelzen.

Ist das Quant um alle Quanten erweitert, welche untereinander synchron kommunizieren, verbleiben nur asynchrone Aufrufe oder keine Aufrufe mehr. Bei Letzterem wäre das System sehr stark gekoppelt. Eine Quant Anzahl von 1 deutet auf einen Monolithen hin und entspricht einer ATD in einem verteilten System, wie Microservices-Systeme.

Nach der Identifizierung aller Quanten des zu betrachtenden Bereiches, erfolgt eine Bewertung der Kopplung zwischen und innerhalb der Quanten. Jede Kopplung sollte einzeln, unter Betrachtung der drei Kopplungsdimensionen (*Tabelle 13*) „*Kommunikationstyp*“, „*Kontistenz*“ und „*Koordination*“, bewertet werden (Neal Ford & Mark Richards, 2021).

[52] https://docs.aws.amazon.com/whitepapers/latest/overview-deployment-options/rolling-deployments.html

Kopplungsdimension	**Starke Kopplung**	**Lose Kopplung**
Kommunikationstyp	Synchron	Asynchron
Konsistenz	ACID	BASE[53]
Koordination[54]	Orchestrierung	Choreografie

Tabelle 13: Ausprägungen der Kopplungsdimensionen

Methodischer Ablauf

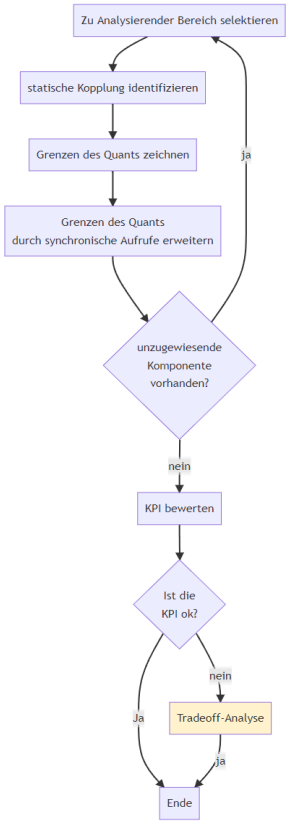

Figure 21: Ablaufdiagramm Architektonisches Quant

Ein möglicher Ablauf des Vorgehens stellt das Ablaufdiagramm in *Figure 21* dar. Als erstes erfolgt die Selektion des zu analysierenden Bereiches. Nachfolgend werden alle statischen Kopplungen im Bereich identifiziert und die vorläufigen Grenzen des Quants oder Quanten gezeichnet. Fortführend werden die vorläufigen Grenzen anhand der synchronen Aufrufe zwischen den Quanten erweitert. Das resultierende Ergebnis ist eine Zuordnung jeweils aller Komponenten zum jeweiligen Quant. Sind destotrotz nicht zugewiesenen Komponenten vorhanden, wird das Verfahren wiederholt und die Ergebnisse zusammengeführt. Dabei wird der zu analysierende Bereich angepasst. Die architektonische Bewertung der Funde erfolgt über die Anzahl der Quanten, welches in einer KPI[55] abgebildet ist und findet sich im Schritt „*KPI bewerten*“ wieder. Die Bewertung ist individuell für den jeweiligen Bereich zu betrachten. Mit der Einhaltung der KPI endet das Verfahren. Wurde die KPI verfehlt, können Tradeoff-Analysen der Kopplungen zwischen den Microservices durchgeführt werden. Die Auswahl der zu analysierende

[53] https://martinfowler.com/articles/microservice-trade-offs.html#consistency

[54] https://solace.com/blog/microservices-choreography-vs-orchestration/

[55] https://wirtschaftslexikon.gabler.de/definition/key-performance-indicator-kpi-52670

Kopplungen kann ebenfalls durch die Betrachtung der Core Produkte erfolgen. Die Abbildung der Quantenanzahl auf eine allgemeingültigen Skala ist nicht das Ziel der Methodik; so kann in bestimmten Fällen eine Quantenanzahl von fünf ausreichend sein, wobei bei anderen Architekturen dies ein Indikator für eine zu starken Kopplung wäre.

In *Figure 22* wurden die architektonischen Quanten definiert. Die Landschaft besteht aus vielen Embedded Systemen (unterer weißer Kasten); rechts im weißen Kasten, mit rot gekennzeichnete Komponenten, sind die Verbindungen zu Endkunden-Systeme dargestellt, wobei der lila Kasten (*Microservice1*, Links) und der hellblaue (*Microservice2*, Mitte) zwei Microservices verkörpern. Die Embedded Systeme senden kontinuierlich Messdaten, welche vom *Microservice1* verarbeitet werden. *Microservice2* bietet über einen API-Gateway eine Schnittstelle für die Endkunden-Systemen an, welche auch Daten vom Microservice1 zur Verfügung stellt.

Kernprodukt des Systems sind die erfassten Daten durch die Embedded Systeme. Der Empfang und das Speichern der Daten muss zu jeder Zeit gewährleistet sein. Um diese Anforderung zu erfüllen, wurde eine asynchrone Verarbeitung eingeführt. Ein Adapter (lila Kasten unten) empfängt die Daten und legt sie in einer Queue ab. Ziel der Architektur war es zwei getrennte Quanten zu ermöglichen und die Ausfallsicherheit zu steigern.

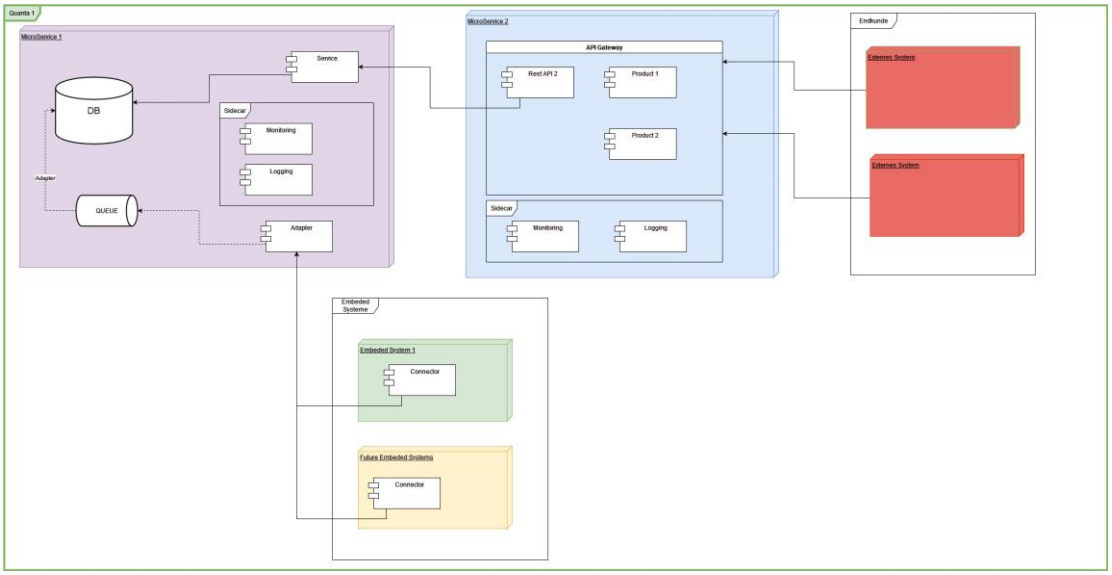

Figure 22: Architektonischen Quant Verstoß Erkennung

Im Widerspruch zu der Anforderung, erweist die Zeichnung der Quanten einen Verstoß. Nach der Betrachtung der synchronen Kommunikation der einzelnen Services, entsteht ein einziges Quant. Problematisch ist die Kommunikation zwischen den Embedded Systemen und den *Microservice1*. Eine Lösung und Trennung in unterschiedlichen Quanten bietet *Figure 23.* Der Adapter wird in einen einzelnen *Microservice3* ausgelagert.

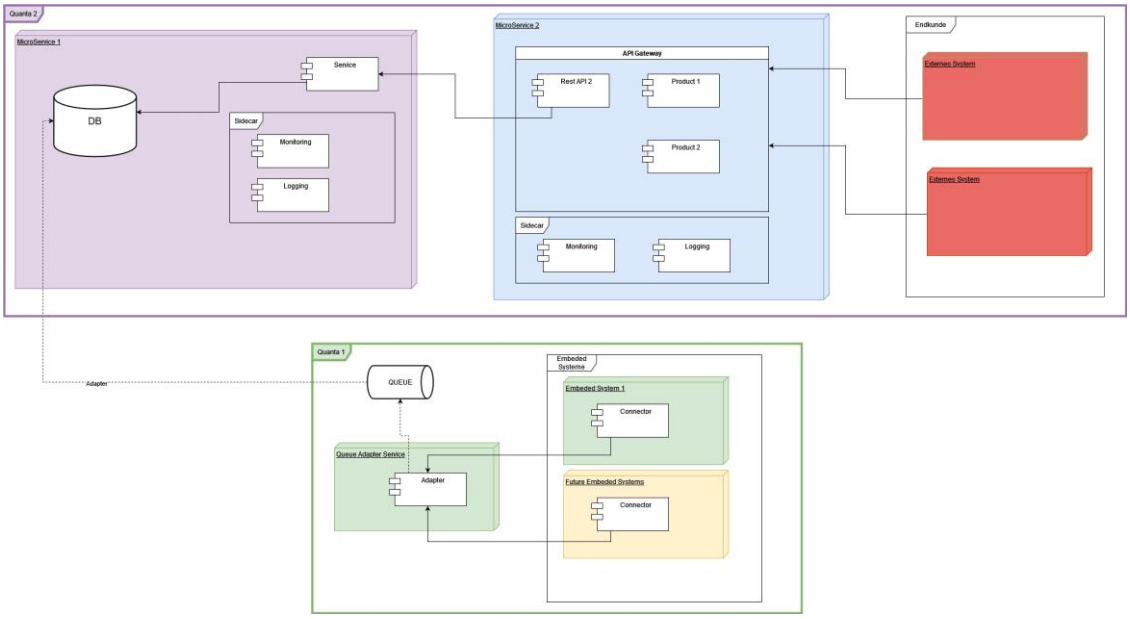

Figure 23: Architektonischen Quant Lösung

Nachteile

Die Schwäche der architektonischen Quant-Analyse ist die Erfassung der tatsächlichen Abhängigkeiten. Das im vorherigen Beispiel verwendete Diagramm reflektiert die Soll-Architektur eines Produktes. Bestehende Differenzen zwischen der Soll-Architektur und der Ist-Architektur verfälschen die Analyse und können ATD verschleiern. Zusätzlich basiert die Idee von Microservices auf Dezentralisierung, somit entstehen keine Diagramme, welche die Architektur des gesamten Systems widerspiegelt. Eine Möglichkeit den Ist-Zustand der Aufrufe zwischen den Microservices zu erfassen, bieten Tracing Tools wie Jaeger[56], welche automatisch generierte Graphen anbieten (*Figure 24*). Darüber hinaus können Abhängigkeiten über

[56] https://www.jaegertracing.io/

abgelegte *Contract Tests*[57] ermittelt werden. Das Tool PACT[58] stellt einen Brooker zur Verfügung, welcher die Verträge bereitstellt. Anhand der Zugriffe und der Bereitstellung der Verträge (engl.: contracts) kann das Tool dynamisch ein Network-Diagramm der Microservices erstellen. Diese Erfassungsmöglichkeiten machen eine Prävention unmöglich. Diagramme entstehen nach der Implementierung der Abhängigkeit und möglicherweise nach der Produktivlegung.

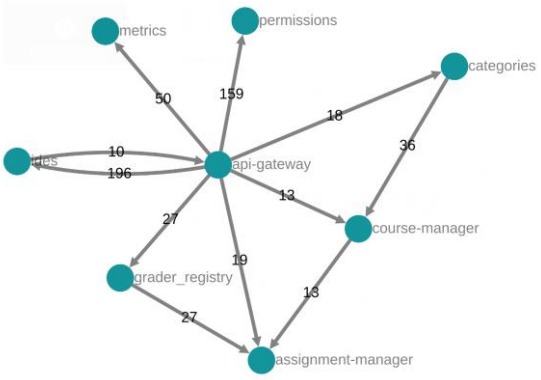

Figure 24: Jaeger Diagramm der Systemkommunikation von OPPSEE[59]

5.5.3 Verteilter Monolith

Ein verteilter Monolith ist ein System, das eine Verteilung wie ein verteiltes System vorweist, aber wie ein Monolith deployt werden muss. Alle Microservices müssen gleichzeitig deployt werden und Ausfälle einzelner Microservices führen zu gravierendem kaskadierendem Versagen des Systems. Zusätzlich erschweren verteilte Monolithen einen Überblick von Prozessabläufe zu erlangen. Indikatoren für verteilte Monolithen sind:

[57] https://pactflow.io/blog/what-is-contract-testing/

[58] https://pact.io/

[59] https://oppsee.haw-hamburg.de/

- Anforderungen an einem Microservices verursachen Änderungen an anderen Microservices.
- Microservices müssen zwingend gemeinsam deployt werden.
- Die Microservices weisen einen zu hohen Kommunikationsfluss auf.
- Einen Architektonischen Quant von eins.
- Es sind zu viele oder zu große shared Kernel[60] im System vorhanden.
- Das vermehrte Auftreten von kaskadierenden Ausfällen.

Verteilte Monolithen besitzen alle Nachteile eines verteilten Systems und eines Monolithen, ohne deren Vorteile aufzuweisen (Newman, 2021, p. 15) und stellen somit eine gravierende ATD dar. Als Beispiel eines verteilten Monolithen dient *Figure 25*, welches die Abhängigkeiten zwischen den Microservice bei airbnb[61] erfasst. Das Bild erinnert an das Antipattern „Big Ball of Mud“.

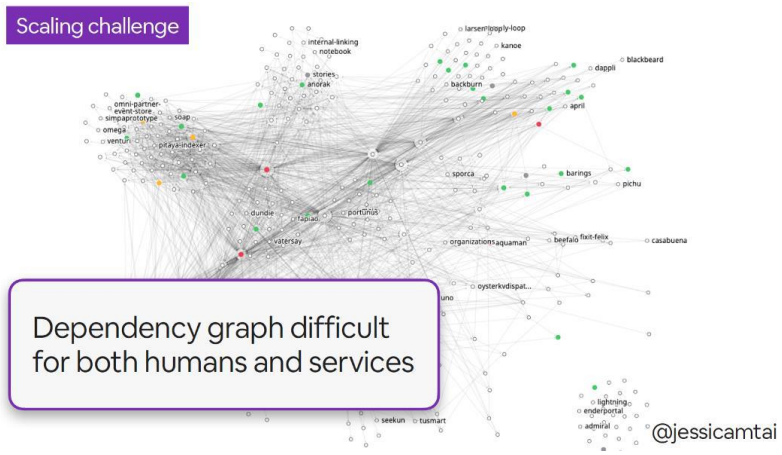

Figure 25: Abhängigkeitsgraph airbnb

Quelle: https://www.infoq.com/presentations/airbnb-culture-soa/ pp. 36

Eine Möglichkeit die stätige Erosion der Architektur zur verhindern und somit von verteilten Monolithen, ist die konstante Überprüfung der Abhängigkeiten. Eine manuelle Überprüfung

[60] Pattern in DDD, um Daten und Source-Code zu teilen.

[61] https://www.airbnb.de

ist bei der Anzahl an Services mühsam. Zusätzlich zur Begutachtung der Abhängigkeiten des Dependency Managers, ist die Etablierung einer Kommunikation über API-Gateways hilfreich (*Figure 26*).

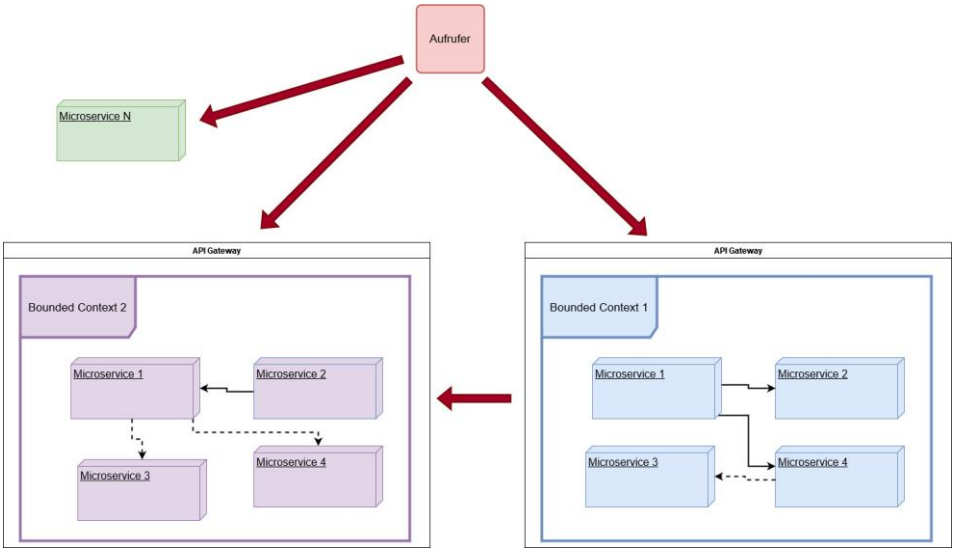

Figure 26: API-Gateway Kommunikation

Dieses Pattern enthält folgende Voraussetzungen und Regeln:

- Jeder Bounded-Context wird durch einen API-Gateway gekapselt.
- Synchrone Kommunikation zwischen Bounded-Contexts laufen über das API-Gateway.
    - Ausnahmen müssen als ADR dokumentiert und in der Whitelist ergänzt werden
- Asynchrone Aufrufe stärken die Entkopplung und müssen nicht über ein zentrales API-Gateway durchgeführt werden.
- Die Kommunikation zwischen Microservices in einem Bounded-Context kann über das API-Gateway stattfinden.

In *Figure 26* sind die Bounded-Context voneinander und für jegliche Aufrufer durch die API-Gateways isoliert. Der *MicroserviceN* verletzt gegen die Einrichtung von einem API-Gateway, welches auf einen Fehler beim Design des Microservice hinweist. Der Microservice ist keinem Bounded-Context zugewiesen. Handelt es sich um einen Microservice mit

Infrastrukturtätigkeiten, zum Beispiel logging Aufgaben, kann der fehlende Bounded-Context gewollt sein. Die Einhaltung der Kommunikationsregeln kann anhand von FF sichergestellt werden. Zum Beispiel können Architekturtests in der CD-Pipeline überprüfen, welche URLs aufgerufen werden oder welche Discovery Services angesprochen werden. Zusätzlich kann die Bildung von Netzwerken die Microservices nach außen isolieren und nur den Zugriff über das API-Gateway erlauben. Vorteile des Patterns ist die Einhaltung von Abhängigkeitsregeln zwischen Bounded-Contexts. Die Teams entscheiden, welche Operation nach außen sichtbar sind, somit können ungewünschte Abhängigkeiten zu einem internen Microservice nicht entstehen. Außerdem besteht die Möglichkeit auf Makroebene zu deklarieren, welche Bounded-Contexts miteinander kommunizieren dürfen. Ist der Aufruf von *Bounded-Context1* durch *Bounded-Context2* nicht erlaubt, ist dieses erneut durch Architekturtests überprüfbar.

Nachteile

Die Schwächen dieses Muster ist die Bildung von Hierarchien, welche durch die Einführung von Abhängigkeitsregeln zwischen Bounded-Contexts entstehen. Microservice-Organisationen basieren auf autonomen Teams, welche nach sozialen Netzwerken geschnitten werden und bilden einen Gegensatz zu Hierarchien.

5.5.4 Source-Code-Analyse

Während der Recherche wurde kein angemessenes Source-Code-Analysetool gefunden, um TD auf Makroebene zufriedenstellend zu identifizieren. Die Hauptprobleme entstehen durch die Verteilung und Entkopplung des Source-Codes auf mehrere Repositories; einfache Compiler Funktionalitäten können nicht mehr verwendet werden.

Figure 27: Codescene Architektur Menu

Eine erste Annäherung, um die Komplexität, die Größe, die verwendete Sprache, den architektonischen Zustand (*Figure 27*) oder die Dora Metriken über mehrere Repositories anzuzeigen, bietet das Tool Codescene[62]. *Figure*

[62] https://codescene.com/

28 spiegelt den Gesundheitszustand durch Hotspots eines Microservices-Systems wider, bestehend aus fünf Microservices und einem Discovery Service. Codescene analysiert die Git-Historie, um Kopplungen festzustellen. Größe und Komplexität der Module wird jeweils anhand der Größe und Farbe der Kreise dargestellt. Codescene bietet eine initiale Konfiguration, welche durch Regeln erweitert werden kann.

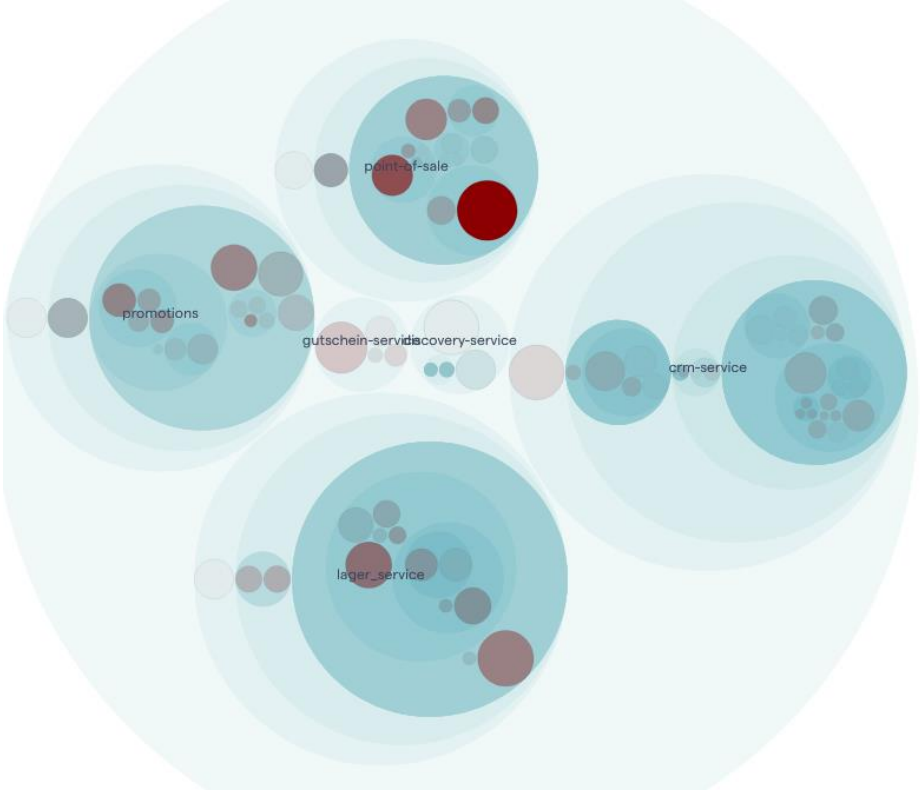

Figure 28: Source-Code Health Darstellung

5.5.5 Mustersprachen

Im Unterkapitel *5.4* wurde die Schwierigkeit der Definition von Mustersprachen für das gesamte System durch die erhöhten Variationsmöglichkeiten behandelt. Zusätzlich zur Abdeckung aller möglichen Programmiersprachen, wirken Definitionen in der Makroarchitektur gegen die Idee von autonomen Teams. Neue Mustersprachenregeln erfordern hohe Koordination in der gesamten Organisation und verlangsamen somit die Vorlaufzeit (TTM), welches eines

der primären Ziele von Microservices-Systemen ist. Mustersprachen weisen Parallelen zur Ubiquitous Languages auf: allgegenwärtige Definition von Konzepten. Die Überlappung kann die Einführung von Mustersprachen für ein Bounded-Context vereinfachen. In *Figure 18* wurde das Konzept einer Bounded-Context-Architektur eingeführt. Die exklusive Zuständigkeit von einem Team zu einen Bounded-Context vermeiden den Koordinationsaufwand bei der Deklaration oder Anpassung von Mustersprachen, welches zu deren Einführung führt. Die Überprüfung wird in die CD-Pipeline mit eingebaut (*Figure 29*: Schritt Architektur-Test). Allgemeine Tests können als shared-Library implementiert werden, welche jeweils über den Dependency Manager oder in der Pipeline Deklaration geladen werden (*Figure 30*).

Figure 29: CD-Pipeline mit Architektur-Test Schritt

```
libraries {
    lib('Mustersprachen-tests-Java-wrapper'),
    lib('Mustersprachen-tests-Python-wrapper')
}
```

Figure 30: CD-Pipeline Libraries

5.6 Bewertung der Microservices-Architektur

Obwohl Microservices viele Ansätze bieten, um positiv die Hauptziele eines langlebigen Systems zu beeinflussen, bringen sie neue Herausforderungen mit sich. Insbesondere leidet die Wartung des Systems. Die Modularisierung und die Granularität reduzieren den Wartungsaufwand einzelner Microservices. Anderseits wird das Ausmachen angemessener Kopplung und die Erkennung von komplexeren und verteilten TD erschwert. Die versteckten Beziehungen der einzelnen Microservices zueinander können ohne geeignete Lösungen nicht nachvollzogen werden und erschweren das Verständnis des Gesamtsystems. Die Stabilität des Systems kann

dank der Ausfallsicherheit, der schnelleren Skalierung und des angemessenen Schneidens der Services erheblich verbessert werden. Das vereinfachte Hinzufügen von neuen Microservices bietet eine starke Anpassungsfähigkeit des Systems, welches die Herausforderung bei der Orchestrierung und Kopplung der Services erhöht und ATD generiert. *Tabelle 14* zeigt den vorherigen beschriebenen Einfluss der Microservices-Architektur auf die Systemziele.

Stabilität	**Geringe Wartungskosten**	**Anpassungsfähigkeit**
Modularisierung ↑	Modularisierung ↑	Modularisierung ↑
Ausfallsicherheit ↑	TD-Managementzyklus ↓	TD-Managementzyklus, Abbau von absichtliche umsichtige TD =
Fehlerreduzierung =	Angemessene Kopplung ↓	Angemessene Kopplung ↓
Qualitätsfunktionen =	Qualitätsfunktionen =	

Tabelle 14: Einfluss von Microservices auf die Beziehung zwischen Methodiken und Systemziele

Die Vielzahl an verwendeten Technologien und die steigende Zahl an Services erfordern hohe technische Expertise, starke Automatisierung und einen reifen Entwicklungsprozess. Die Voraussetzungen reduzieren die Anzahl an einfachen TD, lösen Bottlenecks im Auslieferungsprozess auf, begrenzen die kognitive Belastung und verschieben den Fokus auf ATDs und SPDs. In der *ATD Checkliste* im Anhang A.3 werden Indikatoren und mögliche ATD aufgelistet, die beobachtet werden sollten. Jene Lösungen, deren Implementierung die Einführung von Hierarchien auf die Gesamtsystemebene begünstigen, wirken gegen das Prinzip von autonomen Teams. Die Skalierung von Microservices sollten nach Dunbars Zahl[63] erfolgen (Matthew Skelton and Manuel Pais, 2019, p. 34). Microservices verwenden die Kommunikationsart von sozialen Netzwerken, um deren Wachstumsrate zu erreichen und stehen im Kontrast zu Hierarchien. Insgesamt eignet sich die Microservices-Architektur als langlebige evolutionäre Architektur mit reduzierten TD auf Mikroebene, aber mit dem Potenzial vieler komplexer ATDs auf abstrakter Ebene zu verbergen und erfordert die Bewältigung von SPDs.

[63] https://lexikon.stangl.eu/12337/dunbar-zahl

6 Zukunftsfähigkeit von Microservices

In diesem Kapitel wird anhand von Erfahrungen in der Industrie und den gewonnenen Erkenntnissen dieser Arbeit die Zukunftsfähigkeit von Microservices bewertet.

Microservices bieten eine hohe Wachstumsrate an. Neue Projekte können die entstandenen Produkte in kürzester Zeit an den Markt bringen und möglichen exponentiellen Wachstum durch Skalierung bewältigen. Zusätzlich schaffen Microservices mehr Spielraum für technische Experimente. Jedoch werden die Nachteile nach einem längeren Zeitraum deutlicher. Die Erhöhung der Anzahl komplexer SPDs und die daraus anfallenden Zinskosten können ein operatives Risiko darstellen. Microservices eignet sich für Systeme, die einen hohes Wachstum erwarten (scalability und elasticity), eine starke Verteilung der Services aufweisen (geografisch und logisch), den Fokus auf entkoppelte Services legen und über genügend Ressourcen im Bereich IT verfügen. IT-Teams können in vier Unterschiedliche Typen zerlegt werden (Matthew Skelton and Manuel Pais, 2019, pp. 79-94):

- Stream-aligned Team: Teams, die nach einem wertschöpfenden Produkt, Service oder Feature ausgerichtet sind.
- Enabling Team: Kollaborative und cross-cutting Teams, welche die Recherche Tätigkeiten übernehmen und die Expertise im Unternehmen verbreiten, um somit Stream-aligned Teams zu entlasten.
- Complicated-subsystem Team: Verantwortlich für die Entwicklung und Wartung von Systemen, die stark spezialisiertes Wissen benötigt.
- Plattform Team: Zuständig für die Bereitstellung von Tools und Services, um die Autonomie von Stream-aligned Teams zu fördern.

Die Ausrichtung der Teams zeigt die Anzahl an notwendigen Teams für die erfolgreiche Umsetzung von Microservices-Architekturen. Die höhere Anzahl an Technologien erfordern mehr Enabling und Plattform Teams; die Zuordnungen eines Bounded-Contexts zu einem Stream-aligned Team erhöhen ebenfalls die finale Teamanzahl.

Der zu erreichende oder erwartete Wachstum muss strategisch beurteilt werden. Dabei muss das Unternehmen identifizieren welche Art von Wachstum angestrebt wird; economy of scale (dt.: Skaleneffekt) oder super-linear growth (dt.: superlineares Wachstum) (*Figure 31*). Der Skaleneffekt beschreibt die Abhängigkeit der Produktionsmenge im Verhältnis mit den eingesetzten Produktionsfaktoren, wobei eine höhere Massenproduktion die Stückkosten senkt (Silberston, 1972). Hierarchische Unternehmen bedienen sich dem Skaleneffekt, um die Kosten von neuen Mitarbeitern unter dem Verhältnis von 1-zu-1 zu senken, welches den gleichen Effekt beim Gewinnwachstum aufweist (Lewis, 2019). Im Gegensatz beschreibt superlineares Wachstum, ein Wachstum mit einer proportionalen Rate zur Eingabe T, wobei T die Anzahl neuer Mitarbeiter repräsentiert und ein Gewinnschöpfungsverhältnis von größer als eins ermöglicht. Auf sozialen Netzen basierende Strukturen sind der Treiber für superlineare Wachstum (Lewis, 2019).

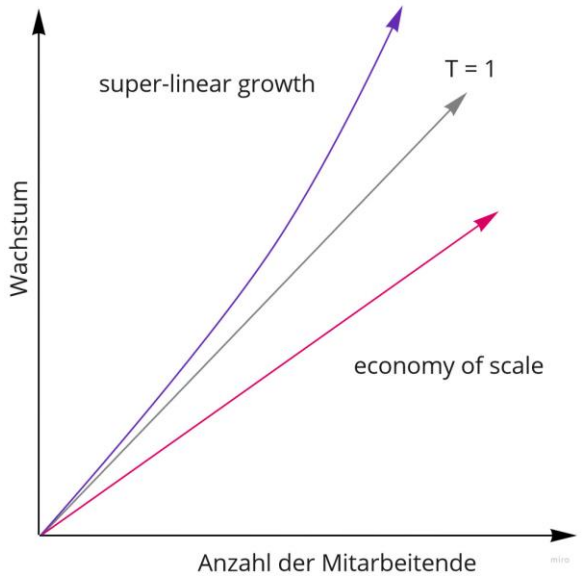

Figure 31: economy of scale vs super-linear growth

Das superlineare Wachstum von Microservices Organisationen, die anhand von Dunbars Nummern skalieren, erhöht die Einnahmen, wie im Fall von Amazon (Lewis, 2019), sowie die Kosten, die anhand der Zinsen der SPDs und der höhere Mitarbeiteranzahl widergespiegelt wird.

Die Zukunftsfähigkeit von Microservices kann nicht ausschließlich aus technischer Sicht entschieden werden, sondern erfordert ebenfalls eine Betrachtung aus strategischer ökonomischer

Sicht. Unternehmen, die einen superlinearen Gewinnwachstum und einen Skaleneffekt in den Kosten erreichen möchten, riskieren das Antipattern verteilter Monolith zu etablieren. Wichtig ist die Findung der angestrebten Wachstumsstrategie und eine Optimierung des Gewinn-Kosten-Koeffizienten.

Eine Möglichkeit eine economy of scale Wachstumsstrategie mit geringeren Kosten zu erreichen, zeigt die Architektur-Trend-Analyse von InfoQ von 2022 (*Figure 32*). Der „*Modular monolith*“ (Modulithen) erregt immer mehr Aufmerksamkeit. Dieser stark modularisierte Monolith weist Parallelen zur Microservices-Architektur auf. Eine positive Vermischung der Microservices- und Modulithen-Architektur könnte der Service Modulith darstellen (Unterkapitel 6.1), welches anhand des nicht vorhandenen Enterprise Service Bus (ESB) von der Serviceorienteierte Architektur (SOA) abgrenzt wird und als ein Muster für ein System gilt und nicht für eine gesamte IT-Landschaft.

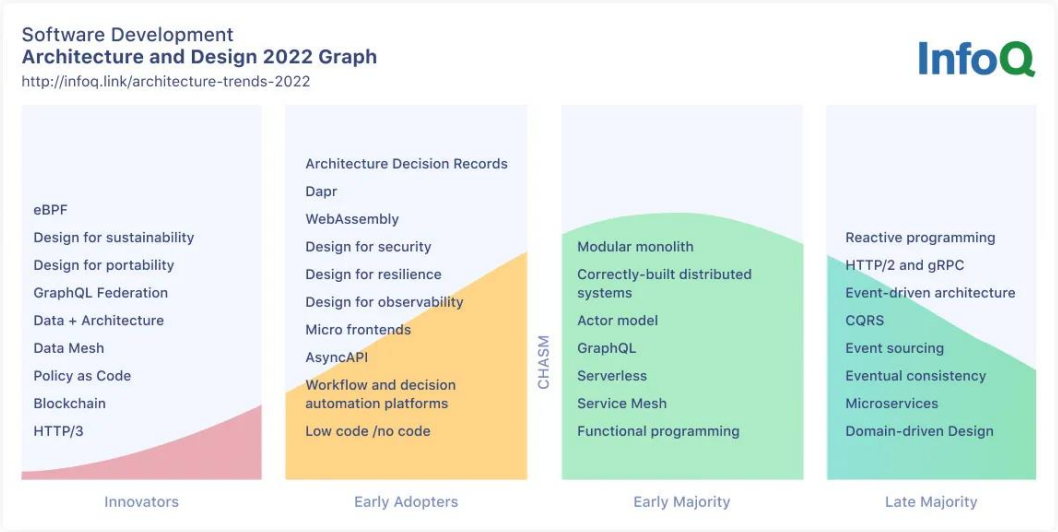

Figure 32: Architektur-Trend-Analyse Q1 2022

6.1 Service Modulith

Der Service Modulith verfolgt die Modularisierung des Monolithen, die Vermeidung der Nachteile von verteilten Systemen, die Reduzierung von Wartungskosten und eine ausgeprägte Anpassungsfähigkeit. *Figure 33* zeigt dessen Aufbau. Das System wird, unter Berücksichtigung

der erarbeiteten Punkte, im Abschnitt 3.1 in fachliche Module zerteilt. Eine Zerteilung anhand des Bounded-Contexts ist ratsam. Jedes Modul deklariert eine externe API, worüber das Modul von außen angesprochen werden kann. Die externe API wird als separates Projekt (Artefakt) deklariert. Aufrufer kennen nur das API-Artefakt, wobei das Modul die Implementierung zur API anbietet. Die Trennung der einzelnen Module und APIs in eigenständige Repositories ermöglich eine höhere Parallelität bei der Entwicklung. Jedes Projekt verfügt über eine CD-Pipeline, welches sich aus den Erkenntnissen von Microservices bedient und Contract-Testing als Integrationstest durchführt. Am Ende einer Pipeline entsteht ein Artefakt, welches deployt werden kann und kürzere Release-Zyklen bietet. Zum Beispiel kann die vertrag.jar durch eine neuere Version ersetzt werden. Das Hochfahren eines neuen Containers mit dem gesamten Modulithen und dem neuen Modul wäre eine Möglichkeit, wenn die Startzeit nicht zu hoch ist.

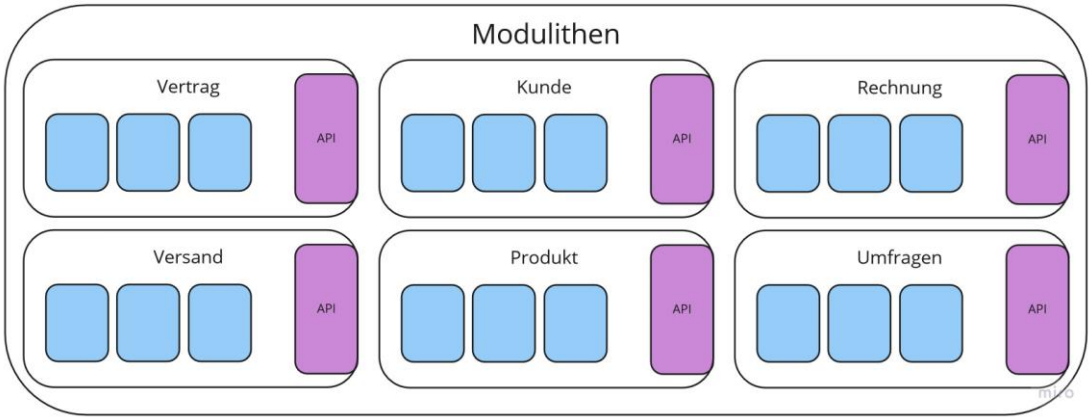

Figure 33: Komponentensicht eines Service Modulithen

Module können in Teilmodule untergliedert werden. Diese sind aber nicht einzeln deploybar. Werden Teilmodule zu groß, muss eine erneute Modellierungsanalyse stattfinden und möglicherweise eine Trennung der Module stattfinden. Die Einführung und Überprüfung von Mustersprachen wie Architekturregeln, können anhand der bekannten Tools umgesetzt werden. Ein Kernbestandteil des Architekturmusters ist das API-Projekt. Das Design der API muss folgende Prinzipien erfüllen:

- Geschäftsbezogen: Die API darf keine Implementierungsdetails beinhalten und rein fachliche Spezifikationen zur Verfügung stellen
- Kontextfrei (Stateless): Eine Annahme über den Transaktionskontext ist nicht erlaubt. Operationen dürfen keine Aufrufreihenfolge voraussetzen.
- Idempotent: Jede Operation sollte beliebig oft aufrufbar sein.
- Kompensierbar: Jede verändernde Operation besitzt ein Gegenstück.
- Grobgranular: Eine API sollte so abstrakt wie möglich sein und so detailliert wie nötig.

Das Schneiden und die Wartung der Module erweisen sich, analog zu der Microservices-Architektur, als überschaubar. Die Komplexität entsteht bei der Haltung der Daten und der Interaktion zwischen den Modulen.

Service Modulithen erlauben die Verwendung mehrerer DB (*Figure 34*), eingeschränkt durch die Richtlinie: Nur ein Datenbankprodukt je Datenbanktyp. Antikorruption-Layer sind zwischen den Modulen möglich und eine Redundanz der Daten ist in angemessenem Umfang erlaubt. Die Redundanz sollte nicht zu einzelnen Schemata je Modul führen, die alle Daten anderer Module beinhalten, sondern Ergänzungen, welche nur für den Bounded-Context relevant sind. Problematisch ist die Konsistenz der Daten über mehrere DB. Erneut können Muster aus der Microservices-Architektur verwendet werden. Eine asynchrone Verarbeitung der Daten ist möglich oder ein Rollback durch eine Programmatische Transaktionsklammer, die in einem Modulithen Vorteile besitzt. Die Transaktion findet in einem Prozess statt, wobei die Kompensationsaktionen nicht auf Netzwerkprobleme stoßen werden. Dabei kann die Transaktion den Two Phase Commit[64] Pattern oder SAGA[65] Pattern verwenden.

[64] https://martinfowler.com/articles/patterns-of-distributed-systems/two-phase-commit.html

[65] https://microservices.io/patterns/data/saga.html

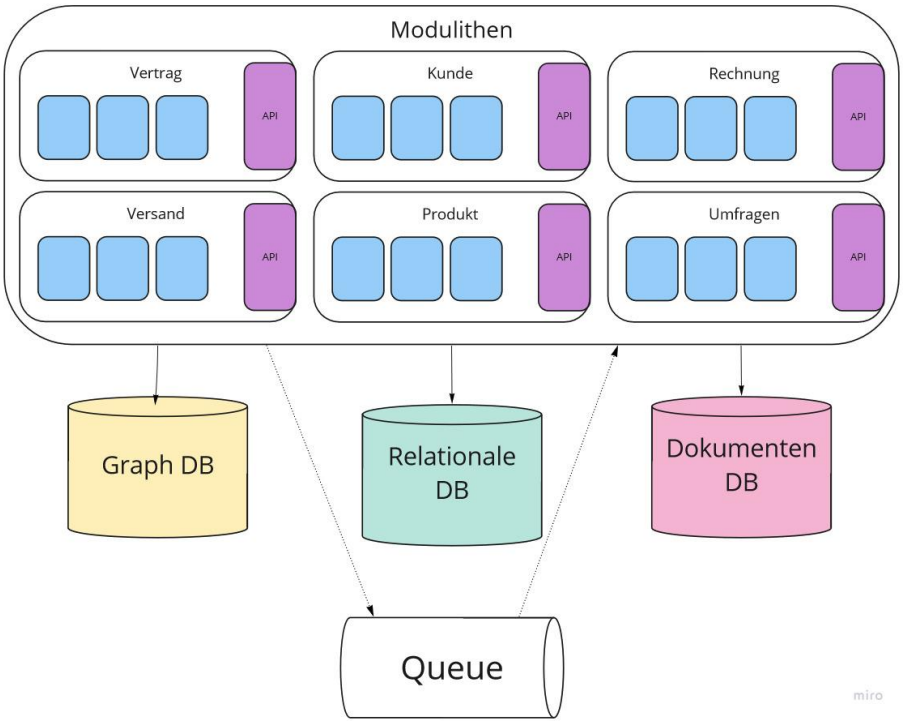

Figure 34: Gesamtarchitektur eines Service Modulithen

Die Kommunikation zwischen den Modulen kann synchron oder asynchron erfolgen. Synchrone Aufrufe können mit Hilfe vom Injektion Pattern und einen Service-Container per Method-Invocation stattfinden. Operationen, die keine Synchronität benötigen, können anhand Event- oder Message-Queues propagiert werden (*Figure 34*). Erneut kann das gewonnene Wissen aus der Microservices-Architektur für das Handling von Messages und Events hilfreich sein. Die Einrichtung von Error-Queues oder Monitoringkonzepte sind Beispiele davon. Zusätzliche Skalierung kann durch die Anpassung des CQRS-Patterns[66] und unterschiedliche Konfiguration der Modulithen erfolgen. Reine Modulithen für die asynchrone Verarbeitung können schneller, durch die reduzierte Initialisierungszeit, bereitgestellt werden.

Die Infrastruktur ist eines der komplexeren Aspekte einer Microservice Architektur, die Einführung flächendeckendes Monitoring, Logging oder Tracing erwies sich als aufwändig. Bei einen Modulithen verwenden alle Komponenten die gleiche Infrastruktur, welches die

[66] https://martinfowler.com/bliki/CQRS.html

Einführung der oben genannten Konzepte vereinfacht. Tracing wird im synchronen Bereich durch Stacktrace oder Stackwalk deutlich vereinfacht. Asynchron kann analog zu Microservices eine Tracing ID mitgereicht werden. Dieses Mechanismus kann durch einen Wrapper verschleiert werden und erfordert keine mehrfache Implementierung. Das gleiche Wrapper Pattern kann fürs Monitoring oder Tracing verwendet werden.

Die Integration des UI stellt die nächste Herausforderung. Die Modularisierung der Komponenten und deren separates Deployment kann durch eine zu stark gekoppeltes UI beeinträchtig werden. Eberhard Wollf definiert eine Anzahl an Integrations- und Kommunikationspattern für Microservices: Single-Page-APP (SPA) per Modul oder für das gesamte System, HTML-Anwendungen per Link Aufrufe, Frontend Server, Mobile Clients oder Rich Clients (Wolff, 2018, pp. 167-179). Die Bereitgestellte API kann lokal, im Backend selbst, per Method-Invocation kommunizieren, wobei die Kommunikation mit der UI über Remote Procedure Call (RPC) oder Web-Calls stattfinden kann. Wichtig ist die Sicherstellung, dass keine synchronen Aufrufe zwischen zwei Modulithen stattfindet.

Nachteile

Im Gegensatz zu Microservices können nur größere Pakete unabhängig deployt werden, zugleich bietet das Architekturmuster eine höhere Flexibilität als Monolithen. Die größere Anwendung und die niedrigere Zahl an DB führen zu einer niedrigeren Skalierung. Daten können sinnhaft in unterschiedliche DB-Typen zerteilt werden, diese müssen aber Skalierungsmöglichkeiten bieten. Im Allgemeinen platziert sich das Architekturmuster im Bereich der Komplexität und Flexibilität zwischen den Monolithen und die Microservices-Architektur. Eine Übersicht der Bewertung der Qualitäten liefert die nach Richards und Ford (Mark Richards & Neal Ford, 2020) erweiterte *Tabelle 15*. Die Bewertung erfolgt von eins (niedrige Erfüllbarkeit) bis zu fünf (komplette Erfüllbarkeit).

Qualität	**Schichten Monolith**	**Microservices**	**Service Modulith**
Deployability	1	4	3
Elasticity	1	5	3
Evolutionary	1	5	3

Fault Tolerance	1	4	2
Modularity	1	5	4
Overall Cost	5	1	3
Performance	2	2	4
Reliability	3	4	4
Scalability	1	5	4
Simplicity	5	1	3
Testabiliy	2	4	3

Tabelle 15: Qualitätsbewertung von Architekturmuster

7 Fazit

In diesem Kapitel werden die Inhalte der Bachelorarbeit zusammengefasst. In dem Unterkapitel „Ausblick“ werden Aspekte genannt, die bei der Bearbeitung und Beurteilung der Microservices-Architektur offengeblieben sind und mit denen das Ergebnis dieser Arbeit fortgeführt werden kann.

7.1 Zusammenfassung

Das Ziel dieser Arbeit war es, Methodiken für die Erkennung, Kategorisierung und Gewichtung von TD herauszuarbeiten, die den aktuellen Stand der Industrie widerspiegelt und Defizite hervorhebt, beziehungsweise ergänzt. Ein untergeordnetes Ziel war die Evaluierung der Microservices-Architektur anhand der Eigenschaften Langlebigkeit und Evolutionär. Die Ergebnisse der Forschung zeigen die Notwendigkeit der Verwendung mehrerer Methodiken für die

Identifikation von TD. Komplexere TD können durch einfache Source-Code-Analyse nicht identifiziert werden und eine vorläufige Ontologie der TD ist vorhanden. Aus dem Befund der Ursachen-Analyse von TD kann daraus geschlossen werden, dass das Konzept von TD bei der Erklärung von diversen Missständen an seine Grenzen gestoßen ist. Die Erweiterung von TD durch das Konstrukt von SPD ist notwendig, um auftretende Zinsen in den Bereichen der Organisation und Fachlichkeit zu veranschaulichen.

Weitere Ergebnisse zeigen die Relevanz der kognitiven Psychologie für das Beibehalten einer verständlichen Architektur. Evolutionäre und langlebige Architekturen implizieren eine gute Architektur; so stellte sich heraus, dass die Einhaltung dieser Eigenschaften erheblich die Stabilität, die Wartungskosten und die Anpassungsfähigkeit positiv beeinflussen. Darüber hinaus ist ein Schutz der wichtigsten Architektureigenschaften anhand von FF relevant.

Durch die entstandene Checkliste *Angemessene Architektur* wurde eine Bewertung der Microservices-Architektur möglich. Das Ergebnis zeigt eine Verbesserung der Stabilität und der Anpassungsfähigkeit. Andererseits leidet die Verständlichkeit des Systems, welches eine Reduzierung der Wartbarkeit und steigende Wartungskosten verursacht. Zusätzlich wird der Fokus von TD im Source-Code auf die Interaktion zwischen den Microservices und ATD verlagert. Aufgrund der Deklaration der wichtigsten Komponenten im System, können die unwirtschaftlichsten ATD identifiziert werden, welches der Erosion der Architektur entgegenwirkt. Als Ergebnis der Forschung für das Entgegenwirken von ATD, wurden die Methodiken „*Architektonischer Quant*“ und diverse Muster auf Bounded-Context Ebene erfasst.

Die hohe Technologische Komplexität von Microservices führte zum Ergebnis, dass nur Organisationen mit einem etablierten und reifen Entwicklungsprozess sowie genügend qualifizierten Ressourcen Microservices-Architekturen umsetzen sollten. Die Betrachtung der angestrebten Wachstums- und Kostenstrategie ist die wichtigste Entscheidung für oder gegen das Verwenden der Microservices-Architektur. Nachgelagert ist die Technische Analyse, welches Problem Microservices beheben sollen, wichtig. Eine mögliche Alternative bildet der Modulith und konkreter der Service Modulith.

7.2 Ausblick

Die in dieser Arbeit erarbeiteten Konzepte und Muster wurden zum Großenteil an realen Situationen erprobt. Deren Einführung und langfristige Beobachtung war in der Kürze der Bearbeitungszeit nicht möglich. Es bleibt offen, ob die erarbeiteten Ideen in komplexere Systeme adäquat für einen langfristigen Einsatz sind. Die Erarbeitung der SPD-Metapher bietet eine neue Möglichkeit, Missstände zu platzieren. Interessant wäre die zukünftige Diskussion der Grenzen von TD und die Veranschaulichung dieser.

Die wissenschaftliche Arbeit zeigte die Wichtigkeit von ATD in Microservices-Architekturen. Es bleibt die Frage offen, inwiefern die erarbeiteten Checklisten durch die Bereiche Security und Plattform ergänzt werden können und ob weitere Techniken für deren Erkennung vorhanden sind. Hieraus ergibt sich die Möglichkeit, in einer zukünftigen Studie zu untersuchen, inwiefern der Aufbau eines ATD-index bei einer Microservices-Architektur hilfreich ist, wenn die Gesamtbetrachtung der Architektur ein Hindernis darstellt.

Weiterhin wäre eine Untersuchung interessant, inwiefern Unternehmen mit Microservices-Architekturen oder Service Modulithen erfolgreicher sind sowie eine detaillierte Gegenüberstellung beider Konzepte.

Literaturverzeichnis

Anderson, J. R., 1980. *John R. Anderson.* 3 Hrsg. s.l.:Spektrum Akademischer Verla.

Anon., 2021. *Reducing Incidents in Microservices by Repaying Architectural Technical Debt.* [Online]
Available at: https://ieeexplore.ieee.org/abstract/document/9582573

Antonio Martini, Jan Bosch, Michel Chaudron, 2014. Architecture Technical Debt: Understanding Causes and a Qualitative Model. *40th Euromicro Conference on Software Engineering and Advanced Applications*, August.

Casey Rosenthal, Nora Jones, 2020. *Chaos Engineering: System Resiliency in Practice*. s.l.:O'Reilly Media, Inc, USA.

CISM, kein Datum *Dynamic Analysis vs. Static Analysis.* [Online] Available at: https://www.cism.ucl.ac.be/Services/Formations/ICS/ics_2013.0.028/inspector_xe/documentation/en/help/GUID-E901AB30-1590-4706-94B1-9CD4736D8D2D.htm
[Zugriff am 29 06 2022].

Darwin, C., 2008. *Die Entstehung der Arten.* s.l.:Nikol.

Fowler, M., 2006. *CodeSmell.* [Online] Available at: https://martinfowler.com/bliki/CodeSmell.html
[Zugriff am 30 07 2022].

Fowler, M., 2015. *Martin Fowler: Yagni.* [Online] Available at: https://martinfowler.com/bliki/Yagni.html
[Zugriff am 29 03 2022].

Fowler, M., 2019. *Software Architecture Guide.* [Online] Available at: https://martinfowler.com/architecture/ [Zugriff am 06 03 2022].

Fowler, M., 2021. *Microservices Guide.* [Online] Available at: https://martinfowler.com/microservices/ [Zugriff am 2021].

Hirschmeier, S., 2021. *Intensivkurs Softwarearchitektur.* Hamburg: OOSE.

INNOQ, kein Datum *Independent Systems Architecture.* [Online] Available at: https://isa-principles.org/ [Zugriff am 20 04 2022].

Irakli Nadareishvili, Ronnie Mitra, Matt Mclarty, Mike Amundsen, 2016. *Microservice Architecture: Aligning Principles, Practices, and Culture.* s.l.:O'Reilly Media, Inc, USA.

James Lewis and Martin Fowler, 2014. *Microservices.* [Online] Available at: https://martinfowler.com/articles/microservices.html [Zugriff am 29 März 2022].

Kruchten, P., Nord, R. L., and Ozkaya, I., 2012. Technical: From Metaphor to Theory and Practice. *IEEE Software,* Issue 26.

Kruchten, P., 2004. *Der Rational Unified Process. Eine Einführung.* 3 Hrsg. s.l.:Addison-Wesley.

Lewis, J., 2019. *Keynote: Software Architecture, Team Topologies and Complexity Science.* [Online]
Available at: https://www.youtube.com/watch?v=uAwJEFLJunk [Zugriff am 07 08 2022].

Lilienthal, C., 2019. *Langlebige Software-Architekturen: Technische Schulden analysieren, begrenzen und abbauen.* s.l.:dpunkt.verlag GmbH.

Maarten van Steen, Andrew S. Tanenbaum, 2018. *Distributed Systems.* s.l.:Maarten van Steen.

Mark Richards & Neal Ford, 2020. *Fundamentals of Software Architecture: An Engineering Approach.* s.l.:O'REILLY.

Matthew Skelton and Manuel Pais, 2019. *Team Topologies.* Portland: IT Revolution.

Michele Lanza, Radu Marinescu, 2006. *Object-Oriented Metrics in Practice.* 1. Hrsg. s.l.:Springer.

N. S. R. Alves, L. F. Ribeiro, V. Caires, T. S. Mendes and R. O. Spínola, 2014. *Towards an Ontology of Terms on Technical Debt.* Sixth International Workshop on Managing Technical Debt: s.n.

NATO SCIENCE COMMITTEE, 1969. *SOFTWARE ENGINEERING TECHNIQUES.* Rom, s.n.

Neal Ford & Mark Richards, 2021. *Software Architecture: The Hard Parts: Modern Tradeoff Analysis for Distributed Architectures.* s.l.:O'Reilly Media, Inc, USA.

Neal Ford, Rebecca Parsons, Patrick Kua, 2017. *Building Evolutionary Architectures.* USA: O'Reilly Media, Inc, USA.

Newman, S., 2015. *Building microservices.* Beijing : O'Reilly .

Newman, S., 2021. *Vom Monolithen zu Microservices : Patterns, um bestehende Systeme Schritt für Schritt umzugestalten.* s.l.:O'Reilly .

Parnas, D., 1972. *On the Criteria To Be,* Carnegie-Mellon University: Association for Computing Machinery.

Parnas, D. L., 1972. *A technique for software module specification,* s.l.: s.n.

Parsons, R., 2021. *Episode 77 - Rebecca Parsons about Evolutionary Architecture (Software Architektur im Stream)* [Interview] (28 September 2021).

Prabath Siriwardena, Nuwan Dias, 2020. *Microservices Security in Action.* s.l.:MANNING PUBN.

Richardson, C., 2018. *Microservices Patterns: With examples in Java.* s.l.:Manning.

Saulo S.de Toledo, Antonio Martini, Dag I.K.Sjøberg, 2021. *Identifying architectural technical debt, principal, and interest in microservices: A multiple-case study,* s.l.: Journal of Systems and Software.

Sebastian Klepper, Bernd Bruegge, 2018. Impact of Hypothesis-Driven Development on Efectiveness,. *Gesellschaft für Informatik*.

Silberston, A., 1972. Economies of Scale in Theory and Practice. *The Economic Journal,* Issue 82.

Spitz, D., 2021. *IT-P*. [Online] Available at: https://www.it-p.de/wardley-mapping-visualisierung-als-strategisches-werkzeug/
[Zugriff am 02 08 2022].

Starke, G., 2020. *Effektive Softwarearchitekturen: Ein praktischer Leitfaden.* s.l.:Carl Hanser Verlag GmbH & Co. KG.

Swamidass, P., 2000. *Encyclopedia of Production and Manufacturing Management.* s.l.:Springer, Boston, MA.

Toth, S., 2015. *entwickler.de.* [Online] Available at: https://entwickler.de/software-architektur/agil-lean-oder-einfach-nur-komplex [Zugriff am 05 08 2022].

Toth, S., 2019. *Vorgehensmuster für Softwarearchitektur.* 3 Hrsg. s.l.:Hanser.

Toth, S., 2022. *Folge 128 - Agilität und Architektur mit Stefan Toth (Software Architektur im Stream)* [Interview] (22 07 2022).

Vernon, V., 2017. *Domain-Driven Design kompakt.* Heidelberg: dpunkt.verlag .

Wardley, S., 2018. *medium.* [Online] Available at: https://medium.com/wardleymaps [Zugriff am 05 08 2022].

Wolff, E., 2018. *Das Microservices-Praxisbuch.* s.l.:dpunkt.verlag.

Wolff, E., 2018. *Microservices Grundlagen flexibler Softwarearchitekturen.* s.l.:dpunkt.verlag.

Wolff, E., 2021. *software-architektur.tv*, s.l.: s.n.

Zazworka, N., Vetro', A., Izurieta, C., 2013. Comparing four approaches for technical debt identification. *Software Quality Journal*, 03 April, pp. 10.1007/s11219-013-9200-8.

A Anhang: Checklisten

A.1 Checklsite Angemessene Architektur

- ☐ Softwarearchitektur-Prozess
  - ☐ Architektur Dokumentationen (Bsp. arc42)
  - ☐ Technologiebeurteilung
  - ☐ Architekturüberprüfung
  - ☐ Architekturziele
- ☐ Modularisierung
  - ☐ Definition von Modulen
  - ☐ Definition von Modulen in Modulen
  - ☐ Definition der Interaktionsregeln zwischen Modulen (API)
  - ☐ Dokumentation der Zerteilungsgründe von Modulen/Komponenten
  - ☐ SOC, Information Hidding, SOLID
- ☐ TD-Managementzyklus
  - ☐ TD Erkennungsvorgehen etabliert (Bsp. aim42)
  - ☐ Verantwortliche für den Zyklus benannt
  - ☐ Code Analyse
    - ☐ Abhängigkeitsverletzungen
    - ☐ Code Smells Überwachung
    - ☐ Tests vorhanden
    - ☐ Automatisierte Tests
    - ☐ Tools sind up-to-date
    - ☐ Sicherstellung der Einhaltung der Mustersprachen
- ☐ Qualitätsfunktionen
  - ☐ Mustersprachen definiert
  - ☐ Use Cases/ Test-Cases/ Anforderungs-Katalog/ Handbücher/ DDD
  - ☐ Qualität
    - ☐ Qualitätsziele Vorhanden
    - ☐ Wichtigste Qualitätsmerkmale definiert
    - ☐ Randbedingungen erfasst
    - ☐ Stakeholder definiert
    - ☐ Kontinuierliche Überprüfung der Qualitätsziele

- ☐ Messung der Anpassungsfähigkeit
  - ☐ Gelöste Tickets vs neue Tickets
  - ☐ Deploymentzyklen
  - ☐ Fehlerrate
  - ☐ Fallback Zeit
  - ☐ Umsetzungszeiten

A.2 Checkliste Angemessene Microservices-Architektur

- ☐ Softwarearchitektur-Prozess
  - ☐ Architektur Dokumentationen (Bsp. arc42)
  - ☐ Technologiebeurteilung
  - ☐ Architekturüberprüfung
  - ☐ Architekturziele
- ☐ Modularisierung
  - ☐ Fachliche Zergliederungsmethodik
  - ☐ Definition von Modulen
  - ☐ Definition von Modulen in Modulen
  - ☐ Definition der Interaktionsregeln zwischen Modulen (API)
  - ☐ Dokumentation der Zerteilungsgründe von Modulen/Komponenten
  - ☐ SOC, Information Hidding, SOLID
  - ☐ Definition von Microservices
  - ☐ Trade-off-Analyse zur Microservices Zerteilung/Zusammenführung
    - ☐ Koordination definiert
- ☐ TD-Managementzyklus
  - ☐ TD Erkennungsvorgehen etabliert (Bsp. aim42)
  - ☐ Verantwortliche benannt
  - ☐ Code Analyse
    - ☐ Abhängigkeitsverletzungen
    - ☐ Code Smells Überwachung
    - ☐ Tests vorhanden
    - ☐ Automatisierte Tests
    - ☐ Tools sind up to date
    - ☐ Sicherstellung der Einhaltung der Mustersprachen
  - ☐ ATD Erkennungsstrategie
    - ☐ Strategie up to date
    - ☐ Makroebene definiert
    - ☐ Bouded-Context-Ebene definiert
    - ☐ Mikroebene definiert
    - ☐ ATD Checkliste
- ☐ Qualitätsfunktionen

- ☐ Mustersprachen definiert
- ☐ Use Cases/ Test-Cases/ Anforderungs-Katalog/ Handbücher/ DDD
- ☐ Qualität
  - ☐ Qualitätsziele Vorhanden
  - ☐ Wichtigste Qualitätsmerkmale definiert
  - ☐ Randbedingungen erfasst
  - ☐ Stakeholder definiert
  - ☐ Kontinuierliche Überprüfung der Qualitätsziele
- ☐ Messung der Anpassungsfähigkeit
  - ☐ Gelöste Tickets vs neue Tickets
  - ☐ Deploymentzyklen
  - ☐ Fehlerrate
  - ☐ Fallback Zeit
  - ☐ Umsetzungszeiten

A.3 Checkliste ATD

- Middleware/ Tranportlayer
  - ☐ Nachrichten Header vorhanden (Tracing ID, etc.)
  - ☐ Adapter für die Business Logik
  - ☐ Dead letter Queue vorhanden
  - ☐ Message/Event Software nicht als DB verwendet
- Infrastruktur
  - ☐ Resiliente Services
  - ☐ Monitoring (Prometheus)
  - ☐ Skalierung und Elastizität möglich
  - ☐ Tracing Software (Jaeger)
  - ☐ Trafik Software (Kiali)
  - ☐ Loadbalancing
  - ☐ Service discovery
  - ☐ Message Broker oder Event-Queue
- Organisation
  - ☐ Microservices nicht als Hype
  - ☐ Retrospektiven vorhanden?
  - ☐ Resiliente Organisation
  - ☐ Team Topologies
  - ☐ Offen für Änderung
  - ☐ Fachliche Zergliederung der Teams
  - ☐ Kollaborationssicht
  - ☐ DDD
- Mikro-Ebene

- ☐ Best-Practices beim API-Design berücksichtigt
- ☐ API-Dokumentation
- ☐ Ubiquitous Language eingehalten
- ☐ Configuration as Code

- Makro-Ebene
  - ☐ Keine Deployment-Abhängigkeiten
  - ☐ Architekturkonzept vorhanden?
  - ☐ Keine zu starken Einschränkungen
  - ☐ Kein Verteilter Monolith
  - ☐ Metadokumentation jedes Microservice
    - ☐ Format Definition
    - ☐ Bereitstellungsdefinition
- Datenschicht
  - ☐ DDD
  - ☐ Kontext-Map Dokumentation
  - ☐ So viel BASE wie möglich
- Life-Cycle
  - ☐ Kein Dead Code vorhanden?
  - ☐ Keine Dead Services vorhanden?
  - ☐ Erstellung neuer Services ohne Hürden
- Test
  - ☐ Contract Testing
  - ☐ Hohe Testabdeckung
  - ☐ Mutation Testing
  - ☐ FF für die wichtigsten Eigenschaften

Erklärung zur selbstständigen Bearbeitung einer Abschlussarbeit

Hiermit versichere ich, dass ich die vorliegende Arbeit ohne fremde Hilfe selbständig verfasst und nur die angegebenen Hilfsmittel benutzt habe. Wörtlich oder dem Sinn nach aus anderen Werken entnommene Stellen sind unter Angabe der Quellen kenntlich gemacht.

Ort	Datum	Unterschrift im Original