MASTERARBEIT

MICROSERVICES

Softwarearchitektur mit Einfluss auf Codequalität

ausgeführt am

Studiengang

Informationstechnologien und Wirtschaftsinformatik

Von: Hannes Kainz

Personenkennzeichen: 1510320012

Graz, am 14. Dezember 2016 .....

Unterschrift

EHRENWÖRTLICHE ERKLÄRUNG

Ich erkläre ehrenwörtlich, dass ich die vorliegende Arbeit selbstständig und ohne fremde Hilfe verfasst, andere als die angegebenen Quellen nicht benützt und die benutzten Quellen wörtlich zitiert sowie inhaltlich entnommene Stellen als solche kenntlich gemacht habe.

.....

Unterschrift

DANKSAGUNG

Jetzt ist es an der Zeit, Danke zu sagen all jenen, die mich während der Zeit des Studiums und speziell in der Phase des Schreibens der Masterarbeit mit Rat und Tat zur Seite gestanden sind. Ein großer Dank gilt meinem Betreuer, FH-Professor DI (FH) Manfred Steyer, MSc., der mich im Entstehungsprozess der Arbeit immer unterstützt hat. Vielen Dank für die konstruktive Zusammenarbeit. Auf diesem Weg möchte ich mich auch bei meinen Interviewpartnern für ihre Bereitschaft zur Mitarbeit bedanken. Der Einblick in ihre persönliche Arbeitsweise und in die Unternehmen, in denen sie tätig sind, war für diese Arbeit sehr hilfreich.

Abschließend möchte ich mich bei meiner Partnerin sowie meiner Familie und meinen Freunden bedanken, welche mich in der schwierigen und stressigen Zeit motiviert und verständnisvoll unterstützt haben.

KURZFASSUNG

Der Fokus dieser Arbeit liegt auf dem Identifizieren und Verifizieren des Einflusses von Microservices auf die Codequalität von Softwareanwendungen. Im ersten Teil der Arbeit sind Definitionen und Architekturen der untersuchten Softwareapplikationen dargelegt. Dies beinhaltet die Definition von monolithischer Software und deren Architektur und die Definition von Microservices mit seinen Vorteilen (zum Beispiel die Skalierung durch automatisiertes Deployment) und Nachteilen (zum Beispiel höhere Komplexität durch die erhöhte Anzahl von Softwareteilen). Darauf folgt eine Analyse von den ähnlichen Softwarearchitekturkonzepten der serviceorientierten Architektur und den Self-contained Services. Der Theorieteil zu Microservices endet mit der Darstellung von Methoden zur Trennung von monolithischer Software in Microservices und wie diese Dienste wieder zu einer Gesamtanwendung zusammengeführt werden können. Zur Vervollständigung ist der aktuelle Forschungsstand auf dem Gebiet der Microservices angeführt und es wird ein Einblick in eine Auswahl an Softwaremetriken zur Erhebung von Softwarequalität gegeben.

Im praktischen Teil der Arbeit wurden die theoretischen Erkenntnisse an einer Softwareapplikation angewandt. Von einem Monolithen wurden Microservices abgespalten und die resultierende Anwendung als auch der Monolith einer Softwarearchitekturanalyse unterzogen. Danach wurden Softwaremetriken für die Ausgangsapplikation und die Microservices berechnet, um festzustellen, ob sich die Codequalität verändert hat. Zum Abschluss wurde eine Analyse der Änderungsgeschwindigkeit in beiden Applikationen durchgeführt, gefolgt von Interviews mit Softwareentwicklern zu dem Qualitätsmerkmal Lesbarkeit von Quellcode.

Die Ergebnisse zeigen eine Verringerung der Komplexität des entstandenen Quellcodes und eine Erhöhung der Lesbarkeit gegenüber der Ausgangsanwendung.

ABSTRACT

The main goal of this thesis was to identify and verify the influence of the microservices software architecture on code quality. To this end, the thesis first presents the current state of research in the field of microservices, as well as definitions and an architecture analysis of software with a monolithic design approach. The first part covers these monolithic applications and then defines microservices, including a discussion of their advantages (e.g. rapid scaling through automated deployment) and disadvantages (e.g. higher complexity due to incrased number of services in an application). Next, the similar software architectures of service-oriented architecture and self-contained services are compared to microservices. The second part of the thesis then explains methods for separating a monolithic application into microservices and then integrating those services. The theory section concludes with a description of software metrics (with a focus on code quality) and how to calculate them.

In the practical portion of the thesis, the theoretical knowledge was then applied to transform a monolithic application into a system containing microservices. The monolith and the resulting microservices were then examined and compared using a series of software architecture analysis methods. Software metric analysis was performed on the resulting applications in order to determine if microservices improved code quality. Next, the time developers needed to change those applications was determined, in order to document differences in development speed. In the final step, the developers were interviewed to obtain their opinions about the influence of microservices on code readability.

The results show that the duration of application changes is not affected by the conversion from monolithic application to microservices. However, the other findings indicate that the complexity decreases and the readability improves.

INHALTSVERZEICHNIS

1 EINLEITUNG

„If you hit the Amazon.com gateway page,
the application calls more than 100 services to collect data and construct the page for you.“
Werner Vogels

Beispiele wie Amazon und Netflix zeigen, dass für zukünftige, und bereits bestehende, Herausforderungen im Bereich von verteilten Systemen, Microservices immer mehr an Relevanz gewinnen. Das Einstiegszitat gab Werner Vogels, Chief Technology Officer (CTO) von Amazon, bereits im Jahr 2006 im Rahmen eines Interviews für das ACM Queue Magazin (O'Hanlon, 2006). Der steigende Bedarf an Ressourcen für Webdienste hat Microservices bereits salonfähig gemacht. Diese Arbeit versucht nun zu beleuchten, ob und inwiefern Microservices die Codequalität von Anwendungen im Gesamten verändern.

1.1 Motivation

Die Motivation für diese Untersuchung wurde aus dem Studium, aus der Neugierde auf neue Technologien und aus dem Arbeitsleben heraus gespeist. Als berufstätiger Softwareentwickler kommt der Autor mit verschiedensten Anwendungen auf dem Branchensektor in Kontakt. Viele haben eines gemeinsam: Sie wurden durch jahrelange Entwicklung zu schwer überschaubaren und aufwendig zu wartenden Anwendungen. Die Lösung dafür verspricht der moderne Ansatz der Microservices. Dieser bietet, unter den gegebenen Voraussetzungen, Möglichkeiten, den sogenannten Monolithen in kleinere Teilaufgaben zu zerkleinern. Das Ergebnis ist ein entscheidender Vorteil. Die Steigerung der Bereitschaft der MitarbeiterInnen sich an diesem nun überschaubaren Teil des Quellcodes aktiv zu beteiligen. In der Theorie wird ein zweiter Vorteil diskutiert, die gleichzeitige Erhöhung der Codequalität durch Reduktion der Komplexität. Diese Situation ist die Ausgangsbasis für die Forschung im Rahmen der vorliegenden Masterarbeit. (Fowler & Lewis, 2015)

1.2 Forschungsfrage

Die Arbeit hat das Ziel, die Forschungsfrage „Wie wirkt sich der Einsatz des Architekturentwurfes Microservices auf die Codequalität aus?“ zu beantworten und geht darin methodisch, wie in Kapitel 1.4 beschreiben, vor.

1.3 Zielsetzung

Um diese Forschungsfrage beantworten zu können, werden im Zuge der Masterarbeit folgende Hypothesen überprüft:

Hypothese 1: Die Anwendung von Microservices beschleunigt die Adaptierung der Software durch EntwicklerInnen.

H0: Die Anwendung von Microservices verändert die Adaptierungsgeschwindigkeit nicht.

Begründung: Durch Microservices wird der Funktionsumfang von Software reduziert und gekapselt. Dadurch können Änderungen an überschaubareren Teilen der Software schneller durchgeführt werden. (Rau, 2016)

Hypothese 2: Die Anwendung von Microservices reduziert die Komplexität von Quellcode. (Rau, 2016)

H0: Microservices haben keine Auswirkung auf die Komplexität von Quellcode.

Begründung: Durch den Einsatz von Microservices wird eine Anwendung in viele kleine Anwendungen unterteilt. Trotz des vermehrten Kommunikationsaufwandes untereinander wird die Komplexität in Summe reduziert.

Hypothese 3: Microservices erhöhen die Codequalität in Form von Lesbarkeit in einer Softwareanwendung.

H0: Microservices verändern die Lesbarkeit von Quellcode nicht.

Begründung: Durch die Vereinfachung von Prozessabläufen in einer Anwendung wird die Lesbarkeit automatisch erhöht.

1.4 Vorgehen

Zu Beginn werden die theoretischen Grundlagen zu den Themen monolithische Software, Microservices und Codequalität und dessen Erhebungsmethoden dargelegt. Ergänzt werden diese theoretischen Grundlagen durch aktuelle Forschungsanwendungen und -berichte aus dem Umfeld von Microservices. Dies wird durch eine systematische Literaturrecherche in den Fachdatenbanken von Ebsco und Emerald ermöglicht.

Der Versuch beschäftigt sich mit der Anwendung der Erkenntnisse aus dem theoretischen Hintergrund auf eine monolithische Anwendung. Diese wird nach der theoretischen Vorlage in kleine Microservices zerlegt und im Anschluss auf deren Codequalitätsveränderung untersucht. Zur Vervollständigung der Erkenntnisse werden mit SoftwareentwicklerInnen Interviews geführt, um durch einen Mehrmethodenansatz die Ergebnisse zu unterstützen.

1.5 Anwendungsfall

Als Ausgangsbasis für den zu untersuchenden Anwendungsfall dient eine Anwendung aus dem Unternehmen des Autors. Diese wird architektonisch analysiert und anschließend unter den Gesichtspunkten der Microservices verändert zu einem Zielprodukt, bestehend aus der verbleibenden monolithischen Anwendung zur Benutzerinteraktion und den abgespaltenen Microservices auf Ebene der Geschäftslogik. Dies wird durch den Autor selbstständig durchgeführt und abschließend, mittels automatisierter Erhebungen von Codemetriken und Interviews mit SoftwareentwicklerInnen, verifiziert.

2 DEFINITION

Microservices sind in der IT-Branche ein noch junger Begriff. Amazon gilt in diesem Bereich als einer der Vorreiter, denn schon im Jahr 2006 hielt der CTO von Amazon, Werner Vogels, einen Vortrag, worin er die Arbeitsweise seiner Teams beschrieb. Aus der Not der Skalierbarkeit war die Idee einzelner Services mit eigener Datenbank geboren. Verteilte Teams kümmerten sich um die Problemlösung eines Business-Problems und erarbeiteten darauf eine Lösung in Form eines Services. (Wolff, 2006)

In diesem Kapitel wird eine Definition der Microservices dargestellt und wie sie sich zu anderen Mustern in der Programmierwelt abgrenzen oder diese ergänzen. Den Einstieg bildet der monolithische Ansatz, Software zu entwerfen.

2.1 Monolith

Dieses Kapitel widmet sich dem monolithischen Anwendungsdesign, wie dieses definiert ist und welche Architektur in Monolithen nach Reenskaug (1979) Anwendung erfahren sollte.

2.1.1 Definition

Eine monolithische Softwarearchitektur definiert sich durch eine umfangreiche Anwendung, in welche sowohl die gesamte Geschäftslogik, als auch die Datenbankzugriffslogik und die Benutzerschnittstelle in einem Prozess kombiniert sind. Meist wird die spezifizierte Architektur in einem einzigen Systembaustein zusammengefasst. (Vogel, et al., 2009) Wie Wolff (2016) ausführt, kann ein Monolith auch aus der Sicht der Installation, beziehungsweise des Deployment, gesehen werden. Aus dieser Perspektive ist ein Monolith ein System, welches nur als Ganzes auf einmal installiert werden kann.

In der Literatur entsteht der Monolith meist aus einem prototypischen Verfahren. Eine Anwendung wird programmiert und bevor es zur Auslieferung kommt, werden von den KundInnen neue Anforderungen an die Anwendung gestellt. Natürlich werden diese Zusätze implementiert. Wird nicht auf einen Softwarearchitekturplan zurückgegriffen kann es passieren, dass die Softwareanwendung zu einer mit den für monolithische Anwendungen typischen Erkennungsmerkmalen wächst. (Hanmer, 2013)

- Ein Quellcode.
- Somit eine Anwendung.
- „Quelltext, der an zahlreichen Stellen im System angepasst werden muss, wenn Systembausteine, wie beispielsweise Datenbank oder Betriebssystem, geändert werden.

- Klassen, die sehr viele ganz unterschiedliche Verantwortlichkeiten abdecken und deshalb nur schwer wiederzuverwenden sind („Monster“-Klassen).
- Fachklassen, deren Implementierungsdetails im gesamten System bekannt sind.“ (Vogel, et al., 2009)

Während es auch Vorteile, wie eine einfache Programmierung, einfaches Deployment und auch einfaches Skalieren bei erhöhten Benutzungsanforderungen durch Load-Balancer gibt, so liegen die Nachteile einer solchen Architektur, auch wenn diese meist über Jahre entsteht, auf der Hand. Die große Quellcodebasis eines Monolithen schreckt neue EntwicklerInnen meist ab, wodurch der Einarbeitungsaufwand dieser MitarbeiterInnen um vieles höher ist. Die Größe spielt auch eine Rolle bei der Implementierung von neuen Anforderungen. MitarbeiterInnen brauchen mehr Zeit, um bestehende Funktionen zu verstehen und neue Anforderungen zu implementieren. Dies resultiert in einer verlängerten Implementierungsdauer (Wartbarkeit, Erweiterbarkeit). Die Größe des Quellcodes betrifft auch den einfachen Implementierungsprozess. Die Entwicklungsumgebung wird substantiell langsamer, je größer das Projekt ist. Der schwerwiegendste Nachteil eines Monolithen ist jedoch die eingegangene Verbindung zur gewählten Technologie. Wenn eine neue Anwendung entworfen werden soll, muss sehr genau überlegt werden, ob und wie lange es diese Technologie noch geben wird. (Richardson, 2014)

2.1.2 Architektur

Architekturmuster sind Entwurfsvorlagen auf höherer Ebene und beschreiben somit den generellen gesamtheitlichen Aufbau einer Softwareanwendung. Das Ziel dieser Muster besteht darin, einzelne Teile übersichtlicher, wartbarer, testbarer und wiederverwendbarer zu gestalten. Der Einsatz dieser Muster gestaltet den Aufbau und beschreibt, wie einzelne Teile des Musters miteinander kommunizieren. Auch diese Muster sind in mehrere Kategorien unterteilt. Buschmann, Meunier, Rohnert, Sommerlad und Stal (1996) sowie Reenskaug (1979) sprechen hier von:

- Chaos zur Struktur durch die Schichtenarchitektur,
- Verteilten Systemen,
- Interaktiven Systemen und
- Adaptiven Systemen.

Die Schichtenarchitektur schreibt vor, die Softwareanwendung in mehrere vertikale Schichten aufzuteilen und den Zugriff so zu regeln, dass jede Schicht nur in sich selbst sowie den angrenzenden Schichten Zugriff hat. Dadurch wird eine Trennung möglich, weil zum Beispiel nicht mehr die Präsentationslogik direkt auf die Datenbankschicht Zugriff erhält, sondern dies nur über eine Verwaltungsschicht ermöglicht wird, wie es in Abbildung 2-1 dargestellt ist. Clientanwendungen sind von der Schicht der Geschäftslogik (Application Layer) getrennt, ebenso wie die Geschäftslogik von der Logik zur Datenhaltung (Data Layer). Die Cross Cutting Concerns bieten Dienste innerhalb einer Softwareanwendung an, welche von allen Schichten verwendet werden. Zum Beispiel einen Mechanismus für einheitliches

Logging. Eine Form der Schichtenarchitektur wurde von Reenskaug (1979) im Xerox Parc entwickelt und beschreibt die Aufteilung in die drei Bereiche Model, View und Controller, wobei das Model das Wissen repräsentiert und aus mehreren Objekten zusammengesetzt sein kann. In modernen Computersystemen spricht man hier auch oft von der Datenbankschicht. In dieser Schicht sind alle Informationen verfügbar. Die View ist die visuelle Repräsentation des Models und hebt bestimmte Merkmale hervor oder verbirgt andere. Sie dient somit als Filter für die Daten aus dem Model. Auch in heutigen Systemen dient die View der Anzeige von Informationen, wird aber nicht eingesetzt, um direkt mit dem Model zu kommunizieren, wie es Reenskaug beschreibt. Dafür wird die Zwischenschicht des Controllers verwendet. Sie ist zuständig für jegliche Interaktion mit dem/der BenutzerIn und entscheidet zum Beispiel, welche Views dem/der BenutzerIn zur Anzeige gebracht werden. Sie nimmt Eingaben durch die View entgegen und modifiziert das Model anhand dieser Daten. In der Abbildung 2-1 sind diese drei Schichten in einer erweiterten Form dargestellt. Daten- und Domänenmodellschicht repräsentieren das Model, die Clients repräsentieren Views und der Application Layer repräsentiert die Schicht des Controllers.

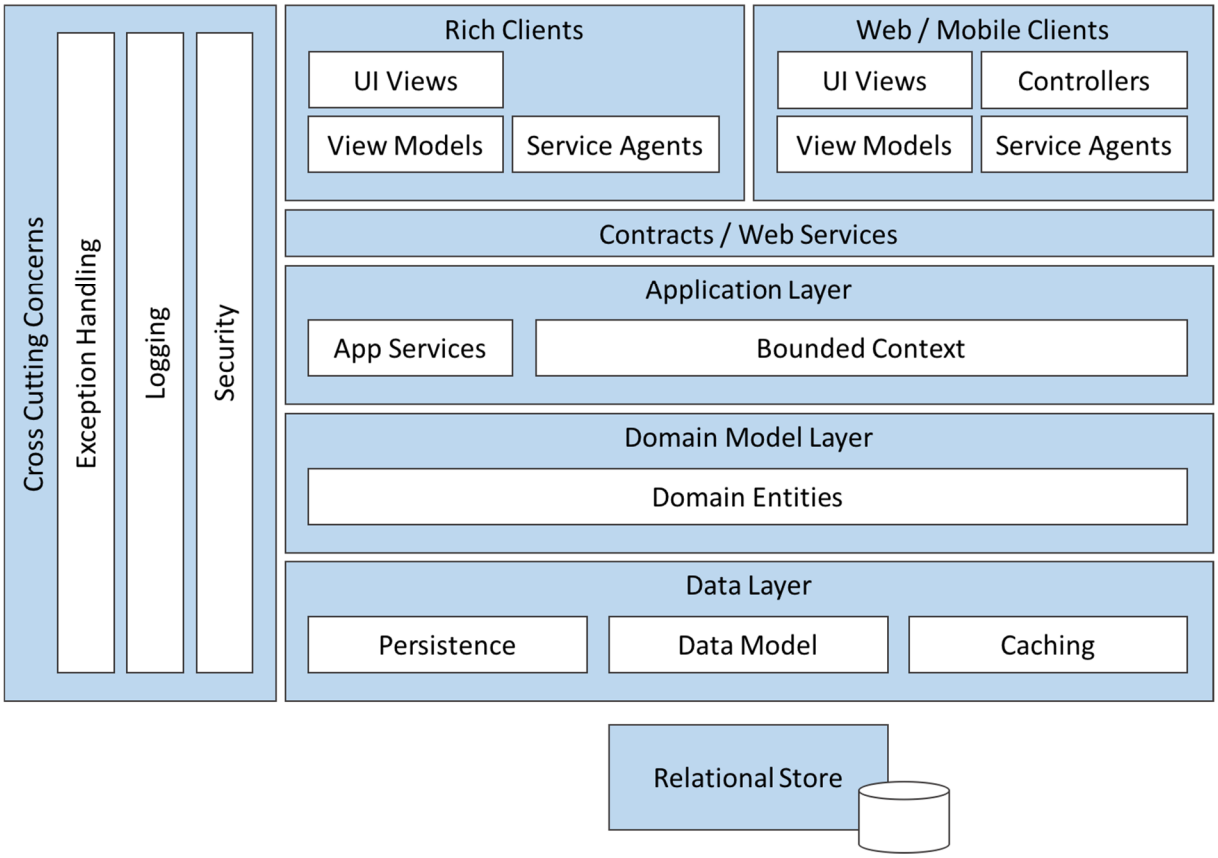


Abbildung 2-1: Schichtenarchitektur (Familiar, 2015, S. 23)

2.2 Microservices

Microservices sind ein neuer Typ von Architekturmuster zur Gestaltung von großen, skalierenden Anwendungen. Der Name Microservices entstand daraus, die gesamte Anwendung in kleine Teile zu zerschneiden, in Services, welche unabhängig voneinander je

einen Teil der Gesamtanwendung ausführen (Daya, et al., 2015). Eine weitere Annäherung an den Begriff Microservices bieten folgende Kriterien:

- Sie sind ein Konzept und erfordern daher ein Umdenken in der Art der Programmierung, beziehungsweise dem gesamten Programmierprozess.
- Sie können in unterschiedlichen Programmiersprachen verfasst sein, da es keine direkte Verknüpfung der einzelnen Programmteile gibt, außer den definierten Serviceschnittstellen.
- Sie haben eine eigene Datenablage oder Datenbank.
- Sie können einzeln erstellt und verteilt werden, ohne die Funktionalität des Gesamtsystems zu beeinträchtigen. (Wolff, 2016)

Die Vorteile der Microservices sind abgeleitet von den Vorteilen der verteilten Systemen im Allgemeinen. Sie gehen jedoch einen Schritt weiter und heben diese auf eine neue Ebene. In den folgenden Kapiteln werden die Eigenschaften und Vorteile von Microservices nach Newman (2015) näher betrachtet.

2.2.1 Eigenschaften

Microservices sind durch zwei Eigenschaften charakterisiert. Sie sind klein und auf deren Aufgaben spezialisiert und sie sind grundlegend autonom. Die Bedeutung der Eigenschaften wird in diesem Kapitel erörtert. (Newman, 2015)

2.2.1.1. Klein und spezialisiert

Wenn eine Anwendung implementiert wird, werden Funktionen hinzugefügt, Anforderungen erfüllt und der Quellcode wächst. Auch wenn Softwarearchitekturmuster zum Einsatz kommen, stößt der monolithische Ansatz an seine Grenzen. Sobald ein Quellcode für die gleiche Funktion, abgeleitet aus einer Geschäftsanforderung, an vielen Stellen im System verteilt liegt, wird die Implementierung von neuen Funktionen, aber auch das Beheben von Fehlern, immer schwieriger. Im Monolithen wird versucht, dieses Problem mit der Kapselung von Funktionen in Modulen zu umgehen und dies ist auch der primäre Ansatz der Microservices. Das Prinzip von Robert C. Martin (2003), das „Single Responsibility Principle“ findet hier Anwendung, um den Quellcode für die gleiche Funktion abzukapseln. Es besagt, dass der Quellcode mit demselben Grund zur Änderung getrennt werden soll von dem Quellcode, welcher wegen eines anderen Grundes verändert werden muss. Microservices heben dieses Prinzip zur Trennung von Funktionalitäten in eigene Services und durch die spezielle Fokussierung auf eine Funktion kann der Quellcode klein, im Sinne von wenig Quellcode, gehalten werden. Eine sinnvolle Grenze zu ziehen, welche Funktionen in einem neuen Service zu implementieren sind, diesem Thema widmet sich das Kapitel 3.1.

2.2.1.2. Autonom

Microservices sind autonom zu betrachten. Das bedeutet, dass Services untereinander nur durch Netzwerkaufrufe miteinander kommunizieren. Dies soll die saubere Trennung der einzelnen Microservices fördern. Durch die isolierte Betrachtung ist es möglich, Microservices lokal als eigenen Prozess zu betreiben oder aber auch in der Cloud als Platform as a Service (PAAS) Dienst.

Um die Unabhängigkeit zu gewährleisten, ist es wichtig, zu Beginn festzulegen, was bietet das Microservice über eine Programmierschnittstelle an und welche Bereiche bleiben privat. Das Ziel eines jeden Microservices ist es, verändert und neu installiert zu werden, ohne eine einzige Zugriffsanwendung auf das Microservice zu verändern. Dies ist gänzlich nur möglich, indem eine möglichst unabhängige Technologie eingesetzt wird, welche das unabhängige Installieren und den losgelösten Zugriff ermöglicht.

2.2.2 Vorteile

Die Vorteile von Microservices ergeben sich primär aus den Vorteilen von verteilten Systemen. Sie vertiefen diese jedoch und erreichen einen höheren Grad der Unabhängigkeit als Systeme nach der serviceorientierten Architektur. (Newman, 2015)

2.2.2.1. Technologieunabhängig

Ein Vorteil von Microservices ist deren grundsätzliche Technologieunabhängigkeit. Grundsätzlich, weil Microservices wie in Kapitel 2.2.1 beschrieben, autonom sind und nur durch standardisierte Schnittstellen miteinander kommunizieren. Wenn jedoch das Unternehmen vorschreibt, dass gewisse unternehmenskritische Dienste in derselben Sprache oder Technologie zu verfassen sind, welche von einem Großteil der EntwicklerInnen verstanden wird, kann es zu Einschränkungen der Technologie-unabhängigkeit kommen.

Der Vorteil besteht in der schnelleren Adaptierung von neuen Technologien. Dies können für den betreffenden Dienst Vorteile durch zum Beispiel schnellere Verarbeitung oder eine spezielle Art der Datenspeicherung sein. Wie in der Abbildung 2-2 ersichtlich, können für verschiedene Microservices verschiedene Technologien auf programmiersprachlicher Ebene, aber auch auf der Ebene der Datenspeicherung gewählt werden. So kann für einen Bildspeicher ein effizienter Blobspeicher gewählt werden, während für ein Freundesnetzwerk eine Graphendatenbank passender ist.

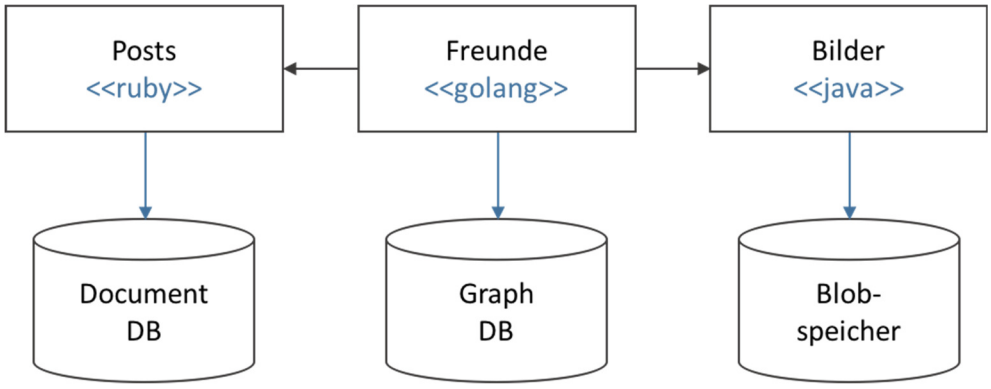


Abbildung 2-2: Microservices erlauben den Einsatz unterschiedlichster Technologien (Newman, 2015, S. 20)

Durch die Eigenschaft der Größe und Spezialisierung können so neue Technologien in kleinen, abgeschotteten und nicht unternehmenskritischen Diensten ausprobiert werden. Die Größe ermöglicht in diesem Zusammenhang eine schnelle Entscheidung, ob die neue Technologie für diesen oder auch mehrere Dienste Anwendung findet. Zuvor, wie in einem monolithischen Ansatz üblich, musste die gesamte Applikation neu implementiert und getestet werden.

2.2.2.2. Ausfallsicher

Die Ausfallsicherheit wird im Rahmen einer Microservices-Strategie erhöht, kann aber auch neue Probleme in der Art der Kommunikation aufwerfen. Durch die einzig zugelassene Kommunikation über das Netzwerk ergeben sich neue Probleme. Netzwerke können und werden ausfallen, ebenso wie Computer, auf denen die verteilten Microservices betrieben werden.

Im Gegensatz zum monolithischen Modell jedoch kann im Microservice-Umfeld auf Ausfälle einzelner Teile reagiert und so der Betrieb des Gesamtsystems, wenn auch nicht in vollem Funktionsumfang, gesichert werden. Bei einem Monolithen funktioniert das System im Regelfall durch den Ausfall eines Teiles nicht mehr. Hier kann nur durch die Duplizierung des Dienstes eine Ausfallsicherheit gewährleistet werden.

2.2.2.3. Skalierbar

Microservices sind skalierbarer im Vergleich zu monolithischen Anwendungen. Durch die Verteilung von Geschäftsfällen auf einzelne Dienste können jene Dienste mit hohem Ressourcenbedarf oder hoher Frequenz einfach dupliziert und somit skaliert werden. Im Vergleich dazu kann eine monolithische Anwendung nur durch das Duplizieren der gesamten Anwendungen und dem Einsatz eines Load-Balancers skalieren. Der Vorteil im Microservice-Ansatz liegt im schonenden Umgang mit vorhandenen Ressourcen und trägt somit ultimativ zu niedrigeren Kosten im Betrieb der Anwendung bei.

2.2.2.4. Einfach zu installieren

Die Deployment-Geschwindigkeit ist eine der großen Herausforderungen in der modernen Anwendungsentwicklung. Viele Unternehmen versuchen dies durch den Einsatz einer geeigneten Organisationsform zu erreichen. Microservices bieten dieselben Vorteile, da auch hier meist nach den Programmiermustern Scrum oder Kanban entwickelt wird (Dräther, Koschek, & Sahling, 2013).

Im Microservice-Ansatz ist es durch die Abstraktion und Spezialisierung möglich, kleine Änderungen schnell zu den KundInnen zu bringen. Dadurch ergibt sich eine schnelle Feedbackschleife, um Qualitätsmängel auszubessern. Im Vergleich dazu muss in monolithischen Applikationen durch eine Änderung an einer Codezeile die gesamte Anwendung neu übersetzt und installiert werden. Daraus resultiert ein hoher Aufwand bei der Fehlersuche, denn es werden viele Änderungen zusammen neu erstellt und installiert. Dies hat den Effekt, dass Probleme nicht schnell identifiziert und behoben werden können, da alle zuletzt installierten Änderungen überprüft werden müssen. In einer Microservice-Anwendung müssen im Vergleich nur kleine Inkremente der Anwendung auf Fehler überprüft werden.

2.2.2.5. Angepasst an die Organisation

Amazon vertritt die Auffassung, dass jenes Team, welches einen Dienst im Sinne der Microservices entwickelt hat, diesen auch in Betrieb nimmt und warten soll. Werner Vogels betitelte dieses Vorgehen mit dem Spruch: „You build it, you run it“ (O'Hanlon, 2006). Der Vorteil liegt in der Auffassung, dass jedes Entwicklungsteam für seinen eigenen Quellcode bis zum Ausscheiden des Dienstes verantwortlich ist. Dadurch kommen gewöhnliche „throw it over the wall“-Probleme nicht zu tragen, wenn Anwendungen vom Entwicklungsteam an das Team für den Betrieb übergeben werden. Viele EntwicklerInnen arbeiten bereits in kleinen und womöglich dezentralen Teams und können durch den Einsatz von Microservices ihren Quellcode besser abschätzen und die volle Verantwortung für ihre Anwendung übernehmen.

Bestätigt wird dies durch das Gesetz von Conway (1968), welches besagt, dass Organisationen als Resultat ihrer Arbeit immer Abbildungen der Organisation selbst liefern. Das bedeutet im Zusammenhang von Microservices, dass die Organisationform des Entwicklungsteams im Unternehmen ein wichtiger Faktor ist. Je losgelöster die einzelnen Teams sind, desto weniger eng gekoppelt sind die Dienste, die von diesen Teams entwickelt werden. Untermauert wird dies auch von Familiar (2015). Dieser beschreibt, dass die Offenheit der Microservices gefördert wird durch dezentrale und funktionsübergreifende Teams.

2.2.2.6. Kombinierbar

Ein weiteres Versprechen der Microservices ist deren Kombinierbarkeit. Der Quellcode kann in verschiedenen Situationen wiederverwendet werden, beziehungsweise durch eine

Kombination von bestehenden Microservices zu einem neuen Service verschmelzen. Durch die immer größer werdende Bandbreite von Zugriffsmöglichkeiten für den/die EndanwenderIn, sei es durch eine herkömmliche Desktopanwendung, durch eine Webseite oder auch durch eine vorgesehene Programmierschnittstelle, müssen Anwendungen immer mehr Anforderungen standhalten. Eine monolithische Anwendung muss alle Anforderungen in sich abbilden und über die gesamte Laufzeit bereitstellen. Microservices gehen den Weg, dass jeder definierte Zugriffspunkt über ein eigenes Microservice bereitgestellt wird, um größtmögliche Flexibilität im Umgang mit den gestellten Anforderungen zu erreichen.

2.2.2.7. Optimiert für Ersetzbarkeit

Microservices bieten durch ihre Größe von nur wenigen hundert Zeilen Code die Möglichkeit, sie einfach und kosteneffektiv zu löschen und neu zu entwickeln, wenn zum Beispiel neue Technologien am Markt erscheinen. Bei einer monolithischen Anwendung gestaltet sich die Ablöse schwieriger. Viele dieser Anwendungen sind horizontal als auch vertikal tief integriert und werden somit nur selten ersetzt. Der Aufwand im Hinblick auf Kosten und Personaleinsatz erscheint enorm.

2.2.3 Nachteile

Fowler (2015) beschreibt in einem Artikel gegenüber den Vorteilen von Microservices auch Nachteile, welche durch deren Einsatz entstehen. Er führt an, dass durch die Unabhängigkeit und Eigenständigkeit und dadurch eingehende Verteilung von einzelnen Services über die Computergrenze hinweg die Komplexität von verteilten Systemen erhöht wird. Dies beinhaltet die ansteigende Zeitdauer von Aufrufen, welche nun über das Netzwerk vorgenommen werden, bis hin zu womöglich fehlschlagenden Aufrufen.

Als zweiten negativen Aspekt führt Fowler die fehlende Konsistenz von Aufrufen und der erschwerten Handhabung von Transaktionen ins Feld. In Kapitel 3.1.3.2 wird näher darauf eingegangen, wie mit Transaktionen nach dem Trennen des Quellcodes umgegangen werden muss, vor allem, dass alle teilnehmenden Systeme auf entweder keine oder zeitverzögerte Konsistenz der Daten abgestimmt sein müssen.

Der dritte Nachteil entsteht durch die Unabhängigkeit der Microservices. Kleine und zustandslose Teile einer großen Anwendung sind einfach und schnell zu verteilen und können im Idealfall schrittweise abgelöst und erneuert werden. All diese Vorteile bergen jedoch den großen Nachteil des erhöhten Aufwandes des Betriebs. Um die Abhängigkeiten der Services untereinander und zu Außensystemen im Griff zu haben, wird ein Team benötigt, welches dieser Aufgabe gewachsen ist. Des Weiteren müssen Tools zur automatisierten Verteilung und Installation sowie Instandhaltung der Microservices eingesetzt werden, ohne diese wäre die Aufgabe des Betriebs nicht zu bewältigen (siehe Kapitel 2.3.2). (Fowler, 2015)

2.2.4 Prinzipien

Microservices liegen, aus der objektorientierten Programmierung abgeleitet, zwei Prinzipien zugrunde. Diese sind die starke Kohäsion sowie die lose Kopplung von Elementen. Nachfolgend sind diese näher erklärt.

2.2.4.1. Starke Kohäsion

Die Kohäsion beschreibt die Zusammengehörigkeit von einzelnen Bausteinen in einem Modul, und wenn man den Modulgedanken weiterspinnt, in einzelnen Microservices (Evans, 2004). Die Kohäsion wird von der losen Kopplung unterstützt, denn „je höher die Kohäsion individueller Bausteine einer Architektur ist, desto geringer ist die Kopplung zwischen den Bausteinen“ (Vogel, et al., 2009, S. 133).

2.2.4.2. Lose Kopplung

Die lose Kopplung ist Voraussetzung für die Vorteile der Ersetzbarkeit und der schnellen beziehungsweise einfachen Installation und Verteilung. Die Unabhängigkeit der einzelnen Microservices untereinander muss gegeben sein, um einzelne Dienste außer Betrieb nehmen zu können sowie deren Verteilung über automatisierte Werkzeuge zu ermöglichen. (Daya, et al., 2015)

2.2.5 Skalierung

Das Architekturmuster der Microservices folgt dem Würfelmodell der Skalierung von Abbott und Fisher (2010). Sie beschreiben drei Skalierungsmöglichkeiten für Organisationen, aber auch technische Anwendungen anhand der Achsen aus Abbildung 2-3.

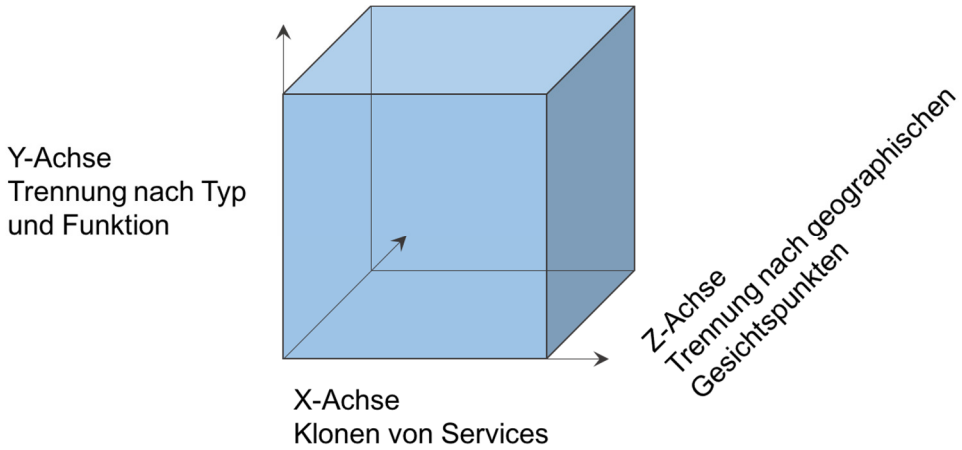


Abbildung 2-3: Würfelmodell der Skalierung nach Abbott und Fisher (2010, S. 334)

Nach Abbott und Fisher ist die einfachste Methode zum Skalieren jene entlang der X-Achse. Diese Skalierung entsteht durch das Klonen von Arbeitseinheiten, wobei jede über das gleiche Set aus Wissen und Werkzeugen verfügt und somit die ihr zugeteilte Arbeit

ausführen kann. Ungeachtet der Tatsache, dass in menschlichen Organisationen Unterschiede in der Bearbeitung, zum Beispiel durch eine effizientere Abarbeitung, auftreten können, kann die Arbeit gleichmäßig auf alle Arbeitseinheiten verteilt werden und, wenn Bedarf besteht, eine neue Arbeitseinheit hinzugefügt werden. Im Anwendungsfall von Monolithen und Microservices bezieht sich dieses Vorgehen der Skalierung auf den monolithischen Ansatz. Hier wird bei erhöhtem Ressourcenbedarf das bestehende System geklont und als zusätzliche Arbeitseinheit verwendet.

Microservices bewegen sich primär in der Y-Achse, ihre Vorteile aus dieser Skalierung können aber auch auf die Z-Achse ausgeweitet werden. Die Y-Achsen-Skalierung beschreibt das Trennen eines Systems in gleiche Teile. Langsame Teile werden ebenso wie schnelle zu Subsystemen zusammengefasst, damit diese effizienter arbeiten können. Im menschlichen Umfeld bedeutet dies die Bildung von Teams von SpezialistInnen, im technischen Umfeld das Abspalten von Funktionen, welche unabhängig voneinander ausgeführt werden können. Diese Art der Skalierung ist das Hauptaugenmerk der Microservices. Durch die Trennung anhand von Funktionen kann in weiterer Folge auch entlang der Z-Achse skaliert werden. Diese Skalierung betrifft die Trennung nach geographischen Gesichtspunkten. Die Kombination der Y- und Z-Achsen-Skalierung ermöglicht die moderne Entwicklung mit verteilten Teams, welche für ihre entwickelten Teilsysteme oder Funktionen die Verantwortung über ihre gesamte Lebensdauer übernehmen. (Abbott & Fisher, 2010)

2.2.6 Architektur

Die Architektur von Microservices kann in zwei Themenbereiche unterteilt werden. Zum einen in die interne Architektur, wie wird ein Microservice entwickelt, und in die externe Architektur, wie fügt sich ein Microservice in eine Gesamtanwendung ein. Fildebrandt (2016) schreibt in Bezug auf die interne Architektur eines Microservices, dass nach wie vor alte und bewährte Prinzipien der objektorientierten Programmierung auf Microservices angewandt werden können. So sollen Microservices, um ihren Vorteilen gerecht zu werden, in Anlehnung an OO-Entwurfsmuster ebenfalls nach folgenden Richtlinien entwickelt werden:

- „Single Responsibility Principle (SRP): Eine Klasse soll nur eine Aufgabe erfüllen.
- Open Close Principle (OCP): Eine Klasse soll offen für Erweiterungen sein, aber geschlossen für Modifikationen.
- Liskov Substition Principle (LSP): Für einen Verwender soll eine Instanz einer Superklasse immer durch seine Subklassen ersetzbar sein.
- Interface Segregation Principle (ISP): Ein Verwender soll eine Schnittstelle angeboten bekommen, die nur seiner Aufgabe entspricht.
- Dependency Inversion Principle (DIP): Die Abhängigkeit soll nur von höheren Modulen zu niedrigeren gehen. (Fildebrandt, 2016, Abs. 2)“

Zum Design einer externen Architektur können Abwandlungen der Enterprise Architecture Integration angewandt werden. Cockburn (2005) führt jedoch eine weitere Herangehensweise mit dem Inhalt an, die Architektur nicht mehr in drei Schichten

aufzuteilen (siehe Kapitel 2.1.2), sondern Anbindungen an Datenbanken oder auch Schnittstellen zu BenutzerInnen als Ports und Adapter in der Hexagonalen Architektur zu betrachten. Dieser Gedanke wurde weitergesponnen (Young, 2014) und resultiert in einer gesamtheitlichen Architektur, dargestellt in Abbildung 2-4. Hierbei können ein oder mehrere Microservices die Geschäftslogik abbilden und die Kommunikation mit Drittsystemen wird wiederum in Microservices gekapselt. Dies erhöht die Flexibilität hin zu externen KonsumentInnen der Geschäftslogik. In der Abbildung wird die Geschäftslogik im Zentrum symbolisiert durch die enthaltenen Microservices. Um diese Geschäftslogik bilden weitere Microservices Schnittstellen zu Fremdsystemen. Zusammengefasst in der Schnittstelle im rechten oberen Bereich sind Microservices zur Anbindung von anderen Systemen im Geschäftsumfeld, in diesem Fall ein Produktionsplanungssystem (PPS) und ein Customer-Relationship-Management-System (CRM). Die Schnittstellen zu einer oder mehreren Datenbanken ist im unteren Bereich zusammengefasst.

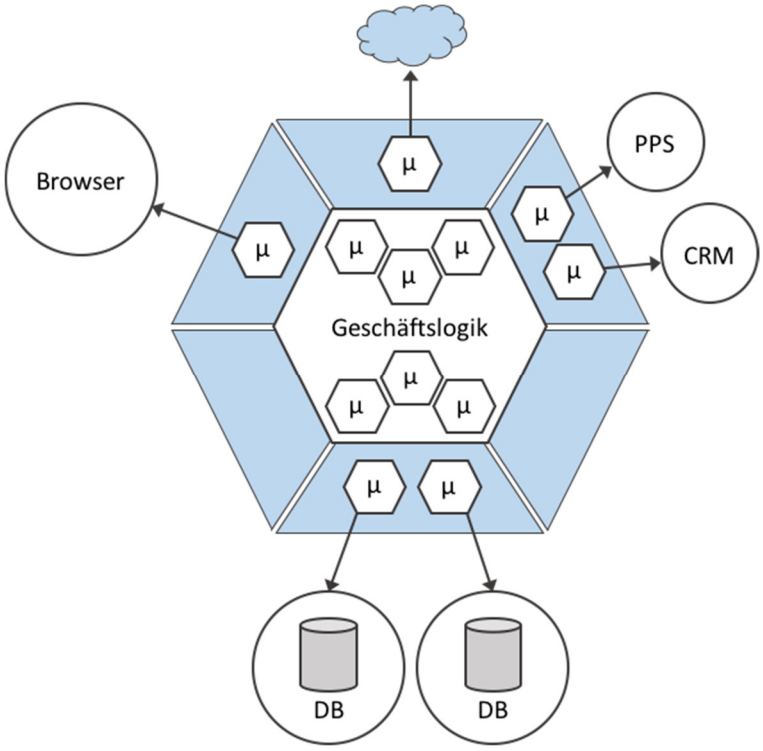


Abbildung 2-4: Microservices in einer Hexagonalen Architektur in Anlehnung an Young (2014)

Wichtig ist die organisatorische Aufteilung, da das Verschieben von Funktionalität in ein anderes Microservice Mehraufwand bedeutet. Nicht nur die Synchronisation verschiedener Projektteams, sondern auch mögliche Implementierungsunterschiede, in Bezug auf das eingesetzte System oder die verwendete Programmiersprache, sind zu bewerten. Somit ist die Wahl einer geeigneten Architektur, welche auch Änderungen im Verlauf der Lebenszeit eines Services zulässt, von hoher Bedeutung. (Wolff, 2016)

2.3 Zusammenhängende Disziplinen

Im Zusammenhang mit Microservices werden in der Literatur Disziplinen genannt, welche die Entwicklung des Architekturmusters beeinflusst haben oder durch welche die Notwendigkeit von Microservices entstanden ist. In diesem Kapitel wird ein Ausschnitt aus diesen Disziplinen dargestellt.

2.3.1 Domain Driven Design

Das Domain Driven Design (DDD) von Eric Evans (2004) stellt eine Methode zur Modellierung komplexer Software anhand von Geschäftsfällen dar. Das Herz des Domain Driven Design ist das Modell des Problems in einer Fachdomänen-spezifischen, der ubiquitären oder universellen Sprache. Das Domänenmodell gilt als Arbeitsgrundlage für die Programmierung und stellt damit auch einen Teil der Dokumentation dar. Deshalb ist das Modell nach Evans immer mit der Anwendung aktuell zu halten. Zusätzlich dient das Modell als Kommunikationsinstrument für EntwicklerInnen und ExpertenInnen ohne Übersetzungsnotwendigkeit der jeweiligen Fachsprache. Die gewählte Sprache im Modell soll für beide Seiten verständlich sein und stellt das Wissen beider Seiten über das Problem dar. Durch die vereinheitlichte Sprache und die gegenseitige Übereinstimmung über diese kann effektiver und effizienter am Problem gearbeitet werden, da die Barriere der Kommunikation niedrig gehalten wird oder gänzlich entfällt.

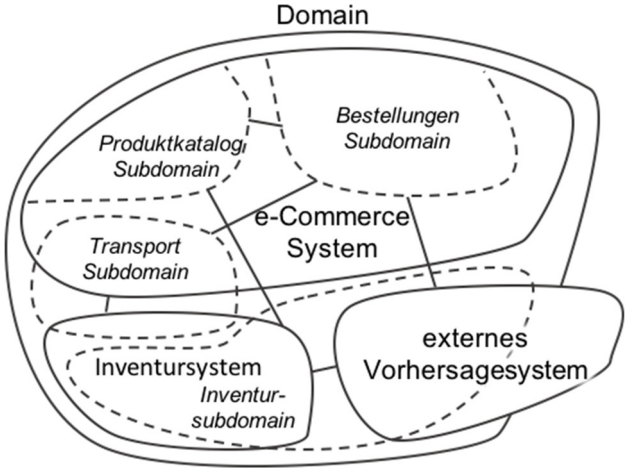


Abbildung 2-5: Fachdomäne mit Subdomänen und beschränkten Kontexten in Anlehnung an Vernon (2013, Abbildung 2.1)

Zusätzlich zur Fachdomäne wird im Domain Driven Design der beschränkte Kontext eingesetzt. Darunter versteht man die Abgrenzung der gesamten Anwendung gegen die Umwelt über definierte Grenzen. Alles in dem gewählten Kontext wird als Teil der Fachdomäne betrachtet und mit der universellen Sprache dieses Kontexts beschrieben. Das bedeutet nicht, dass die Nomenklaturen des Geschäftsfalles oder der Technologie verwendet werden müssen, sondern jene Sprache vorherrscht, auf welche sich alle Teamteilnehmer verständigt haben. (Vernon, 2013) Zusätzlich kann von einem weiteren Kontext ein Teil desselben Modells verwendet beziehungsweise implementiert werden. Dies

ist in Abbildung 2-5 dargestellt in Form von geschlossenen Kreisen und Kreisen mit unterbrochener Linie. Die gesamte Geschäftsdomäne ist zur Umwelt abgegrenzt und intern in Subdomänen unterteilt. Wie ersichtlich wird vom beschränkten Kontext „e-Commerce System“ in der Abbildung die Fachdomäne (Subdomain) des Produktkataloges, der Bestellungen und ein Teil des Transportwesens abgebildet. Um große Geschäftsfälle in ganzheitliche Modelle zu überführen, wird das Modell in beschränkte Kontexte geteilt. In jedem dieser Kontexte gilt die einheitliche, universelle Sprache. Über die Grenzen des Kontexts hinweg kann sich diese jedoch ändern. Dies hat den Vorteil, dass verteilte Entwicklungsteams untereinander dieselbe Sprache sprechen, über Teamgrenzen hinweg, sofern das Team nicht an mehreren Sub-Kontexten arbeitet, jedoch keine Festlegung auf eine Sprache notwendig ist. Die richtige Wahl der Kontextgrenzen und das dahinterstehende strategische Design findet Anwendung bei der Teilung einer Softwareanwendung in Microservices. (Evans, 2004)

2.3.2 Continuous Integration

Continuous Integration ist eine zentrale Aufgabenstellung in etablierten Entwicklungsteams. Es beschreibt das automatisierte Abholen des bearbeiteten Quellcodes aus der Quellcodeverwaltung und das anschließende Übersetzen der Anwendung auf einem Build-System. Im Anschluss werden die definierten Integrationstests durchgeführt, um so schnellstmöglich Fehler in der Entwicklung zu erkennen. Somit wird der gesamte Prozess ab dem Speichern des bearbeiteten Quellcodes in einem Versionierungssystem automatisiert, mit dem Ergebnis eines getesteten und in manchen Fällen installierten Endproduktes. Im Bereich von Microservices spielt diese Vorgehensweise eine zentrale Rolle, denn durch den hohen Grad an eigenständigen und unabhängigen Applikationen, in Form von Diensten, welche im Endeffekt zusammenarbeiten müssen, kommt dem frühzeitigen automatisierten Integrationstesten die erste Aufgabe der Qualitätssicherung zu. Diese Technik hilft, Inkompatibilitäten der Softwarekomponenten frühzeitig zu erkennen und entsprechende Gegenmaßnahmen zu setzen. (Spanneberg, 2015)

2.3.3 Service Oriented Architecture

Nach Josuttis (2007, S. 12) Definition ist die serviceorientierte Architektur (SOA) „ein Paradigma und keine konkrete Architektur: Es ist etwas, das zu einer konkreten Architektur führt.“ SOA ist ein Hilfsmittel, um im Geschäftsumfeld die richtigen Entscheidungen zu treffen, mit denen eine stabile IT-Infrastruktur aufgebaut werden kann. In den Grundsätzen unterscheidet sich die serviceorientierte Architektur nur unwesentlich von den Zielen der Microservices. SOA versucht, einzelne Teile eines Geschäftsprozesses in Form von Webservices oder ähnlichen Diensten aufzubereiten, um den gesamten Geschäftsprozess über verschiedene Computersysteme und -anwendungen hinweg abbilden zu können. Nach Wolff (2016) ist genau dies das Hindernis des Durchbruchs von SOA. Systeme, welche das gesamte Geschäftsfeld umspannen und verschiedenste Dienste im Geschäftsleben integrieren, zum Beispiel CRM- und PPS-Systeme mit kundenspezifischen Produktions-

systemen, wurden zu mächtig, umfangreich und kompliziert, und lassen deren ursprüngliche Intention der Einfachheit und Unabhängigkeit vermissen.

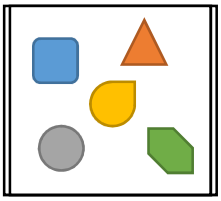
In einer serviceorientierten Architektur existiert, wie beschrieben, oftmals ein Orchestrierungssystem, um alle Services miteinander kommunizieren zu lassen. Dies kann geschehen über einen Service Bus oder ähnliche Dienste. Genau darin unterscheidet sich die SOA von Microservices. Diese sind im Gegensatz zu SOA-Diensten unabhängig im Sinne von: Jedes Service ist selbstverantwortlich für die Kommunikation nach außen und kann dadurch die Ziele der Ersetzbarkeit und losen Kopplung erreichen. (Wolff, 2016)

2.3.4 Self-Contained Systems

Das Prinzip der Self-Contained Systems (SCS) wurde von Tilkov ins Leben gerufen. Dieser spricht bei SCS von Programmeinheiten, welche das etablierte Schichtenmodell auf kleinerer Ebene umsetzen und dennoch die Vorteile von Microservices verfolgen. Im Vergleich zu Microservices werden in SCS, von der Benutzerschnittstelle bis zur Datenbank, alle Schichten des Modells umgesetzt. Der Ansatz der SCS versucht die Integration auf der Schicht der Benutzerschnittstelle herzustellen, im Gegensatz zur bevorzugten Integration auf Geschäftslogikebene der Microservices. Zusätzlich sollten SCS nicht direkt miteinander kommunizieren und müssen deshalb weiter gefasst, beziehungsweise umfangreicher sein als Microservices. (innoQ, 2016)

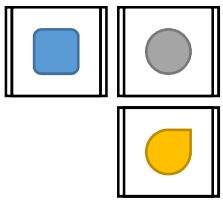
2.4 Zusammenfassung

Wie in den vorangehenden Kapiteln über monolithische Software und Microservices beschrieben, kann man eine einfache Unterscheidung (Lewis & Fowler, 2014) zwischen diesen Architekturtypen feststellen. Die Grafiken aus Abbildung 2-6 stellen Prozesse und Funktionen dar und wie diese auf Computersystemen skalieren.



Ein Monolith vereint alle Funktionen der Anwendung in einem einzigen Prozess.

Während eine Microservices Architektur jedes Stück Geschäftslogik in einen eigenen Prozess kapselt.



Der Hauptunterschied liegt in der Handhabung der Skalierung. Der Monolith kann nur durch Duplizieren der Anwendung als Ganzes skalieren. Dadurch werden womöglich auch Ressourcen bereitgestellt, die im Skalierungsfall nicht benötigt werden.

Die Microservices skalieren dagegen je nach Bedarf. Nimmt ein Prozess mehr Ressourcen in Anspruch als ein anderer, so wird nur dieser dupliziert. Durch diesen Ansatz werden die vorhandenen Hardware-Ressourcen besser genutzt und im Idealfall weniger Hardware benötigt.

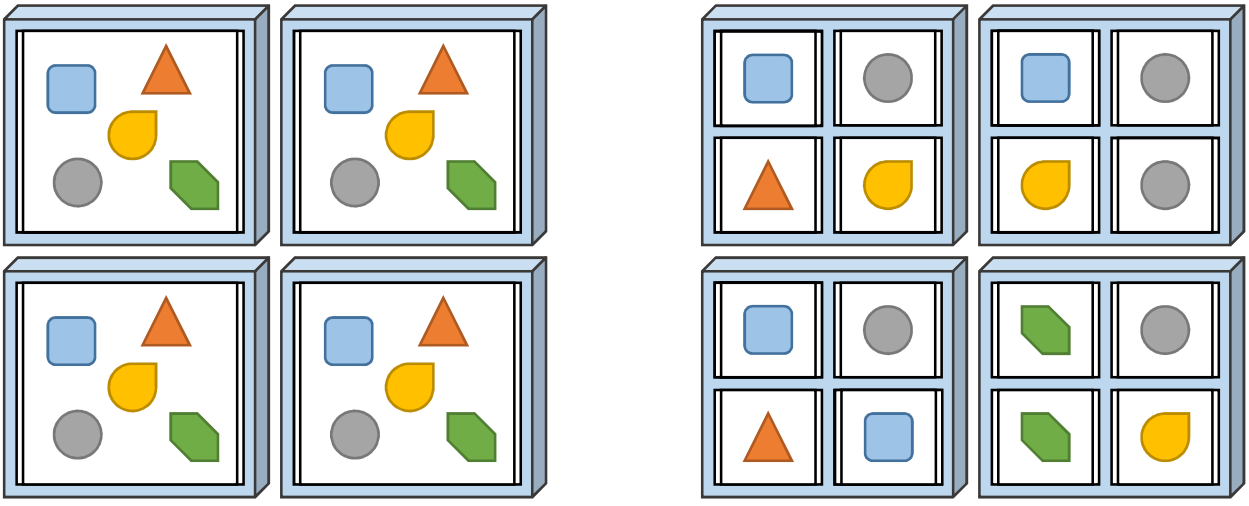


Abbildung 2-6: Zusammenfassung der Unterschiede zwischen Monolithen und Microservices (Lewis & Fowler, 2014, Abb. 1)

3 VOM MONOLITHEN ZUM MICROSERVICE

In vielen Szenarien der Softwareentwicklung werden Altsysteme gewartet und erweitert. In diesem Umfeld gibt es die Möglichkeit, Monolithen mit Microservices zu erweitern, um neue Funktionalität hinzuzufügen oder die Altanwendung schrittweise zu ersetzen (Richardson, 2016). Dieses Kapitel gibt einen Ausblick und beschreibt Strategien zum Hinzufügen von Microservices beziehungsweise zum Trennen von monolithischer Software in Microservices.

3.1 Trennen eines Monolithen

Die Trennung eines Monolithen kann verschiedene Ansätze verfolgen. Hier dargestellt ist eine Kombination der Vorgehensweisen nach Newman (2015) und Richardson (2016), welche drei Strategien zur allgemeinen Trennung eines Monolithen sowie eine Darstellung der Trennung von Daten und Quellcode beinhaltet.

3.1.1 Service hinzufügen

Die erste Strategie von Richardson (2016) beschreibt eine Vorgehensweise, wenn eine neue Funktion einem bestehenden Monolithen hinzugefügt werden soll. Nach seiner Ausführung sollen neue Funktionen nicht den Monolithen vergrößern und dessen Codebasis erweitern, sondern neue Funktionen von Grund auf nach den Richtlinien und Prinzipien der Microservices entwickelt werden. Dazu muss dem Monolithen ein Request-Router vorgeschaltet werden, um eingehende Anforderungen von Clientanwendungen zu untersuchen und je nach Anwendungsfall dem Monolithen oder dem neuen Microservice zur Bearbeitung weiterzureichen. Dieser Vorgang ist in Abbildung 3-1 schematisch dargestellt. Des Weiteren wird noch ein Verbindungscode zur Kommunikation zwischen dem Service und der alten Anwendung benötigt. Der Verbindungscode hat die Aufgabe, zwischen den Datenmodellen der zwei Anwendungen zu übersetzen, sodass nach den Ansichten des Domain Driven Design (siehe Kapitel 2.3.1) die Grenzen der Modelle gewahrt bleiben.

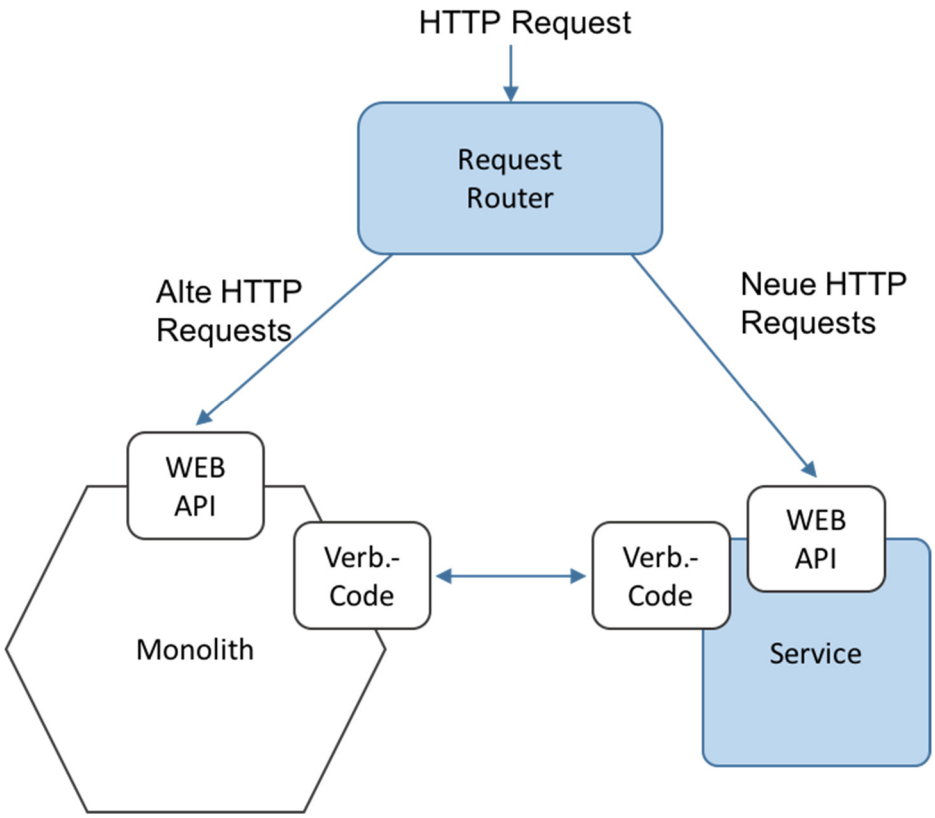


Abbildung 3-1: Trennungsstrategie 1 in Anlehnung an Richardson (2016)

3.1.2 Trennung von Frontend und Backend

Nach Richardson (2016) besteht „eine typische Geschäftsanwendung aus drei Teilen“, welche dem Schichtenmodell nach Reenskaug (1979) ähnelt. Sie besteht aus einer Präsentationsschicht, in einer Webanwendung ist das die HTML-Benutzerschnittstelle, der Schicht der Geschäftslogik und der Datenbankzugriffsschicht. Die Trennung von Frontend und Backend beschreibt die Trennung der Präsentationsschicht von der Geschäftslogik und dem Datenbankzugriff. Dieser Schnitt wird als effektivster betrachtet. In diesem besteht bereits die Kapselung der Funktionen, welche auf Netzwerkzugriffe umgebaut werden kann. Somit können ab dieser Trennung die Darstellung und die Geschäftslogik getrennt entwickelt und vor allem installiert werden, ohne sich gegenseitig zu beeinflussen. Die Abbildung 3-2 stellt den Trennungsvorgang schematisch dar. Die Benutzerschnittstelle wird von der ursprünglichen Anwendung gelöst und als eigenständiger Dienst betrieben. Es ist jedoch nur ein Teil der Lösung, denn es löst nicht das Problem einer großen Quellcodebasis auf Seiten der Geschäftslogik.

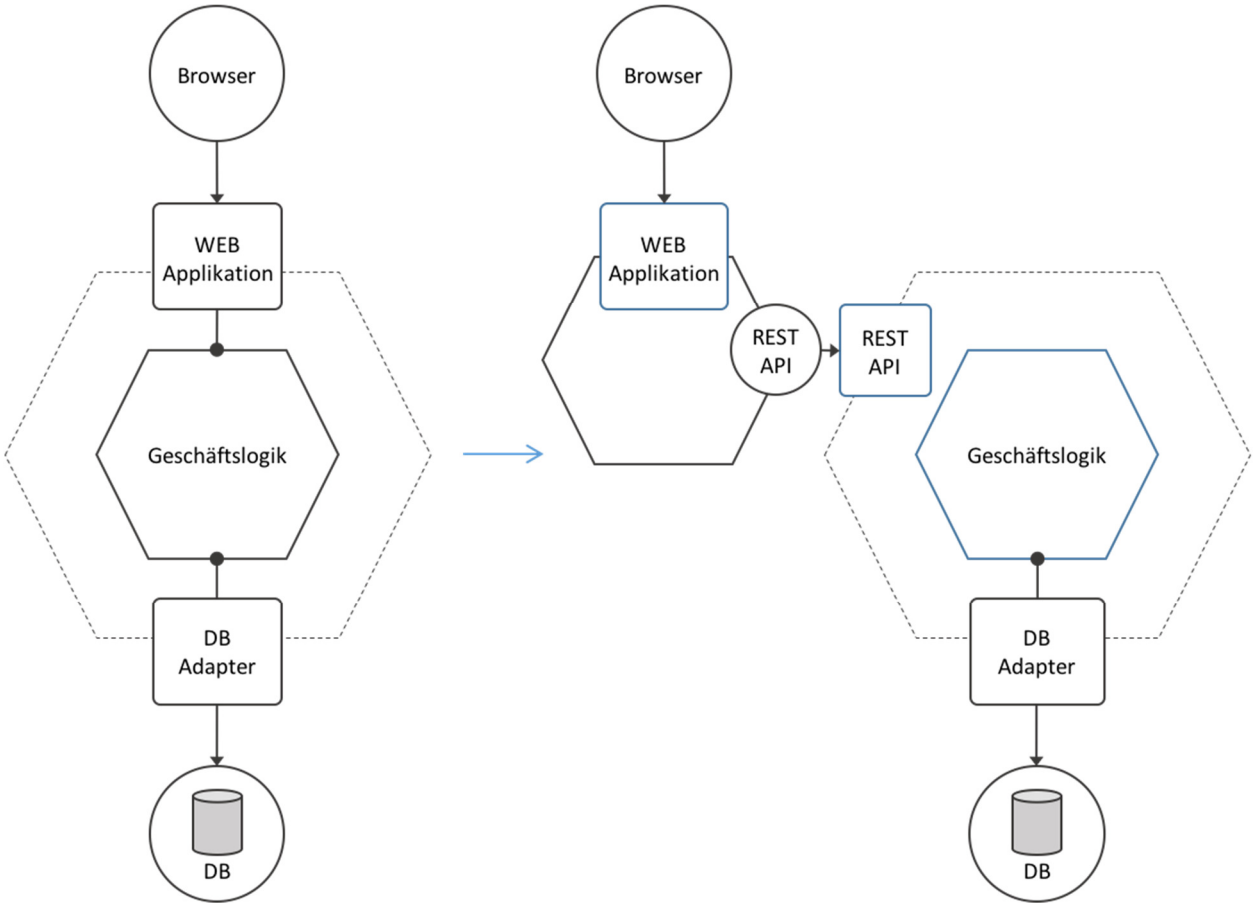


Abbildung 3-2: Trennungsstrategie 2 in Anlehnung an Richardson (2016)

3.1.3 Extrahieren von Funktionen

Die dritte Trennungsstrategie eines Monolithen in Microservices (Richardson, 2016) ist die tatsächliche Entnahme von Funktionalität aus dem Monolithen und der Bereitstellung dieser in einem separaten und autonomen Service. Dies wird solange wiederholt, bis der Monolith nicht mehr vorhanden ist oder der Rest selbst einen Microservice darstellt. Welche Module des Monolithen in welcher Reihenfolge extrahiert werden sollen, kann anhand verschiedener Faktoren bestimmt werden. Zu Beginn sollte mit kleinen, abgekapselten Modulen begonnen werden, um Erfahrung zu sammeln. Sobald das Entwicklungsteam dazu fähig ist, sollte jenes Modul entnommen werden, welches den größten Mehrwert für die Organisation darstellt. Dies kann ein Modul sein, welches sich häufig ändert oder eines, deren technische Hürde bereits zu groß für das Entwicklungsteam wurde. Eine weitere Möglichkeit ist ein Modul zu extrahieren, welches die Skalierungsgrenzen des Monolithen bereits erreicht.

Zur Trennung der Module von der monolithischen Anwendung müssen zwei Aspekte genauer betrachtet werden. Wie werden die Daten getrennt und wie trennt man schließlich den Quellcode selbst. Diese Fragen werden in den Kapiteln 3.1.3.1 und 3.1.3.2 behandelt. Die Reihenfolge dieser Trennung schlägt Newman (2015) in Anlehnung an Ambler und Sadalage (2006) mit der Schematrennung vor, also Datenbanktrennung, bevor die Services getrennt werden.

3.1.3.1. Trennung der Daten

Um den Anforderungen der Microservices nach Autonomie für den Quellcode, die Funktionen und die Daten gerecht zu werden, muss auch die Datenbank beziehungsweise der Datenspeicher getrennt werden. In monolithischen Anwendungen ist dies historisch bedingt meist eine relationale Datenbank. Der erste Schritt ist die Trennung der Datenbanktabellen und der darin gespeicherten Daten.

Fremdschlüsseltrennung

Die Fremdschlüsseltrennung beschreibt den Vorgang der Trennung von miteinander verknüpften Tabellen auf Fremdschlüsselebene in unabhängige Tabellen. Dies muss geschehen anhand der Trennung nach den Kontextgrenzen. Zur Trennung sind zwei Schritte notwendig. Zum einen muss auf Quellcodeebene sichergestellt werden, dass Klassen aus unterschiedlichen Kontexten (begrenzte Kontexte nach dem Domain Driven Design aus Kapitel 2.3.1) nicht mehr direkt auf Daten in Tabellen zugreifen, deren Kontextzugehörigkeit nicht gegeben ist. Zum anderen müssen die Mechanismen des Datenbankmanagement-systems aufgehoben werden, das bedeutet, Fremdschlüssel werden gelöscht. Damit die Kontexte und deren Inhalte miteinander kommunizieren können, wird eine Programmier-schnittstelle zwischen den Kontexten erstellt. Dies ist ein Vorläufer für die Kommunikation über das Netzwerk der Microservices, wird aber in diesem Schritt noch innerhalb des Monolithen verwendet. (Newman, 2015)

Statische Daten

Die zweite zu stellende Frage ist: Wie wird mit statischen Daten umgegangen? Newman (2015) beschreibt drei Möglichkeiten, um mit beispielsweise Länderdaten umzugehen, welche von vielen Teilen des Systems verwendet werden. Der erste Zugang ist die Duplikation der Tabellen und somit der Datenhaltung in jedem Kontext. Dies führt zu einem erhöhten Aufwand bei der Aktualisierung der Daten, denn die Konsistenz der Daten muss weiterhin gewährleistet werden. Die zweite Option stellt die Umwandlung in Quellcode dar. Dies kann passieren als statischer Code oder durch das Einlesen von Konfigurationsdateien. In jedem Fall, schreibt Newman (2015), gestaltet sich das Aktualisieren und Installieren von Software über die Continuous Integration Pipeline (siehe Kapitel 2.3.2) als einfacher gegenüber einer Live-Datenbankaktualisierung. Die dritte und zugleich sauberste Lösung stellt die Ausgliederung in ein eigenes Service dar. Wenn die Daten einer hohen Änderungs-frequenz unterliegen, ist dies die sinnvollste Option.

Geteilte Daten und Tabellen

Diese Ansätze stellen ein Problem der falschen Aufteilung der Domänenobjekte dar. Das kann geschehen sein durch verteilte Teams oder durch funktional nach Expertise getrennte Teams. Hier werden Domänenobjekte zum Beispiel in der Datenbank richtig modelliert, der

darauf zugreifende Quellcode ist dies jedoch nicht. Der richtige Ansatz der Trennung hierfür ist die Einführung derselben Domänenobjekte im Quellcode und die Abstrahierung der bisherigen Zugriffe auf die Datenbank über eine Schnittstelle im Quellcode. Damit werden Kontexte geschaffen, welche eine bestimmte Domänenaufgabe erfüllen.

Das zweite Problem ist eine zu grobe Modellierung der Domänenobjekte. Als Beispiel dafür sei eine Artikeltabelle angeführt. Wenn die Modellierung auf einem zu hohen Abstraktionsniveau stattgefunden hat, kann es sein, dass in dieser Tabelle Informationen aus verschiedenen Bereichen vorhanden sind. Zum Beispiel kann festgehalten sein, in welcher Lagerreihe der Artikel liegt oder welchen Preis der Artikel hat. In diesem Fall muss die Artikeltabelle getrennt werden und nur die relevanten Informationen für den jeweiligen abgegrenzten Kontext in neue Tabellen überführt werden. So kann diese Trennung in mehreren Tabellen wie Lagerartikel und Bestellartikel enden, denn für die Lagerverwaltung spielt der Preis keine Rolle. (Newman, 2015)

Vorgehen

Ambler und Sadalage (2006) definieren einen Prozess zum Refactoring der Datenbank, welcher sich Methoden zur Trennung beziehungsweise dem evolutionären Entwickeln der Datenbank widmet. Sie teilen den Prozess in zehn Schritte: dem Review, ob die geplante Änderung überhaupt notwendig ist, der Methodenauswahl, der Übergangsphase in einem Umfeld mit mehreren Anwendungen, dem Testen, der eigentlichen Änderung, dem Migrieren der Daten, der Änderung der Zugriffsroutinen, Regressionstests sowie Versionierung der Änderung und der Veröffentlichung der Änderung für andere Anwendungen und Parteien. Newman (2015) betrachtet allgemeiner die Datenbanktrennung vor der Trennung der Funktionen beziehungsweise Kontexte innerhalb des Monolithen und später die Trennung des Quellcodes in separate Microservices als zu verfolgendes Vorgehensmodell. Dies ist in Abbildung 3-3 dargestellt.

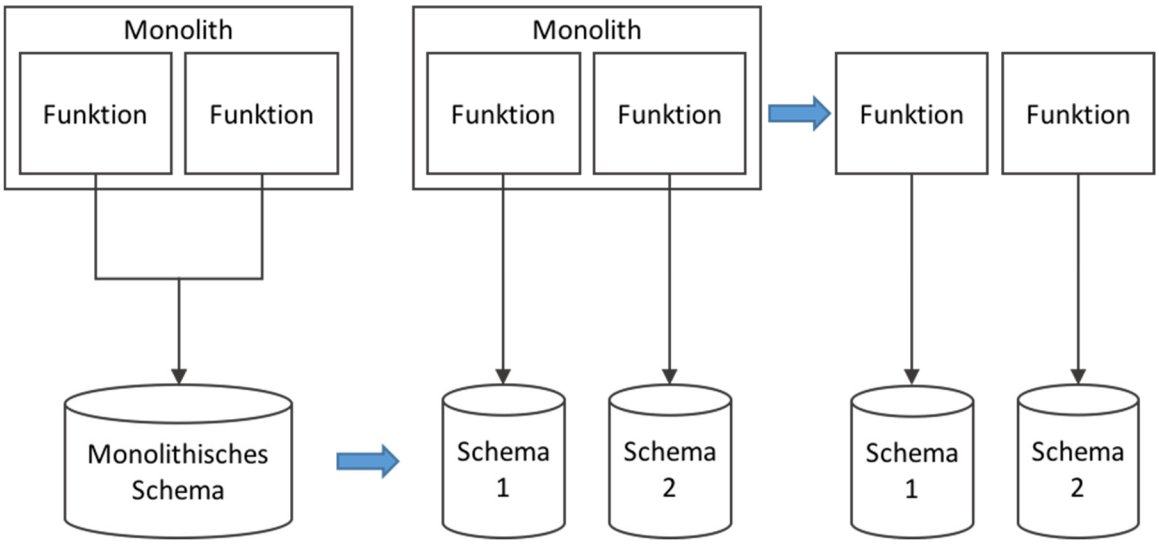


Abbildung 3-3: Vorgehen von der Trennung der Daten bis zur Trennung des Monolithen in Anlehnung an Newman (2015, S. 166)

3.1.3.2. Trennung des Quellcodes

Bei der Trennung des Quellcodes müssen drei Bereiche beachtet werden. Als erstes müssen die richtigen Schnitte gesetzt werden. Hier kann der Ansatz des Domain Driven Design verfolgt werden, um Kontexte gegeneinander abzugrenzen und fachliche Zusammengehörigkeit zu identifizieren. Des Weiteren ergibt sich daraus die Entscheidung, ob sich ein System mit vertretbarem Aufwand in Microservices überführen lässt oder nur Teilstrategien umgesetzt werden (Kapitel 3.1.1 und 3.1.2). Zu guter Letzt müssen bisherige Transaktionen, welche in der monolithischen Anwendung direkt im Prozess abgehandelt wurden, fortan über die Anwendungsgrenzen hinweg gesichert werden. Hierfür gibt es Ansätze aus der Literatur, auf welche in diesem Abschnitt eingegangen wird.

Eventually Consistent

Transaktionsorientierte Problemlösungen wurden bereits im Umfeld der serviceorientierten Architekturen gefunden. In manchen Situationen wird dabei darauf verzichtet, die Transaktion wirklich durchzuführen, sondern für den Anwendungsfall reicht es zu wissen, dass zum Beispiel ein Event registriert wurde, welches später verarbeitet wird. Das bedeutet, dass die Anwendung zu einem späteren Zeitpunkt wieder konsistent wird. Ein Beispiel hierfür ist die Annahme einer Bestellung in einem Onlineshop. Die Bestellung wurde erfolgreich registriert und es gibt genug Waren auf Lager, jedoch die zeitgleiche Weitergabe des Auftrages an das Versandteam konnte noch nicht gewährleistet werden. Hier kommen abhängig von der verwendeten Technologie Message-Queues zum Einsatz, welche die Verarbeitung der Weitergabe zu einem späteren Zeitpunkt gewährleisten. (Newman, 2015)

2-Phasen-Commit

In Problemstellungen, in denen eine verspätete Verarbeitung eines Auftrages nicht möglich ist, zum Beispiel einer Hotelbuchung mit passendem Flug, kann das Muster des 2-Phasen-Commits eingesetzt werden. Bereits Gray (1978) beschreibt wie verteiltes Transaktionsmanagement in Datenbanken funktionieren kann. Diese Vorgehensweise findet nun auch in Microservices Verwendung. Beim 2-Phasen-Commit sendet der Transaktions-koordinator jedem/jeder TeilnehmerIn eine Vote-Request-Nachricht. Jeder/jede TeilnehmerIn, der/die die Transaktion lokal durchführen kann, antwortet darauf mit Ja, die anderen mit Nein. Der Koordinator sammelt alle Antworten und entscheidet danach, ob die Transaktion durchgeführt wird. Zum Abschluss bekommen alle Ja-TeilnehmerInnen eine Commit- oder Abort-Nachricht zugesendet (Bernstein, Hadzilacos, & Goodman, 1987). Durch diesen Mechanismus kann auch in verteilten Systemen Transaktionssicherheit gewährleistet werden.

3.2 Verbinden von Microservices

Microservices lassen sich auf unterschiedlichen Ebenen verbinden. Wolff (2016) beschreibt diese als Integration auf Ebene der Benutzerschnittstelle (User Interface, UI), auf der Ebene der Logik oder auf der Ebene der Datenhaltung (Abbildung 3-4). Je nach Ebene der Integration gibt es verschiedene Technologien, welche zum Datenaustausch eingesetzt werden können. Auf der UI-Ebene kann das zweite Microservice, am Beispiel einer Webanwendung, als weitere Seite eingebunden werden oder als weiteres Modul in eine Single Page Application.

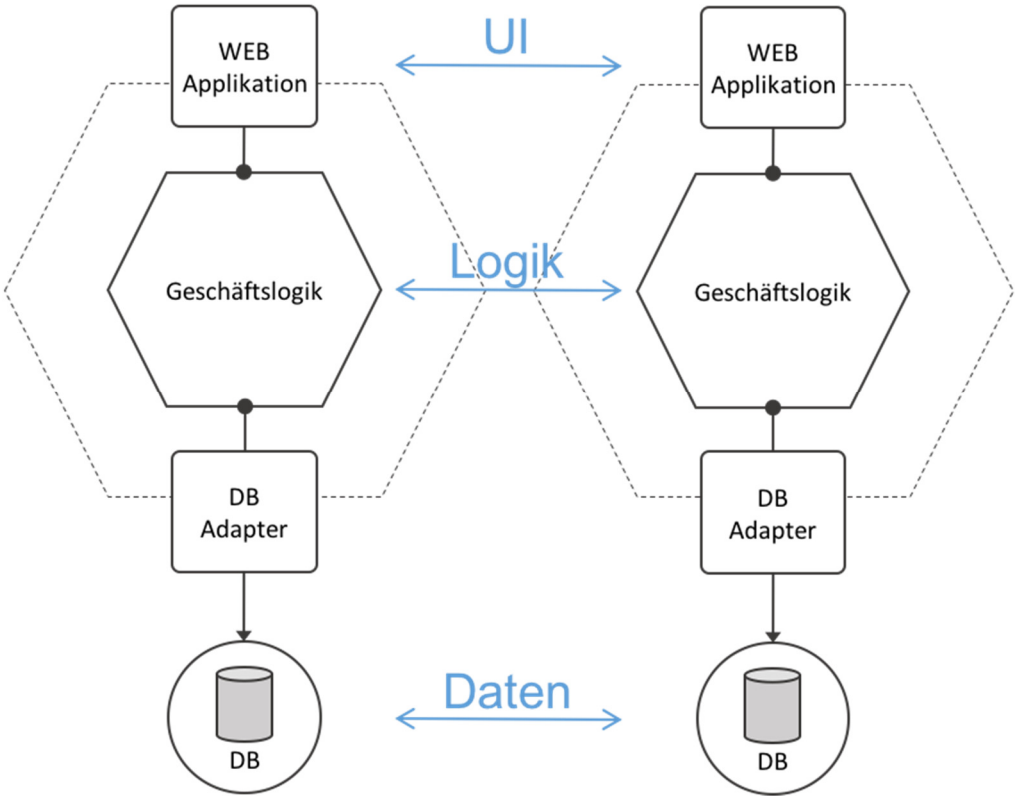


Abbildung 3-4: Integrationsebenen von Microservices in Anlehnung an Wolff (2016, Abb. 9-1)

Die Ebene der Geschäftslogik bietet sowohl die Möglichkeit eines synchronen Datenaustausches, als auch die Möglichkeit einer zeitlich unabhängigen Form der Kommunikation. Mögliche Technologien sind unter anderem der Einsatz von Remote Procedure Calls in einer konkreten Implementierungsform, der Einsatz von Webtechnologien in Form des Representational State Transfer (siehe Kapitel 3.2.2) oder der Einsatz von nachrichtenbasierten, asynchronen Systemen. Schließlich können Microservices auch über eine Datenablage in Form einer Datenbank oder dem Dateisystem miteinander in Verbindung stehen.

3.2.1 Remote Procedure Calls

Remote Procedure Calls (RPC) sind das Gegenstück zu lokalen Funktionsaufrufen über Prozessgrenzen hinweg. Die zwei beteiligten Prozesse können auf demselben System betrieben werden, aber auch auf unterschiedlicher Hardware über das Netzwerk miteinander

kommunizieren. RPCs kapseln hierbei eine aufwendige Netzwerkkommunikations-implementierung in synchrone Funktionsaufrufe zu anderen Prozessen. Wie bei lokalen Funktionsaufrufen werden bei RPCs Parameter an die Gegenstelle übergeben und der aufrufende Quellcode blockiert solange, bis eine Antwort vom aufgerufenen Prozess zurückgeliefert wird. Es existieren zahlreiche Implementierungen für RPCs für gängige Programmiersprachen. (Marshall, 1999)

3.2.2 Representational State Transfer

Representational State Transfer (REST) ist eine Kommunikationstechnik auf Basis von HTTP und verwendet dessen Schlüsselwörter zur Kommunikation. Jeder Aufruf, sei es, um Daten zu erhalten oder Funktionen anzustoßen, wird mit REST über einen Uniform Resource Locator (URL) oder einen Uniform Resource Name (URN) definiert. Es gibt eine strikte Trennung zwischen Client und Server, mit dem Ergebnis, dass der Server die Datenhaltung übernimmt und der Client diesen nur über URL-Aufrufe abfragen kann. Dafür muss der Server zustandslos agieren um vorgeschaltetes Caching oder Load-Balancing zu ermöglichen. REST gilt als einfache Alternative zu aufgeblasenen Simple Object Access Protocol (SOAP) Nachrichten, da der Fokus auf Einfachheit gelegt ist. (Fielding, 2000)

Definiert sind alle möglichen Zugriffsmethoden für REST im Request for Comments (RFC) des Hypertext Transfer Protocol - HTTP/1.1 (RFC 2616, 1999). Die Zugriffsmethoden können eingeteilt werden in sichere und idempotente Methoden.

Methode	Beschreibung	Sicher	Idempotent
GET	Liest Daten und retourniert diese an den Anfrager.	Ja	Ja
HEAD	Liest Daten, retourniert jedoch nur den Header, nicht die Daten im Body.	Ja	Ja
POST	Entität in der Nachricht soll zu den Entitäten der URI hinzugefügt werden.	Nein	Nein
PUT	Entität in der Nachricht soll zu den Entitäten der URI hinzugefügt werden, wenn diese bereits existiert, so ist die neue Entität als modifizierte Version zu betrachten.	Nein	Ja
DELETE	Löscht Daten, welche durch die URI identifiziert sind.	Nein	Ja
OPTIONS	Liefert alle möglichen Methoden des Servers.	Ja	Ja

Tabelle 3-1: REST-Methoden nach dem RFC 2616 (1999)

Die Tabelle 3-1 listet die Zugriffsmethoden und deren Einteilung in Sicher und Idempotent auf. Der Standard beschreibt GET und HEAD als sichere Methoden, welche per Definition nur das Lesen und Zurückliefern der gelesenen Daten beinhaltet. Als EntwicklerIn soll, auch wenn Frameworks dafür Möglichkeiten bieten, davon Abstand genommen werden, Seiteneffekte für diese Methoden zu implementieren. Idempotente Methoden können

Seiteneffekte aufweisen, jedoch haben nach dem RFC alle N > 0 gleichen Anfragen denselben Seiteneffekt wie eine einzelne Anfrage. Dazu zählen die Methoden GET, HEAD, PUT und DELETE sowie OPTIONS. Letztere gibt auf Anfrage die möglichen Methoden des Servers preis.

3.2.3 Nachrichten

Diese Art der Kommunikation setzt Nachrichten und Nachrichten-Systeme ein und gewährleistet dadurch einen Teil der Vorteile von Microservices: die Ausfallsicherheit, sowie einen hohen Grad an Skalierbarkeit. Nachrichten-Systeme bringen selbst Vorteile gegenüber synchronen Technologien zum Datenaustausch mit:

- Durch die Verwaltung der Nachrichtenwarteschlange durch ein explizites System können Nachrichten übertragen werden, auch wenn zum Zeitpunkt des Sendens der/die EmpfängerIn nicht verfügbar ist. Das System gewährleistet die Zustellung der Nachrichten, sobald das Transportmedium wieder einsatzfähig ist.
- Nachrichtensysteme und darauf aufsetzende Anwendungen sind mit dem Hintergedanken von hohen Latenzen entwickelt worden und sind daher in unsicheren Übertragungsmedien zu bevorzugen.
- Nachrichtensysteme können die Übertragung und die Verarbeitung gewährleisten und bieten, je nach Implementierung, die Möglichkeit der Transaktionsorientierung. Das bedeutet, dass einzelne Systeme die Orchestrierung von Transaktionen über mehrere Microservices übernehmen können.
- Durch die Anwendung von Topics und Warteschlangen bieten Nachrichtensysteme die Möglichkeit, dass SenderIn und EmpfängerIn entkoppelt sind und es zu einem/einer SenderIn auch mehrere EmpfängerInnen geben kann. Dadurch kann die Skalierbarkeit gewährleistet werden. Wenn sich eine Warteschlange mit Nachrichten füllt und ein/eine EmpfängerIn die Abarbeitung dieser nicht bewerkstelligen kann, gibt es die Möglichkeit, einen/eine zweiten/zweite EmpfängerIn auf diese Warteschlange zu registrieren, um die Abarbeitung der Nachrichten mit zwei Arbeitseinheiten zu ermöglichen.

Zu den genannten Vorteilen gesellt sich ein erheblicher Nachteil. Für den Einsatz eines nachrichtenbasierten Systems muss im Gesamtkonstrukt von mehreren Microservices zusätzlich der Betrieb und Einsatz des Nachrichtensystems als zentrale Verteilerstelle gewährleistet werden. (Wolff, 2016)

3.2.4 Datenreplikation

Ein gemeinsamer Datenspeicher für mehrere Microservices ist nach Ansicht von Wolff (2016) nicht zu empfehlen, da die Nachteile überwiegen. Der Hauptnachteil ist die gefährdete Unabhängigkeit der Microservices. Datenformate können nicht effizient geändert werden, da meist viele verschiedene Systeme darauf zugreifen. Auch können obsolete

Änderungen kaum wieder entfernt werden, wenn nicht sichergestellt ist, dass kein Microservice mehr darauf zugreift.

3.3 Zusammenfassung

Dieses Kapitel widmete sich dem Prozess der Umwandlung einer monolithischen Anwendung zu einer Anwendung mit Microservices. Dazu wurden Methoden zur Trennung eines Monolithen dargelegt und Techniken zur Integration von Microservices zu einer Gesamtanwendung.

Die Trennung kann durch unterschiedliche Anforderungen ausgelöst sein. Einerseits durch die Notwendigkeit von neuer Funktionalität in der Anwendung, sei es durch einen neuen Geschäftsfall oder auch durch eine neue Zielplattform. Andererseits kann eine Trennung auch durch Probleme der Wartung einer vorhandenen Anwendung entstehen. In diesem Fall kann reduzierend vorgegangen werden, indem Funktionen Schritt für Schritt aus der ursprünglichen Anwendung herausgelöst und als eigenständige Dienste wieder in Betrieb genommen werden. Wichtig ist in diesem Teil des Prozesses die Betrachtung der Anwendung selbst, als auch der benötigten und beteiligten Daten.

Zur anschließenden Verbindung der entstandenen Microservices können abhängig von der vorhandenen Expertise und den Anforderungen des verbleibenden Monolithen unterschiedliche Technologien eingesetzt werden. Genannt wurden zum Beispiel Remote Procedure Calls oder die im Web verbreitete Technologie REST. Ebenso können Nachrichten-Systeme zur asynchronen Integration der Dienste eingesetzt werden.

4 FORSCHUNGSSTAND

Dieses Kapitel widmet sich aktuellen Studien und Veröffentlichungen zu den Themen Microservices und Softwarearchitekturen mit einem Effekt auf Codequalität im Allgemeinen. Dazu wurden die technischen Datenbanken von Ebsco und Emerald durchsucht. Die Ergebnisse dazu sind in den entsprechenden Unterkapiteln dargestellt. Beide Datenbanken lieferten zum Suchbegriff „microservices codequality“ keine Ergebnisse. Deshalb wurde die Suche auf die Begriffe „microservices“ und „codequality“ ausgeweitet. Um einen aktuellen Stand der Forschung zu erhalten, wurden nur Veröffentlichungen ab dem Jahr 2010 ausgewählt und deren Ergebnisse sowie Untersuchungsmethoden ausgewertet.

4.1 Selbst verwaltende Microservices

Mit dem Thema von selbst verwaltenden Microservices haben sich Toffetti, Brunner, Blöchlinger, Dudouet und Edmonds (2015) an der Züricher Hochschule für Angewandte Wissenschaften beschäftigt. Sie untersuchten, wie sich Microservices im Falle von Skalierung oder dem Betrieb auf nicht hoch-zuverlässiger Hardware selbstständig überwachen und organisieren können. Sie sind der Meinung, dass der Einsatz eines externen Managementsystemes zusätzlichen Verwaltungsaufwand mit sich bringt und zu Abhängigkeiten beziehungsweise Lock-In-Effekten von Lieferfirmen führt. Ihr Vorschlag ist ein automatisierungsfähiges System zur automatischen Überwachung von Microservices in Cloudumgebungen einzusetzen. Dazu werden selbst programmierte Komponenten wie ein Überwachungssystem, geteilte Konfigurationen und Service-Discovery oder automatisierte Deployment verwendet. Ihre Forschung ist in diesem Paper von theoretischer Natur. Sie beschreiben und konzeptionieren ein einsatzfähiges System, basierend auf verteiltem Speicher und herkömmlicher Hardware.

Parallel zum Paper wurde ein Versuchsaufbau als Experiment gestartet, welches das Ziel verfolgt, einen Prototypen der beschriebenen Architektur zu erstellen. Dieser konnte bis zum Ende des Papers noch keiner Evaluierung unterzogen werden. Der Erfolgsfaktor der von ihnen beschriebenen Architektur zur Verwaltung von Microservices ist nach ihrer Ansicht die Unabhängigkeit und Zustandslosigkeit von einzelnen Services, um Skalierung und Ausfallkompensation zu gewährleisten in Zusammenhang mit einem eng gekoppelten Überwachungssystem. (Toffetti et al., 2015)

4.2 Zukunft von Softwarearchitekturen

An der Universität „Alexandru Ioan Cuza“ in Iași, Rumänien, verfassten Strîmbei, Dospinescu, Strainu und Nistor (2015) ein Paper zur Gegenwart und Zukunft von Softwarearchitekturen. Darin vergleichen sie historische Ansätze der Softwareentwicklung bis zur Gegenwart, die noch jungen Microservices, und versuchen eine geeignete Architektur für den universitären Einsatz zu finden. In diesen Umgebungen sind Anwendungen der Kategorie „Learning Management Systems“, welche die Verwaltung der StudentInnen ermöglichen, im Einsatz. Solche Systeme haben ihrer Ansicht nach ein Problem, sie skalieren in den meisten Fällen nicht. Zu zwei Zeitpunkten im Jahr, zum jeweiligen Anmeldeschluss für das nächste Semester, sind diese Systeme zu sehr ausgelastet. Das Forschungsdesign des Teams wurde auf die Analyse von bestehenden Verwaltungssystemen ausgelegt und der Interpretation der Problemfelder aus Sicht der Geschäftsanforderungen und der Sicht der technischen Anforderungen. Im Paper wird nicht genauer darauf eingegangen, in welchem Umfang welche Software analysiert wurde. Als Resultat ihrer Forschung wird eine Anleitung dargelegt, wie zukünftige Softwarearchitekturen im universitären Bereich gestaltet werden sollen. Sie stellen fest, bestehende monolithische Anwendungen als Dienste in einer größeren Microservice-Umgebung anzusehen und über Proxyanwendungen oder -dienste einzubinden. Für neue Services, welche von einem homogenen Team mit einer gemeinsamen Basis an Infrastruktur und Programmiersprachen-kenntnissen bearbeitet werden können, biete sich nach der Einschätzung von Strîmbei et al. (2015) die serviceorientierte Architektur an.

4.3 Performanceerhalt beim Umbau zu Microservices

An der Universität von Kiel schrieb Knoche (2016) ein Paper über die Modernisierung von monolithischen Anwendungen hin zu Microservices mit dem Fokus auf dem Performanceerhalt des Ausgangssystems. Das Paper legt eine Vorgehensweise in acht Schritten dar, um Teile der monolithischen Anwendung zur Modernisierung zu identifizieren. Diese Teile werden in mehreren alternativen Formen implementiert und jeder einer Simulation von Echtdaten unterzogen und schließlich die Entscheidung getroffen, welcher getestete Teil der Anwendung ohne Verschlechterung der Performance in die Ausgangsanwendung übernommen werden kann. Der Forschungsplan des Teams sieht vor, in fünf Schritten vorzugehen. Schritt eins dient der Analyse und Studie der Literatur und der Verfeinerung des Plans. Im zweiten Schritt sind weitere Recherchen geplant, im Speziellen zum Thema der Überführung von bestehenden Anwendungen. Schritt drei und vier dienen der Implementierung der Modelle zur Simulation sowie einem Experiment mit dem Ziel, diese Modelle an einer Software zu evaluieren. Abschließend sollen die Ergebnisse dieser Forschung aufbereitet und verarbeitet werden. Das Resultat des Papers ist der Forschungsplan sowie die Grundlagen zur Vorgehensweise eines Umbaus einer Softwareanwendung. Die Abschlussergebnisse erwartet das Team nach der Bearbeitung dieser Masterarbeit des Autors.

5 CODEQUALITÄT

Die Codequalität wird als Teilaspekt der Softwarequalität angesehen, welche nach Balzert (1998, S. 257) als „die Gesamtheit der Merkmale und Merkmalswerte eines Software-Produktes, die sich auf dessen Eignung beziehen, festgelegte und vorausgesetzte Erfordernisse zu erfüllen“, definiert ist. Dabei stellen Merkmale und dessen Werte Anforderungen an das Endprodukt dar, mit deren Hilfe die Qualität des Systems bestimmt werden kann. Dieses Kapitel widmet sich der Definition von Codequalität, den Einflüssen auf Lesbarkeit und Wartbarkeit von Quellcodes und der Auswahl geeigneter Metriken zur Evaluierung von Codequalität, dem internen Blick auf Softwarequalität (ISO/IEC 25010:2011-03, 2011) im Zusammenhang mit Microservices anhand des Goal-Question-Metric-Modells.

5.1 Definition

Zur Messung von Merkmalen werden Metriken eingesetzt. Diese können in Bezug auf Software nach Liggesmeyer (2009) sowie Fenton und Bieman (2015) in drei Kategorien unterteilt werden:

- Produktmaße,
- Prozessmaße und
- Projektmaße.

Wie in Abbildung 5-1 durch Liggesmeyer (2009) erklärt, können unter die Kategorie der Produktmaße die Information der Komplexität oder die Information über den Umfang eines Softwareproduktes fallen. Sie dienen der Klassifizierung und Vergleichbarkeit von Softwareprodukten. Prozessmaße beschreiben Informationen über den Entwicklungsprozess eines Produktes und Projektmaße die Projektkostenplanung. Fenton und Bieman (2015) unterteilen die Klassifizierung der Produkte weiter in interne und externe Eigenschaften von Produkten, wobei externe Eigenschaften und Messungen Dimensionen wie Hardware und Benutzung beinhalten. Diese beeinflussen eine Messung bezüglich Stabilität und müssen berücksichtigt werden. Interne Produktmaße betreffen die internen Eigenschaften eines Softwareprodukts und können unter anderem in Qualitätsmaßen und Komplexitätsmaßen gemessen und bestimmt werden (Dumke, Foltin, Koeppe, & Winkler, 1996).

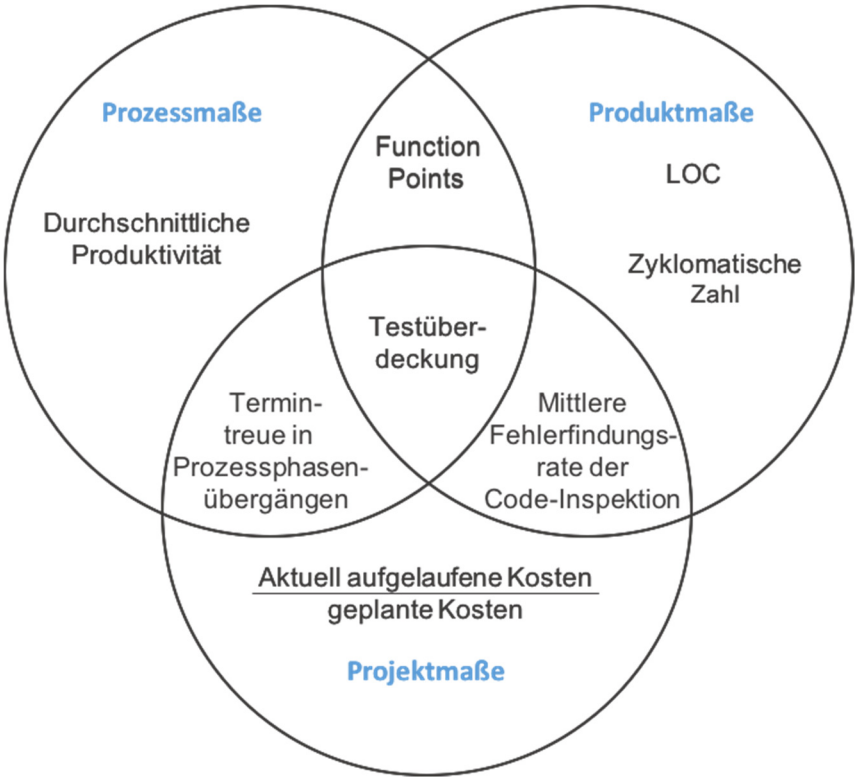


Abbildung 5-1: Software-Maße (Liggesmeyer, 2009, S. 235)

5.2 Code Metriken

Zur Messung von Software können Metriken eingesetzt werden. Dieses Kapitel spezialisiert sich auf interne Produktmaße, um die Forschungsfrage nach der Veränderung der Qualität des Quellcodes beantworten zu können. Diese Maße stammen aus der objektorientierten Metriksammlung von Chidamber und Kemerer (1994) und wurden ergänzt durch Metriken von Kan (2002).

5.2.1 Änderungsgeschwindigkeit

Die Änderungsgeschwindigkeit wird von Kan (2002) beschrieben als Mittelwert der Bearbeitungszeit aller Probleme beziehungsweise Tickets vom Zustand „Aufgenommen“ bis „Geschlossen“. Zusätzlich kann die Änderungsgeschwindigkeit für Aufwandskategorien von „Einfach“ bis „Schwierig“ erfasst und berechnet werden. Für den Fall von hohen Schwankungsbreiten kann auch der Medianwert als Berechnungsmittel zugrunde gelegt werden.

5.2.2 Lines of Code

Lines of Code ist ein Quantitätsmaß für Software und beschreibt die Anzahl der Zeilen des Quellcodes. Die Metrik stammt aus den Zeiten der Lochkarten, in denen eine Zeile der Karte gleichzusetzen war mit einer Anweisung für das Programm. In der heutigen Zeit kann auch

auf die Anzahl an Anweisungen im Programm, genannt Number of Statements (NOS), zurückgegriffen werden. (Sneed, Seidl, & Baumgartner, 2010)

Der Einsatz der Lines of Code ist auch aus empirischer Sicht interessant, da durch Forschungen belegt wurde, dass es einen statistischen Zusammenhang zwischen der einfachen Codezeilen-Messung und den aufwändigeren Berechnungen nach McCabe existiert. (Jay, et al., 2009)

Die Entwicklungsumgebung Visual Studio von Microsoft in der Version 2015 zählt Codezeilen Programmiersprachen unabhängig in Form von Anweisungen im IL-Code. Somit können bei dem Einsatz von C# im Vergleich zu VB.NET keine Abweichungen auftreten. (Microsoft, 2016)

5.2.3 Method Lines of Code

Die Metrik der Codezeilen in Methoden wird berechnet durch das Zählen von Codezeilen ohne Leer- oder Kommentarzeilen in Methoden. Dies ergibt eine einfache Aussage über die Komplexität von Methoden und der Aufarbeitung der Klassen. Martin (2008, S. 34) schreibt über Methoden: „The first rule of functions is that they should be small. The second rule of functions is that they should be smaller than that. Functions should not be 100 lines long. Functions should hardly ever be 20 lines long.“ Dies ist eine subjektive Betrachtung des Problems, jedoch lässt sich ein Rückschluss daraus ziehen, dass in umfangreicheren Methoden und Klassen statistisch mehr Fehler auftreten (Kan, 2002).

5.2.4 Halstead Volume

Ein Vertreter der Komplexitätsmetriken ist Halsteads „Software Sience“, welche Sneed et al. (2010) und Al Qutaish und Abran (2005) genauer beschreiben. Halstead teilt in seinen Metriken Software in einen Wortschatz ein, wobei dieser aus Operatoren und Operanden besteht. Basierend auf dem Zählen dieser Token werden folgende Variablen aus dem Quellcode gelesen:

$$n_1 = Anzahl\ eindeutiger\ Operatoren$$
$$n_2 = Anzahl\ eindeutiger\ Operanden$$
$$N_1 = Anzahl\ aller\ Operatoren$$
$$N_2 = Anzahl\ aller\ Operanden$$

Zusätzlich definiert Halstead auch eine Summe aus minimal notwendigen Operanden beziehungsweise Operatoren:

$$n_1^* = Anzahl\ potentieller\ Operatoren$$
$$n_2^* = Anzahl\ potentieller\ Operanden$$

Aus diesen Ausgangsvariablen lassen sich die Metrikwerte nach Halstead ableiten:

1. „die Länge N eines Programmes,

$$N = N_1 + N_2$$

2. das Vokabular n,

$$n = n_1 + n_2$$

3. das Volumen V eines Programmes.“ (Al Qutaish & Abran, 2005, S. 3)

$$V = N * \log_2 n$$

Nach Halstead lassen sich aus den Grundvariablen noch weitere Metriken, wie zum Beispiel die Schwierigkeit eines Programmes, den Aufwand, um einen Algorithmus in ein Programm zu überführen oder die Zeit, diesen Aufwand umzusetzen, ableiten. Es ergibt sich jedoch nach Al Qutaish und Abran das Problem, wie genau zwischen Operator und Operand zu unterscheiden ist. Sie konnten weitere Metriken von Halstead aufgrund dieses Umstandes nicht empirisch nachweisen. Verifiziert wurden die Modelle der Codegröße und Codekomplexität (Sneed, Seidl, & Baumgartner, 2010).

5.2.5 Depth of Inheritance

Die Codemetrik der Depth of Inheritance von Chidamber und Kemerer (1994), sie wird auch als Depth of Inheritance Tree bezeichnet (Vererbungstiefe), beschreibt die Anzahl der Klassen von der aktuellen Klasse bis zur Wurzel des Programms. Wie in Abbildung 5-2 veranschaulicht, erhält die Klasse E einen Wert für die Vererbungstiefe von 3 und die Klasse C einen Wert von 2. Die Autoren stützen ihre Metrik auf drei Sichten auf das Problem der Komplexität:

1. „Je tiefer eine Klasse in der Hierarchie angesiedelt ist, umso höher ist die Wahrscheinlichkeit, Methoden zu erben, und desto komplexer sind Vorhersagen zu dessen Verhalten.
2. Tiefere Bäume resultieren in erhöhter Designkomplexität, da mehr Methoden und Klassen involviert sind.
3. Je tiefer eine Klasse in der Hierarchie angesiedelt ist, desto häufiger werden vererbte Methoden potentiell wiederverwendet. (Chidamber & Kemerer, 1994, S. 483)“

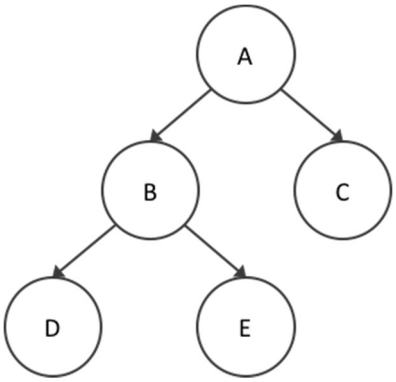


Abbildung 5-2: Beispiel für Depth of Inheritance

5.2.6 Coupling between object classes

Die Klassenkopplung nach Chidamber und Kemerer (1994) zählt die Abhängigkeiten von Klassen untereinander, sowohl auf gleicher Ebene als auch in der Vererbungshierarchie. Sie definieren die Kopplung wie folgt: „Zwei Klassen sind gekoppelt, wenn Methoden aus einer Klasse auf Methoden oder Instanzvariablen einer anderen Klasse zugreifen“ (Chidamber & Kemerer, 1994, S. 486) und verbinden damit folgende Hypothesen:

1. Hohe Kopplung verhindert ein modulares Design und erschwert somit die Wiederverwendbarkeit.
2. Je geringer die Kopplung, desto einfacher sind Designänderungen und damit die Wartbarkeit.
3. Je höher die Kopplung, desto höher der Testaufwand.

5.2.7 Lack of Cohesion in Methods

Die Softwaremetrik Lack of Cohesion in Methods (LOCM) ist ein Maß für die Kohäsion einer Klasse. Chidamber und Kemerer (1994) definieren darin die Gleichheit von Methoden in Klassen. Je höher das Maß, desto ähnlicher sind sich die Methoden, das bedeutet, dass Methoden auf dieselben Klasseneigenschaften zugreifen. Das Ziel ist es, ein möglichst hohes Maß an Kohäsion herzustellen. Dies beruht auf den Annahmen, dass

1. der Zusammenhang von Methoden hoch sein soll, wenn die Kapselung einer Klasse erreicht werden soll,
2. und bei niedriger Kohäsion ein Designfehler vorliegt, da die Klasse sich um mehrere Dinge kümmert, wodurch die Fehlerhäufigkeit und Komplexität steigt.

Daraus ergibt sich eine Skala, in welcher 0 beziehungsweise ein niedriger Wert der Metrik für gutes Architekturdesign steht und ein hoher Wert für schlechtes Design, welches nicht Ziel der Softwarearchitektur sein soll.

Berechnet wird die Metrik LOCM durch die Anzahl der Methodenpaare, deren Ähnlichkeit in Bezug auf Zugriffe auf Klasseneigenschaften gleich 0 ist, minus der Anzahl der Methodenpaare, deren Ähnlichkeit ungleich null ist.

$$X = \sum unähnliche\ Methodenpaare - \sum ähnliche\ Methodenpaare$$

Dies ist in Listing 5-1 als Beispiel dargestellt. Die Summe der Methodenpaare, deren Ähnlichkeit im Zugriff auf Klasseneigenschaften gleich null ist, ist 2 ({fnA, fnC}, {fnB, fnC}). Dem gegenüber steht die Summe der Methodenpaare, deren Ähnlichkeit ungleich null ist, mit dem Wert 1 ({fnA, fnB}). Daraus ergibt sich für das gegenwärtige Beispiel ein Metrikwert von 1.

$$X = (\{fnA, fnC\}, \{fnB, fnC\}) - (\{fnA, fnB\}) = 1$$

```
public class A
{
  private int _memberA;
  private int _memberB;
  private int _memberX;

  public void fnA()
  {
    _memberA = 1;
    _memberB = 2;
  }

  public void fnB()
  {
    _memberB = 3;
  }

  public void fnC()
  {
    _memberX = 9;
  }
}
```

Listing 5-1: Quellcodebeispiel zur Berechnung der Metrik LOCM

5.2.8 Zyklomatische Komplexität

Die zyklomatische Komplexität von McCabe (1976) ist ein Versuch, die Komplexität von Software in die Graphentheorie zu überführen und anhand dieser abzubilden. McCabes Forschungsansatz war die Einordnung von Software in eine Skala mit dem Ziel, deren Wartbarkeit und Testbarkeit vorherzusagen. Die zyklomatische Komplexitätszahl V(G) berechnet sich aus einem Graphen G mit n Vertizen, e Kanten und p zusammenhängenden Komponenten durch die Formel

$$V(G) = e - n + 2p.$$

Im Listing 5-2 ist ein Beispiel der Berechnung der zyklomatischen Komplexität an einem Quellcode veranschaulicht.

```
public void Benotung(int prozent)               //A
{
  if (prozent < 50)                             //B
  {
    Console.WriteLine("Nicht bestanden");       //C
  }
  else
  {
    Console.WriteLine("Bestanden");             //D
  }

  if (prozent >= 90)                            //E
  {
    Console.WriteLine("mit Auszeichnung");      //F
  }
}                                               //G
```

Listing 5-2: Quellcodebeispiel für McCabes zyklomatische Komplexität

Aus dem gezeigten Quellcode lässt sich ein Kontrollflussgraph, wie in Abbildung 5-3 dargestellt, ableiten. Der Graph beginnt mit dem Knoten A, der Eingabe in die Funktion. Im Knoten B wird eine IF-ELSE-Anweisung ausgewertet, welche zu den Knoten C oder D überleitet. Zum Abschluss folgt eine einfache IF-Anweisung, welche im Positiv-Fall zur Anweisung F überleitet und die Funktion abschließt. Daraus ergeben sich nach der Berechnung von McCabe mit sieben Vertizen, acht Kanten und einer Komponente eine Komplexitätszahl von 3.

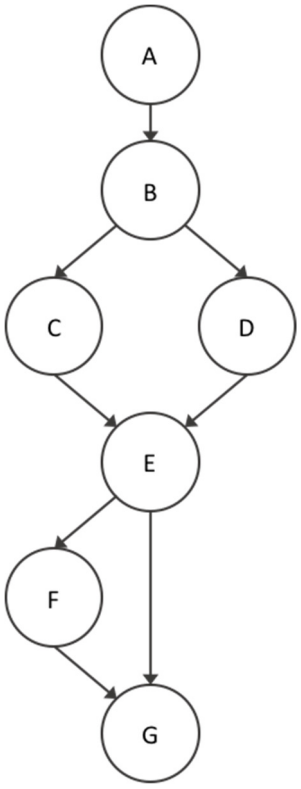


Abbildung 5-3: Kontrollflussgraph für McCabes zyklomatischer Komplexität

5.2.9 Wartbarkeitsindex

Der Wartbarkeitsindex ist eine Metrik von Oman und Hagemeister (1992), welche den Versuch unternahmen, mehrere bestehende Metriken zu einer zusammenzuführen. Sie analysierten die Literatur und leiteten daraus ab, dass die Wartbarkeit von Software im Grunde auf drei Säulen steht. Auf dem Management, bestehend aus dem zur Verfügung stehenden Personal und den eingesetzten Prozessen, der Einsatzumgebung, bestehend aus Ist- und Sollzustand, sowie dem Zielsystem der Anwendung, bestehend aus dem Reifegrad der Software, dem Quellcode selbst und der verfügbaren Dokumentation. Aus jedem dieser Unterbereiche können Metriken ausgewählt und über eine Formel zu einem Wartbarkeitsindex zusammengefasst werden.

Microsoft (2007) zieht für seine Berechnungen in der Entwicklungsumgebung Visual Studio ab der Version 2008 folgende Formel für die Berechnung heran, wobei MI für Maintainability Index, V für Halstead's Volume, Z für die zyklomatische Komplexität und L für die Anzahl der Codezeilen steht:

$$MI = \frac{\left(171 - 5{,}2 * \ln(V) - 0{,}23 * (Z) - 16{,}2 * \ln(L)\right) * 100}{171}$$

Dies ist eine verfeinerte Version der Metrik von Oman und Hagemeister, um diese auf einen Prozentbereich abbilden zu können. Der Index wird interpretiert als eine Skala von 0 bis 100 Prozent mit den Bereichen 0 bis 9 Prozent für schlechte Wartbarkeit, 10 bis 19 Prozent für moderate Wartbarkeit und 20 bis 100 Prozent für einen Quellcode, dessen Wartbarkeit nach Ansicht von Microsoft für in Ordnung befunden wird.

5.3 Auswahl der Untersuchungsmerkmale

Welche Metriken geeignet sind, um die Forschungsfrage zu beantworten, soll die Auswahl mit Hilfe des Goal-Question-Metric-Modells (GQM-Modell) von Basili, Caldiera und Rombach (1994) ermöglichen. Dazu wird in diesem Kapitel das GQM-Modell vorgestellt und die Operationalisierung der Forschungsfrage mit den aufgestellten Hypothesen vorgenommen.

5.3.1 Goal-Question-Metric-Modell

Das Goal-Question-Metric-Modell ist ein Ansatz, in einem Projekt oder auch einem gesamten Unternehmen effektiv zu messen. Um feststellen zu können, was gemessen werden soll, bietet das Modell ein Top-Down-Verfahren, um, abgeleitet aus einem Ziel, die richtigen Fragen zu stellen und diese anhand von Messungen adäquat beantworten zu können. Das Modell dient auch dazu, in Zukunft ein gezieltes Set an Messungen für die Evaluierung des Status quo bereit zu haben. Nach Basili et al. (1994, S. 528) „benötigt die Softwareentwicklung, wie auch jede andere technische Disziplin, einen Mechanismus von Messungen für Feedback und Evaluierung". Damit dieser Mechanismus effektiv ist, muss dieser auf ein spezifisches Ziel fokussiert sein, auf alle Objekte angewendet werden

(Produkt, Prozess und Ressource) und auf den organisatorischen Kontext zugeschnitten sein.

Das Goal-Question-Metric-Modell beinhaltet, wie der Name verrät, 3 Ebenen, wie in Abbildung 5-4 dargestellt. Die erste Ebene ist die konzeptionelle Ebene („Goal“), welche das Untersuchungsobjekt aus den Kategorien Produkt, Prozess oder Ressource auswählt. In der zweiten Ebene, der operationalen Ebene „Question“, wird anhand von Fragen festgestellt, wie das gewählte Ziel erreicht werden kann. Sie sollen das Objekt der Ebene eins aus qualitativer Sicht charakterisieren. Auf der quantitativen Ebene der „Metrics“ werden jeder Frage aus Ebene zwei quantitative Messungen zugrunde gelegt, um diese in Zahlen beantworten zu können. (Basili et al., 1994)

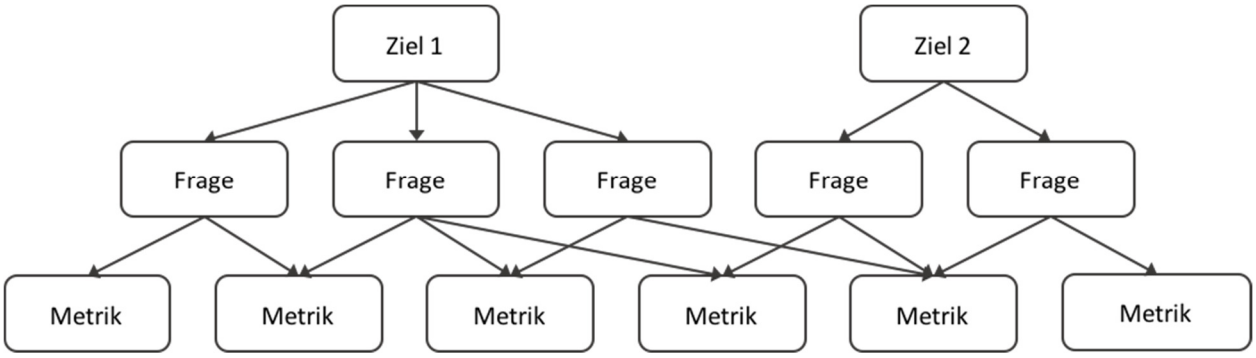


Abbildung 5-4: Aufbau des GQM-Modells in Anlehnung an Basili et al. (1994, S. 529)

5.3.2 Operationalisierung

In der gegenständlichen Arbeit wird zur Forschungsfrage erhoben, ob und wie sich Microservices gegenüber monolithischen Anwendungen im Detail der Codequalität verändern. Dazu wurde festgestellt, dass sich die Codequalität als Produktmetrik definiert, welche über Maße wie Größe, Komplexität oder Wartbarkeit gemessen und verglichen werden kann. (Kan, 2002)

In Anlehnung an die GQM-Methode wird die Beantwortung der Forschungsfrage als Ziel ausgegeben und die Fragen als Hypothesen formuliert. Dadurch kann eine Hypothese aus der Literatur von Rau (2016) aufgestellt werden, wonach die Anwendung des Architekturmusters von Microservices die Adaptierung einer Software an geänderte Anforderungen durch EntwicklerInnen beschleunigt werden kann. Dieser Umstand kann durch die Metrik der Änderungsgeschwindigkeit gemessen werden, welche im Vergleich zwischen monolithischer Anwendung und derselben Anwendung in Form von Microservices erhoben werden kann. Dadurch ergibt sich als erstes Untersuchungsmerkmal die Softwarequalitätsmetrik der Änderungsgeschwindigkeit, welche Rückschlüsse auf die Wartbarkeit einer Software zulässt. Die Erhebung wird durchgeführt an erfahrenen EntwicklerInnen im Umgang mit der Programmiersprache des zu untersuchenden Systems.

Als zweite Hypothese zur Forschungsfrage wurde ebenfalls durch Rau (2016) festgestellt, dass Microservices eine geringere Komplexität als dieselbe Anwendung in monolithischer Form aufweisen. Dies kann erhoben und gemessen werden durch die Softwaremetrik der

Komplexität, sowie aus verschiedenen Größenmetriken und der Metrik der Kohäsion. Im Detail werden die Metriken Lines of Code, Methode Lines of Code, Depth of Inheritance, Coupling between object classes, die zyklomatische Komplexität und der Wartbarkeitsindex nach Microsoft herangezogen.

Zu guter Letzt wird in dieser Arbeit zur Bewertung der Codequalität auch eine qualitative Komponente erhoben. Diese ist definiert durch Interviews von SoftwareentwicklerInnen, welche das subjektive Empfinden bei der Analyse und Veränderung der Software einfangen soll. Hierzu wurde die Hypothese durch den Autor, nach Erkenntnisstand des theoretischen Teils der Arbeit, aufgestellt, dass Microservices durch ihre Vorgabe zur Einfachheit und möglichst kleinen Ausführung zur Erhöhung der Lesbarkeit in einem Quellcode führen.

6 LABORVERSUCH

Der praktische Teil dieser Arbeit gliedert sich in mehrere Methoden zur Untersuchung der Codequalität von Microservices. Wie in den vorhergehenden Kapiteln dargelegt, eignen sich verschiedene Metriken für die statische und quantitative Analyse von Quellcodes, als auch Metriken für die qualitative Analyse, wie zum Beispiel die Lesbarkeit von Software für neue Anforderungen. Dieses Kapitel widmet sich der Erklärung des Versuchsaufbaus. Dieser setzt sich zusammen aus einer Ausgangssoftware und der Entwicklung einer abgewandelten Software auf Basis der Erkenntnisse aus der Theorie zu Microservices, der Analyse dieser beiden Softwareprodukte anhand der dargestellten Metriken (siehe Kapitel 5) und einer abschließenden Befragung von EntwicklerInnen zu dem Thema Microservices und deren Lesbarkeit des Quellcodes. Das Vorgehen ist deduktiv und überprüft die Hypothesen.

6.1 Erklärung

Das Ausgangsprodukt der Untersuchung ist ein Softwareprodukt von der Firma INTECO atec automation GmbH mit dem Namen AVS - Alarmverwaltungssystem. Die Firma ist angesiedelt im Bereich der Automatisierungstechnik und bietet spezielle Lösungen für industrielle Produktions- und Fertigungsprozesse (INTECO atec automation GmbH, 2016). In diesem Zusammenhang wurde auch die Software AVS entwickelt, welche es ermöglicht, auftretende Alarme zu sammeln, zu archivieren, auszuwerten und bei Bedarf direkt über verschiedene Kanäle an betroffene MitarbeiterInnen des Betriebes weiterzuleiten. Alarme können in Industrieanlagen, während des Bearbeitungsprozesses oder des Stillstandes auftreten, und signalisieren Störungen oder fehlerhafte Anlagenzustände, welche eine Interaktion mit einem/einer menschlichen BedienerIn erfordern. Durch den hohen Grad an Automatisierung in modernen Fertigungsbetrieben ist es möglich, dass sich nur wenig Personal um die Funktionstüchtigkeit von mehreren Anlagen kümmert und ist daher auf Systeme wie das Softwareprodukt der Firma INTECO atec automation GmbH angewiesen, um die Qualität und Leistung des Betriebes aufrecht erhalten zu können.

6.2 Versuchsaufbau

Das Kapitel des Versuchsaufbaus widmet sich der Darstellung der Ausgangssoftware als monolithische Software mit dessen innerer Architektur, sowie der Anwendung der Techniken aus Kapitel 3.1 zur Trennung der monolithischen Anwendung in verschiedene Teile und den Techniken aus Kapitel 3.2 zur Zusammenführung der entstandenen Teilservices zu einer Gesamtanwendung.

6.2.1 Analyse der Ausgangssoftware

Die Software AVS ist als Client-Server-Anwendung ausgeführt und bietet in der aktuellen Ausbaustufe die Anbindung an zwei verschiedene Quellsysteme für Alarme an. Zur Versendung von Alarmen an die Bediener der Anlage sind ebenfalls zwei Möglichkeiten vorgesehen. Zum einen die Alarmierung per Short Message Service (SMS) oder zum anderen durch den verstärkten Einsatz von Smartphones, als E-Mail. Das Hauptartefakt der gesamten Anwendung ist ein Alarm. Die Alarme in der Anwendung sind definiert durch

- einen Identifier, einem eindeutigen Merkmal des Quellsystems,
- einen Kommentar,
- einen Wert,
- ein entstammendes Quellsystem,
- einen Zeitpunkt des Auftretens,
- einen Zeitpunkt, zu dem der Alarm erlischt,
- einen Zeitpunkt, in dem ein/eine BenutzerIn dieses quittiert,
- den/die BenutzerIn, der/die den Alarm quittiert
- und einer Priorität.

Zur Analyse der Softwarearchitektur wird das Sichtenmodell aus dem Template arc42 (Starke & Hruschka, 2016) angewendet und die Ergebnisse der Analyse in diesem Kapitel dargestellt.

6.2.1.1. Kontextsicht

Die kontextuelle Abgrenzung der Software basiert auf dem fachlichen und technischen Kontext. Aus fachlicher Sicht werden Daten aus zwei verschiedenen Quellsystemen abgerufen. Diese sind eine Visualisierungsapplikation des Herstellers Wonderware mit dem Namen InTouch und eine generische Schnittstelle nach dem Ole for Process Control (OPC) Standard. Aus dem Visualisierungssystem InTouch werden Alarme gelesen und in das System importiert. Hierbei verwaltet das Visualisierungssystem diese Alarme selbstständig und es kann von versierten AnwenderInnen parametriert werden, um neue Alarme hinzuzufügen, deren Einstellungen zu verändern oder Alarme zu entfernen. Für die OPC-Schnittstelle wird diese Parametrierung, welche Werte Alarme auslösen und wie deren Namen, Kommentare und Fehlermeldungen sind, in der Anwendung AVS vorgenommen. Somit müssen beide Systeme unterschiedlich betrachtet werden. Zum Export der Alarme verwendet das System zwei externe Systeme. Zum einen ein Modem, über welches SMS Nachrichten versendet werden, und zum anderen einen E-Mail-Server, über welchen E-Mail Nachrichten versendet werden. Eine grafische Darstellung dieser Systeme und der Schnittstellen bietet die Abbildung 6-1.

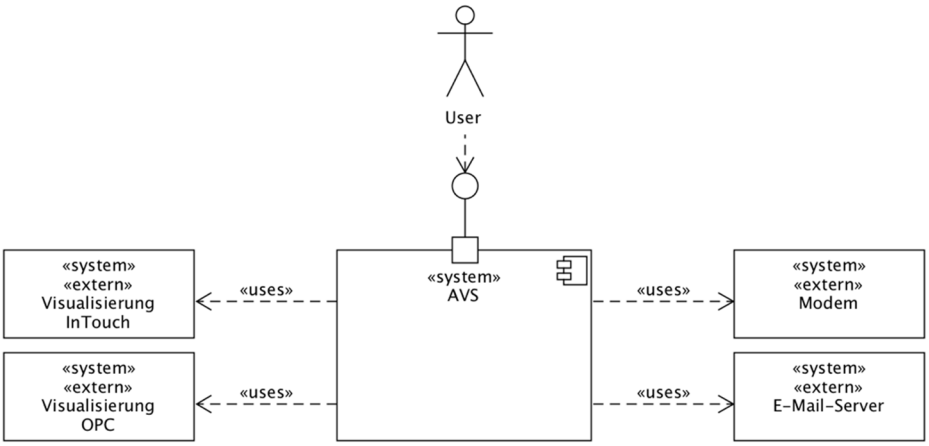


Abbildung 6-1: Kontextsicht des Ausgangssystems

Aus technischer Sicht verwendet jedes angebundene System eine eigene Schnittstelle. Zu den Quellsystemen gibt es eine Schnittstelle auf Basis einer relationalen Datenbank, in diesem Fall einen SQL-Server von Microsoft, und die standardisierte Schnittstelle OPC. Mit den Versendesystemen wird über eine serielle Schnittstelle oder per Simple Mail Transfer Protocol (SMTP) kommuniziert. Diese Schnittstellen und deren Datenformate sind in der Tabelle 6-1 ausführlich aufgelistet.

Nachbarsystem	Schnittstelle	Technisches Protokoll
Visualisierung InTouch	Alarme lesen	ADO-Schnittstellte .NET
Visualisierung OPC	Rohdaten lesen	OPC Classic DA, DCOM
Modem	SMS versenden	AT-Commands
E-Mail-Server	E-Mail versenden	SMTP

Tabelle 6-1: Kontextsicht des Ausgangssystems

6.2.1.2. Bausteinsicht

Das System im monolithischen Design folgt grundsätzlich der Basisarchitektur nach dem Schichtenmuster (siehe Kapitel 2.1.2). Dem/der BenutzerIn präsentiert sich das System durch die Benutzeroberfläche mit der Hauptansicht einer Liste von aktuellen Alarmen. Diese Clientanwendung kommuniziert mit der Datenbank, in welcher die gesammelten Alarme abgelegt sind. Die Hauptkomponente im System nimmt der AVS-Dienst ein. Dieser wird als Windows-Dienst ausgeführt und sammelt über die entsprechenden Module die Alarme von den Quellsystemen ein, legt diese Alarme ab, filtert sie, und wenn ein Filter es verlangt, werden die Alarme über die Module versendet. Der Aufbau des Systems auf Ebene 1 der Bausteinsicht ist in Abbildung 6-2 dargestellt.

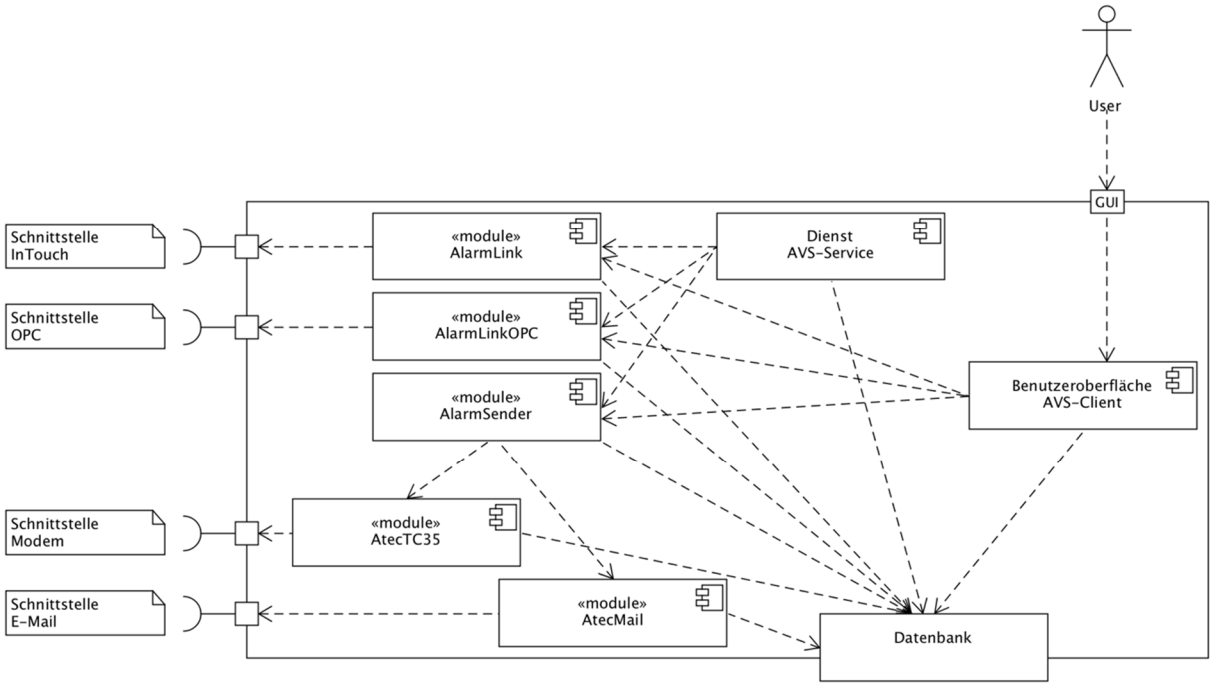


Abbildung 6-2: Bausteinsicht des Ausgangssystems

In der Hauptansicht der Benutzeroberfläche können von dem/der BenutzerIn alle gesammelten Alarme aus allen angebundenen Quellsystemen an einem Ort eingesehen werden. Hinzu kommen detailliertere Ansichten zur chronologischen Reihenfolge der Alarme, einer Ansicht zur Häufigkeitsverteilung und durchschnittlichen Alarmdauer sowie einer Ansicht über alle versendeten Alarme. Diese Listen können zur Anzeige nach Zeitraum, den Alarmtexten, der Quelle oder der Priorität gefiltert werden. Die Daten werden aus einer relationalen Datenbank (in diesem Fall SQL-Server) gelesen. Die Konfiguration der Module findet in den Modulen selbst statt, dadurch gibt es eine direkte Schnittstelle von der Komponenten-benutzeroberfläche zu den einzelnen Modulen und über diese werden die Konfigurationen in der Datenbank gespeichert. Eine Ausnahme bildet das Sendemodul „AlarmSender“, welches diese Aufgabe für die Submodule zum eigentlichen Versenden der Nachrichten übernimmt.

6.2.1.3. Laufzeitsicht

Die Laufzeitsicht stellt den Informationsfluss über mehrere Module dar. Der primäre Use-Case der Software ist das Sammeln der Daten von den Quellsystemen, der Verarbeitung über den Filter und zum Abschluss das Versenden der Daten. Dieser Vorgang wird durch eine Timer-Komponente, einen Trigger, gestartet und läuft ohne Interaktion eines Benutzers beziehungsweise einer Benutzerin ab (Abbildung 6-3).

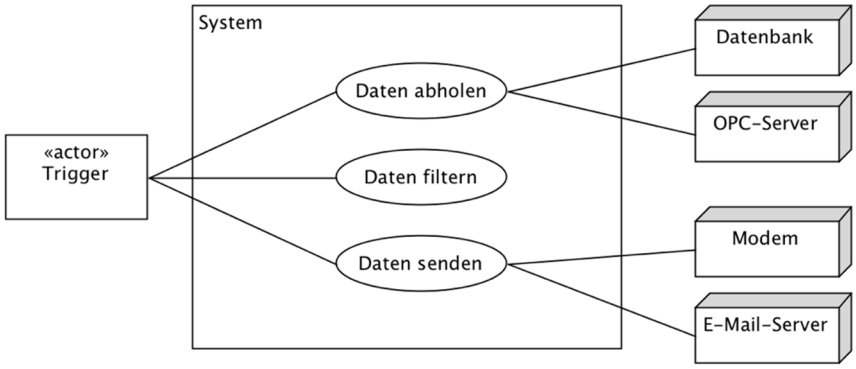


Abbildung 6-3: Use-Case Alarm lesen und versenden

Wie in der Laufzeitsicht in Abbildung 6-4 dargestellt, initiiert der AVS-Dienst für jedes konfigurierte Quellsystem die Datenverarbeitung. Das bedeutet für das Modul zum Visualisierungssystem InTouch das Lesen der Daten aus der bereitgestellten Quelldatenbank und das Speichern dieser Alarme in der Datenbank der Anwendung. Für das Modul der OPC-Kopplung bedeutet dies das Lesen der konfigurierten Datenpunkte von einem OPC-Server, das Verifizieren, ob eine Zustandsveränderung vorliegt, und das Ablegen der Daten in der Datenbank der Anwendung. Nach dem Abschluss des Vorganges „Daten abholen“ werden diese verarbeitet und einem Filter unterzogen. Dieser setzt sich zusammen zum einen aus der Auswahl eines Empfängers beziehungsweise einer Empfängerin, welcher/welche in einer Empfängergruppe mit entsprechendem Filterkriterium für Alarm vorhanden sein muss, sowie einer zeitlichen Einschränkung, ob der/die EmpfängerIn in der richtigen Arbeitsschicht vorhanden ist und für die Alarmierung per Nachricht in Frage kommt. Wenn für diesen/diese EmpfängerIn eine gültige Konfiguration zum Empfang von Nachrichten vorliegt, in Form einer gültigen Telefonnummer oder E-Mail-Adresse, wird der Alarm in die Sendeliste der Anwendung eingetragen. Dadurch kann es vorkommen, dass ein Ausgangsalarm multiple Nachrichten an verschiedene Empfänger auslöst. Zum Abschluss des Durchlaufes wird der Vorgang „Daten senden“ in der Komponente des „AlarmSenders“ bearbeitet. Dieser liest die zu sendenden Daten aus der Sendeliste und versucht über die bereitgestellten Module die Nachrichten zuzustellen. Gelingt dies im ersten Anlauf nicht, werden die Daten nicht als gesendet markiert und im nächsten Durchlauf wird erneut versucht, die Nachrichten zuzustellen.

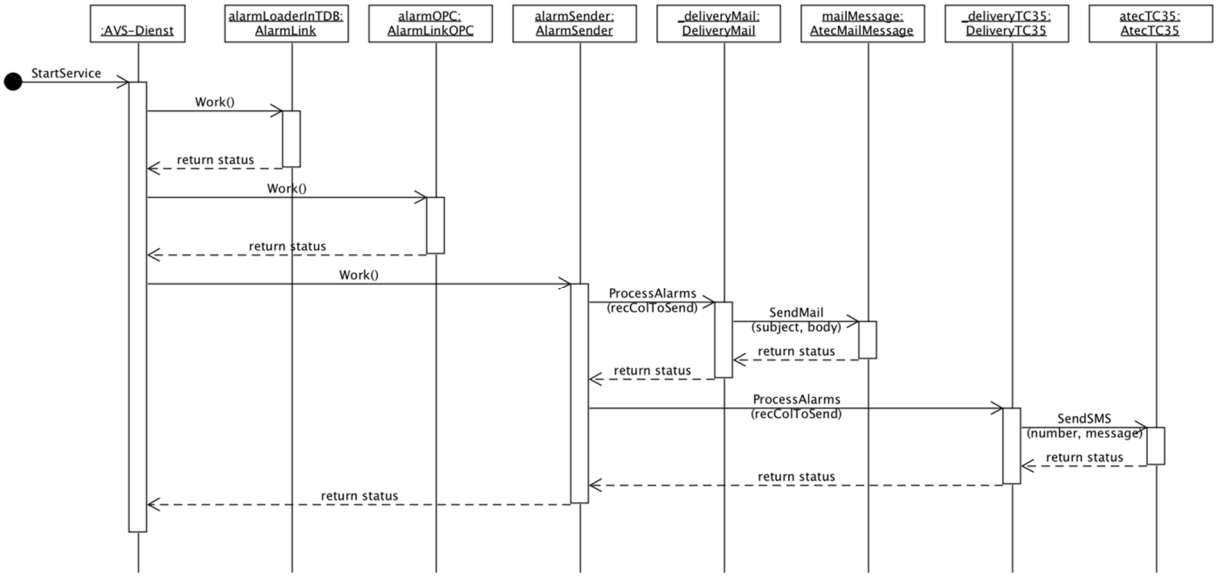


Abbildung 6-4: Sequenzdiagramm des Ausgangssystems

6.2.1.4. Verteilungssicht

Durch den Aufbau der Anwendung im Client-Server-Prinzip kann diese Anwendung auf zwei Computersystemen betrieben werden, der Betrieb auf einer Singlestation ist jedoch auch möglich. Die Verteilungssicht veranschaulicht, welche Komponente des Systems auf welcher Hardware läuft und wie die Kommunikation der Anwendungsteile untereinander funktioniert. Diese Darstellung dient dem Entwickler sowie auch dem Endkunden beziehungsweise der Endkundin des Produktes, um die ideale Umgebung für den Softwareeinsatz zu visualisieren (Gharbi, Koschel, Rausch, & Starke, 2015).

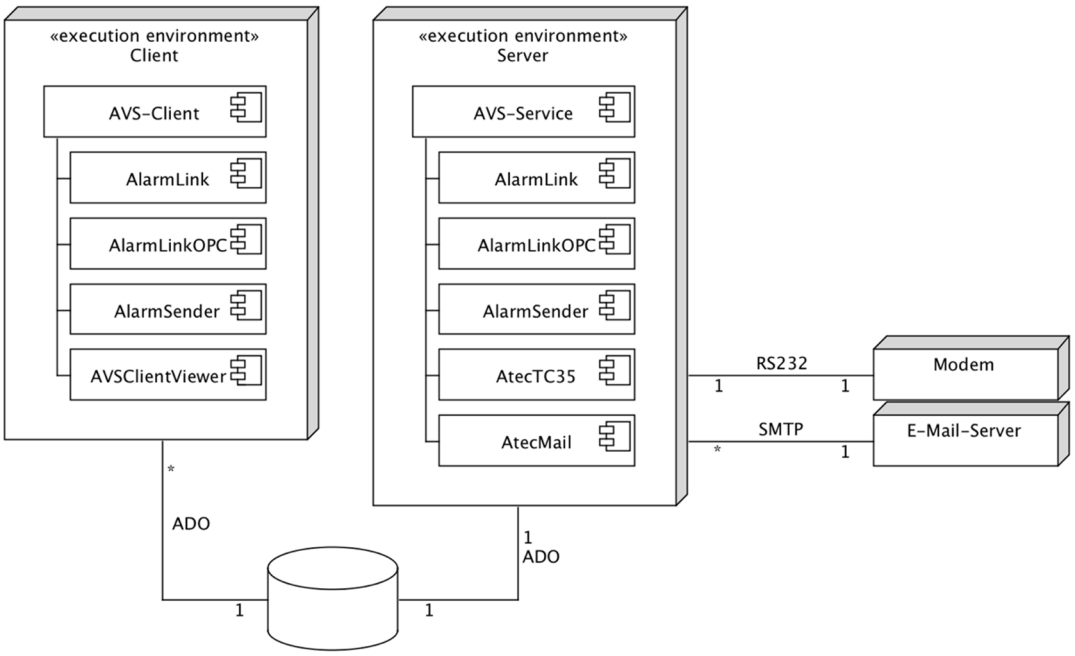


Abbildung 6-5: Verteilungssicht des Ausgangssystems

Wie in Abbildung 6-5 ersichtlich, werden jene Komponenten, welche auf spezielle Hardware zugreifen und vom Dienst angesprochen werden, nur am Server bereitgestellt. Dieser benötigt eine serielle Verbindung zu einem kompatiblen Modem. Außerdem muss die Netzwerkverbindung zu einem E-Mail-Server konfiguriert sein, damit der E-Mail-Versand der Nachrichten vom Server an die Empfänger durchgeführt werden kann. Das Softwaredesign setzt voraus, dass diese Komponenten am Server vorhanden sind und im gleichen Ausführungskontext wie der Dienst angesprochen werden können. Das bedeutet, dass alle Teilkomponenten und Module des Dienstes und die Teilkomponenten der Clientanwendung auf dem jeweiligen Rechner vorhanden sein müssen, da keine Intercomputerkommunikation im System stattfindet.

6.2.2 Veränderung der Software

Um die Forschungsfrage nach der Veränderung von Codequalität mit Microservices beurteilen zu können, wird das beschriebene System aus Kapitel 6.2.1 mit den Techniken aus Kapitel 3 in Microservices getrennt und die Kommunikation dieser Komponenten mit dem verbleibenden Monolithen wiederhergestellt. Zur Trennung wird die Methode der Extrahierung von Funktionen aus Kapitel 3.1.3 gewählt.

6.2.2.1. Identifizieren von Funktionen

Zur Identifikation von Funktionen zur Extraktion kann das Domain Driven Design (Evans, 2004) aus Kapitel 2.3.1 herangezogen werden. Wenn die Ausgangsanwendung betrachtet wird, können drei primäre Funktionen abgeleitet werden. Funktion eins stellt die Hauptfunktion dar, die Benutzerschnittstelle, die Verarbeitung von Alarmen und das Generieren von Nachrichten. Als abgekapselte Funktionen zwei und drei können die Quellsysteme sowie die Versendesysteme betrachtet werden. Zusätzlich soll für zukünftige Erweiterungen die Möglichkeit bestehen, das System einfach um neue Quell- oder Zielsysteme zu erweitern. Die Schnittstelle zwischen diesen Systemen sind definierte Objekte, welche über einen Kanal ausgetauscht werden. In der Ausgangsanwendung sind diese Schnittstellen Modulgrenzen, welche mittels einfacher Funktionsaufrufe überwunden werden. In einem Microservice-System müssen diese Übergänge über Servicegrenzen hinweg mittels Transportmechanismen, wie in Kapitel 3.2 beschrieben, bewältigt werden. Wenn diese Hauptfunktionen als eigenständige Kontexte nach dem Domain Driven Design angesehen werden, stellt sich ein Übersichtsbild wie in Abbildung 6-6 dar. Es beschreibt auf einfache Weise, welche Funktionen und Komponenten zusammengehören und von einem Team entwickelt werden können, wo Schnittstellen zwischen diesen Funktionen liegen und welche Daten über diese Schnittstellen ausgetauscht werden.

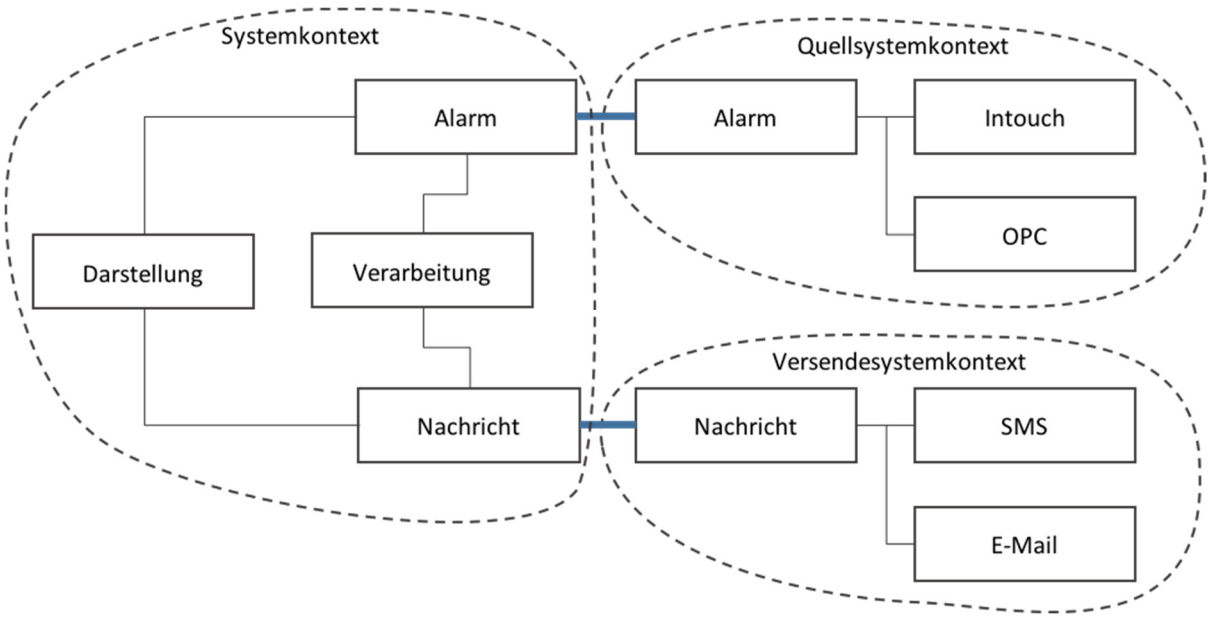


Abbildung 6-6: Beschränkte Kontexte in der Anwendung AVS

6.2.2.2. Extrahieren von Funktionen

Für den Versuch wird aus den getrennten Kontexten der Quell- und Versendesysteme jeweils eine Funktion als Microservice implementiert. Im konkreten Fall wird aus dem Modul „AlarmLink“ die Zugriffslogik auf das Fremdsystem InTouch in ein Microservice ausgegliedert. Dieses Microservice behandelt den Zugriff auf die Datenbank des Fremdsystems, bietet gleichzeitig eine REST-Schnittstelle für das Gesamtsystem AVS an und wird als Windows-Dienst ausgeführt. Als zweiter Dienst wird das Modul zum Versenden von SMS-Nachrichten ausgewählt. Das Modul „AtecTC35“ wird als Gesamtes in ein Microservice überführt und die Zugriffslogik aus dem Modul „AlarmSender“ angepasst. Das Service behandelt die Kommunikation mit dem verwendeten Modem über eine serielle Schnittstelle, bietet eine REST-Schnittstelle an und wird ebenfalls als Windows-Dienst konzipiert.

Zur Ausführung eines C# Projektes als Windows-Dienst wird die Funktionsbibliothek Topshelf eingesetzt. Diese bietet die Möglichkeit, Windows-Dienste in der Entwicklungsumgebung im Debug-Modus auszuführen und dennoch im Release-Modus als Windows-Dienste zu registrieren. Dafür stellt die Bibliothek einen Mechanismus zur Verfügung, um über Commandlineparameter die erstellte Anwendung zur Laufzeit als Dienst zu installieren. Zur Zeit der Implementierung der Microservices wird die Bibliothek in der Version 4.0.3 eingesetzt. (Patterson, Smith, & Sellers, 2016)

6.2.2.3. Zusammenführen des Microservice 1 mit dem Monolithen

Für das Microservice 1, dem Lesen der Alarme vom Quellsystem InTouch, wurde als Kommunikationstechnologie aus den vorgestellten Technologien von Kapitel 3.2 REST gewählt. Die Anwendung dieser Technologie in einem eigenen Projekt wird von Microsoft durch Bibliotheken für die Serviceseite sowie die Clientseite unterstützt. Unter dem

Schlagwort WebAPI bietet Microsoft die Möglichkeit für REST-Zugriffe, über ihre Implementierung des Standards OWIN, für Webseiten und Applikationen im .net Framework an. Diese Bibliotheken werden über die Paket- und Bibliothekverwaltung NuGet in das Projekt eingebunden:

- Microsoft.AspNet.WebApi.OwinSelfHost (Version 5.2.3)
- Microsoft.AspNet.WebApi.Client (Version 5.2.3)

Wie aus den Kontexten des Domain Driven Design in Abbildung 6-6 ersichtlich, wird in diesem Szenario der Alarm als Entität im Modell von dem Service an die aufrufende Anwendung übertragen. Die Abrufmöglichkeiten dieser Entität sind in der Tabelle 6-2 dargestellt und sie wird je nach Anforderung des Clients als Extensible Markup Language (XML) oder JavaScript Object Notation (JSON) encodiert übertragen. Aus dem Listing 6-1 ist die Darstellung der Entitäten Alarm und Status im Quellcode ersichtlich. Zusätzlich zu den Alarmen wird über dieselbe Schnittstelle der Status des Microservice angeboten. Über die Entität „Status“ kann der Zustand der Verbindung des Microservice zum Fremdsystem abgefragt und der Zeitpunkt des letzten aufgetretenen Alarms ermittelt werden.

```
public class Alarm
{
  public DateTime EventStamp { get; set; }
  public string State { get; set; }
  public string TagName { get; set; }
  public string Description { get; set; }
  public string Area { get; set; }
  public string Type { get; set; }
  public string Value { get; set; }
  public string CheckValue { get; set; }
  public short Priority { get; set; }
  public string Category { get; set; }
  public string Provider { get; set; }
  public string Operator { get; set; }

  public override string ToString()
  {
    return EventStamp.ToString("s") + ": " + TagName.PadRight(10) +
           State.PadRight(10);
  }
}

public class Status
{
  public bool IsConnected { get; set; }
  public DateTime LastEvent { get; set; }
}
```

Listing 6-1: Entitäten aus dem Microservice 1

Methode	URI	Parameter	Übertragene Daten	Statuscode
GET	intouchdb	-	Status	200 – OK
GET	intouchdb/{since}	Datetime	List<Alarm>	200 – OK 400 – Bad Request

Tabelle 6-2: REST-Methoden des Microservice 1

Auf der Clientseite, dem ursprünglichen Modul zum Lesen der Alarme aus dem Quellsystem, wurden die Aufrufe des SQL-Adapters durch Aufrufe des WebAPI Clients mit den entsprechenden Parametern ersetzt. Der Statuscode 400 wird vom Service ausgelöst, wenn der übertragene Zeitpunkt vom Service nicht als gültiger Zeitstempel interpretiert werden kann. Generell wird der Statuscode 503 zurückgeliefert, wenn das Service aktuell nicht ausgeführt wird.

6.2.2.1. Zusammenführen des Microservice 2 mit dem Monolithen

Das zweite Service bietet die Möglichkeit, Nachrichten, welche über die gewählte REST-Schnittstelle empfangen wurden, als SMS an den/die angegebenen/angegebene EmpfängerIn weiterzuleiten und SMS von dem angeschlossenen Modem zu lesen oder zu löschen. Diese Schnittstelle liefert als Ergebnis je nach Aufruf nur einen HTTP-Statuscode oder zusätzlich im Datenteil der Antwort, als XML beziehungsweise JSON encodiert, die Entitäten (dargestellt in Abbildung 6-7)

- „Status“, einer Implementierung des Interface „ISMSServiceStatus“, oder
- „ShortMessage“ zurück.

Die Anfrage an das Microservice beinhaltet im Sendemodus einen Request, welcher das Interface „ISendRequest“ implementiert.

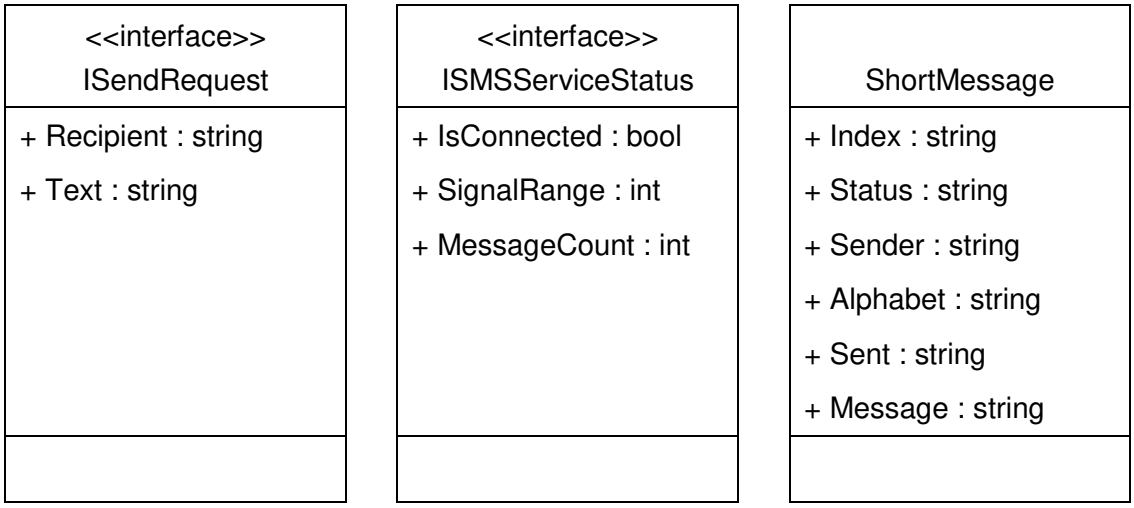


Abbildung 6-7: Klassendiagramm der Nachrichtenformate aus dem Microservice 2

Als serverseitige Implementierung wird in diesem Microservice der Technologiestapel Windows Communication Foundation (WCF) von Microsoft verwendet. Die WCF ist ein Framework zur Implementierung von serviceorientierten Anwendungen und kann für

verschiedenste Einsatzzwecke parametriert werden. Unter anderem kann über die WCF ein REST-Service abgebildet werden, indem ein „webHttp“-Endpunkt der Konfiguration hinzugefügt wird und die Serviceimplementierung zusätzlich mit den Attributen „WebInvoke“ versehen wird. Im Listing 6-2 ist das angewendete Service-Interface mit den notwendigen Attributen aufgeführt. Diese sind die von der WCF vorgegebenen Contract-Attribute, „ServiceContract“ und „OperationContract“. Zusätzlich muss über das WebInvoke-Attribut die Methode des Aufrufes bekannt gegeben werden und es kann bereits eine Einschränkung auf das Nachrichtenformat getroffen werden. Als Ergebnisse werden bei allen Aufrufen der Statuscode 503: „Service not available“ zurückgegeben, wenn der Dienst nicht verfügbar ist. SendSMS liefert bei erfolgreichem Absetzen der SMS den Statuscode 200 für Ok zurück, im Fehlerfall 406: „Not Accepted“.

```
[ServiceContract]
interface ISMSService
{
  [OperationContract]
  [WebInvoke(Method = "POST", RequestFormat = WebMessageFormat.Json,
    ResponseFormat = WebMessageFormat.Json, UriTemplate = "")]
  void SendSMS(SendRequest request);

  [OperationContract]
  [WebInvoke(Method = "GET", UriTemplate = "",
    ResponseFormat = WebMessageFormat.Json)]
  Status GetStatus();

  [OperationContract]
  [WebInvoke(Method = "GET", UriTemplate = "{index}",
    ResponseFormat = WebMessageFormat.Json)]
  ShortMessage ReadSMS(string index);

  [OperationContract]
  [WebInvoke(Method = "DELETE", UriTemplate = "{index}",
    ResponseFormat = WebMessageFormat.Json)]
  void DeleteSMS(string index);
}
```

Listing 6-2: WCF-Interface aus dem Microservice 2

Auf der Clientseite wird, wie in der Anwendung des Microservice 1, die Clientbibliothek von Microsoft zur WebAPI herangezogen, um Daten mit dem Service auszutauschen. Im Modul „AlarmSender“ werden die Funktionsaufrufe in das nicht mehr vorhandene „AtecTC35“-Modul durch REST-Serviceaufrufe zum Service ersetzt und der entsprechende Statuscode des Microservice abgefragt. Durch das Abarbeiten der zu sendenden Alarme ist es möglich, mehrere Versuche zu starten, die Nachricht hintereinander oder parallel zu senden. Hier ist die Stärke der Microservices zu erkennen, da es nun möglich ist, auf mehr als nur einem Computer ein Modem mit dem Sendedienst zu betreiben.

6.2.3 Analyse der Software mit Microservices

Im Endausbau der Programmierung des Monolithen mit Microservices wurden zwei Dienste abgespalten. Zum einen eine Schnittstelle zu einem Quellsystem und zum anderen eine Schnittstelle zu einem Versendesystem. In diesem Kapitel wird das System mit dem Sichtenmodell, welches bereits im Kapitel 6.2.1 angewendet wurde, analysiert.

6.2.3.1. Kontextsicht

Der fachliche Kontext der Anwendung wurde aufgeteilt. Durch die Analyse im Kapitel 6.2.2.1 konnten Übergänge aus der monolithischen Anwendung in Microservices und deren gekapselte fachliche Domäne identifiziert werden. Daraus ergeben sich aus Kontextsicht des neuen Systems, das Hauptsystem „AVS“ als monolithische Anwendung, die Quell- und Versendesysteme, sowie die neuen Dienste zur Anbindung dieser: „InTouch-Service“ und „SMS-Service“.

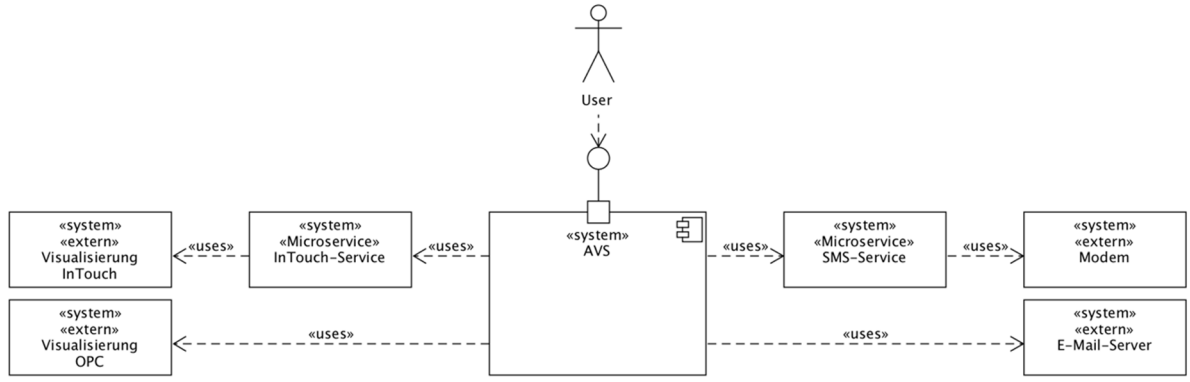


Abbildung 6-8: Kontextsicht des Zielsystems

Wie in der Abbildung 6-8 ersichtlich, wurde für diese Arbeit jeweils ein Quellsystem und ein Versendesystem in die Microservice-Architektur überführt. Die verbleibende monolithische Anwendung wurde umgestellt von Prozeduraufrufen im Prozess hin zu REST-Serviceaufrufen zu den neuen externen Microservices. Dem/Der BenutzerIn bleibt dies verborgen, da dessen/deren Interaktion mit dem System weiterhin über die Benutzerschnittstelle des monolithischen Systems abgehandelt wird. Die daraus resultierenden Schnittstellen und Übertragungstechniken sind in der Tabelle 6-3 aufgelistet.

Nachbarsystem	Schnittstelle	technisches Protokoll
InTouch-Service	Alarme lesen	REST (WebAPI) über HTTP
Visualisierung OPC	Rohdatendaten lesen	OPC Classic DA, DCOM
SMS-Service	SMS versenden	REST (WebAPI) über HTTP
E-Mail-Server	E-Mail versenden	SMTP

Tabelle 6-3: Kontextsicht des Zielsystems

6.2.3.2. Bausteinsicht

Die Bausteinsicht hat sich in Bezug auf das Ausgangssystem an zwei Stellen verändert. Wie in Kapitel 6.2.2.2 beschrieben, wurde die Logik für den Zugriff auf das Fremdsystem in ein Microservice ausgegliedert. Dieses Service ist idempotent entwickelt und enhält keinen eigenen Datenspeicher. Es kann als eine Art Proxy auf das ursprüngliche Quellsystem betrachtet werden. Für den Zugriff bietet das Service eine Schnittstelle mit REST als Zugriffsmöglichkeit an. Darüber kann der aktuelle Status, der letzte Zeitpunkt eines Alarmeintrages im Quellsystem und eine Liste von Alarmen ab einem bestimmten Zeitpunkt eingesehen werden. Das Service wird durch eine eigene Konfiguration, mit Hilfe von Microsoft's Configuration Bibliothek, parametriert und erhält darüber eine Verbindungszeichenfolge zur Datenbank des Quellsystems sowie eine URL, unter welchem der Dienst seine REST-Schnittstelle anbietet. Der Zugriff darauf erfolgt aus dem Modul „AlarmLink" und ist in der Abbildung 6-9 dargestellt.

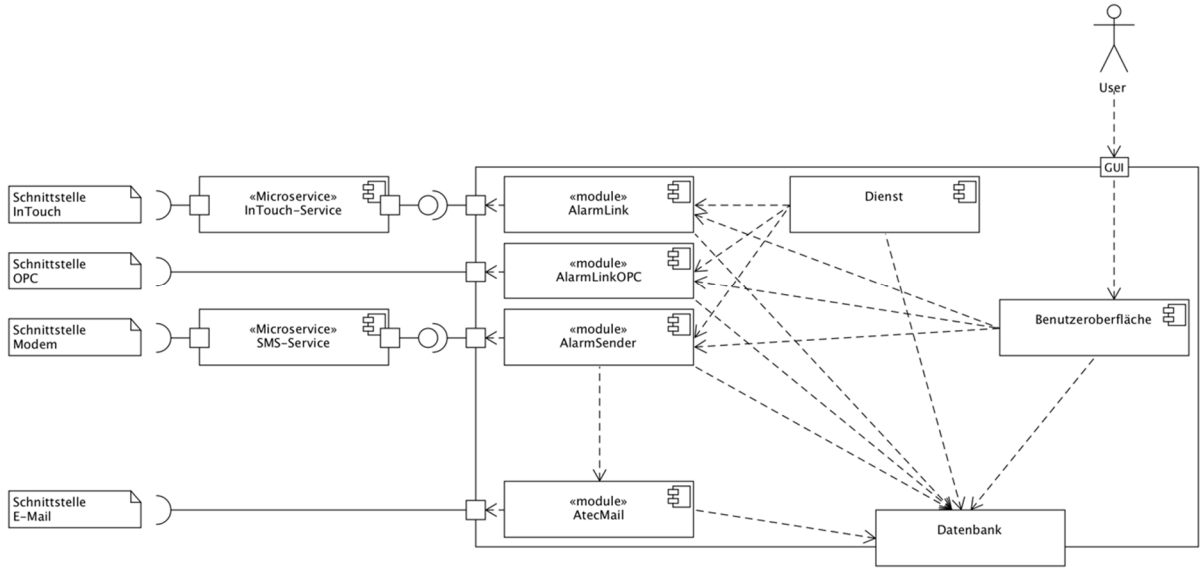


Abbildung 6-9: Bausteinsicht des Zielsystems

Aus der Abbildung 6-9 kann ebenfalls die Veränderung des Sendemechanismus abgelesen werden. Hierzu wurde das Modul „AtecTC35" zur Kommunikation mit einem Modem über eine serielle Schnittstelle in das neue Microservice „SMS-Service" ausgegliedert und dessen Dienste mittels REST-Schnittstelle verfügbar gemacht. Die Zugriffslogik dafür wurde in das Modul „AlarmSender" der monolithischen Anwendung eingefügt. Die Schnittstelle bietet die Möglichkeit, den Status des Service abzufragen, SMS zu versenden, zu lesen und zu löschen. Dieses Service wird, wie das Service zum Lesen von Alarmen aus dem Quellsystem InTouch, mittels einer Konfigurationsdatei parametriert. Darüber wird die URL für die REST-Schnittstelle und folgende spezifische Informationen für die Kommunikation mit dem Modem hinterlegt:

- Servicenummer des Netzanbieters,
- PIN der eingelegten SIM-Karte
- und der Name der zu verwendenden COM-Schnittstelle.

6.2.3.3. Laufzeitsicht

Das Durchlaufen der Alarme von einem Quellsystem bis zum Versenden hat sich im Vergleich von der monolithischen Ausgangsanwendung zur Anwendung mit Microservices an zwei Stellen geändert. Das Modul „AlarmLink“ liest in der geänderten Anwendung die Alarme über eine REST-Schnittstelle ein. Dies ist in der Abbildung 6-10 im ersten Abschnitt erkennbar. Der Aufruf der Statusabfrage erfolgt über die Methode GET auf die URL „/intouchdb“. Das Microservice antwortet darauf im dargestellten Fall mit dem HTTP-Statuscode 200, gleichbedeutend mit „OK“, und der Entität „Status“. Ist diese Verbindung und damit die Kommunikation mit dem Quellsystem sichergestellt, wird darauffolgend eine Liste von Alarmen seit dem letzten Durchlauf über die Methode GET abgerufen. Dies erfolgt über die URL „/intouchdb/{since}“, wobei der Platzhalter „since“ mit dem tatsächlichen Datumswert ersetzt wird und somit das Service durch URL-Parameter abgefragt wird. Im Gut-Fall retourniert das Service den HTTP-Statuscode für OK mitsamt einer Liste der Entität „Alarm“. Die bisherige direkte Abfrage des Quellsystems ist ab sofort gekapselt in dem Microservice und kann gegebenenfalls ohne Änderung der monolithischen Hauptanwendung angepasst oder gänzlich getauscht werden.

Die zweite Veränderung gegenüber dem Ausgangssystem liegt im Verfahren zur Zustellung von Nachrichten über den Kommunikationskanal SMS. Das Ausgangssystem verarbeitete diesen Vorgang mit einem direkten Funktionsaufruf aus dem Modul „AlarmSender“ auf das Modul „AtecTC35“. Im geänderten System wurde das Modul „AtecTC35“ in das neue Microservice „SMS-Service“ überführt. Dadurch wird aus dem Modul „AlarmSender“ durch die Klasse „DeliveryTC35“ die REST-Schnittstelle des Microservice angesprochen und Nachrichten darüber versendet. Das Service verwaltet das angeschlossene Modem und gibt auf Anfrage dessen Status preis. Im gegebenen Anwendungsfall, wie in Abbildung 6-10 dargestellt, wird die zu sendende Nachricht als JSON encodiert im Datenteil des POST-Aufrufes übertragen und das Service antwortet darauf im Gut-Fall mit dem HTTP-Statuscode für OK.

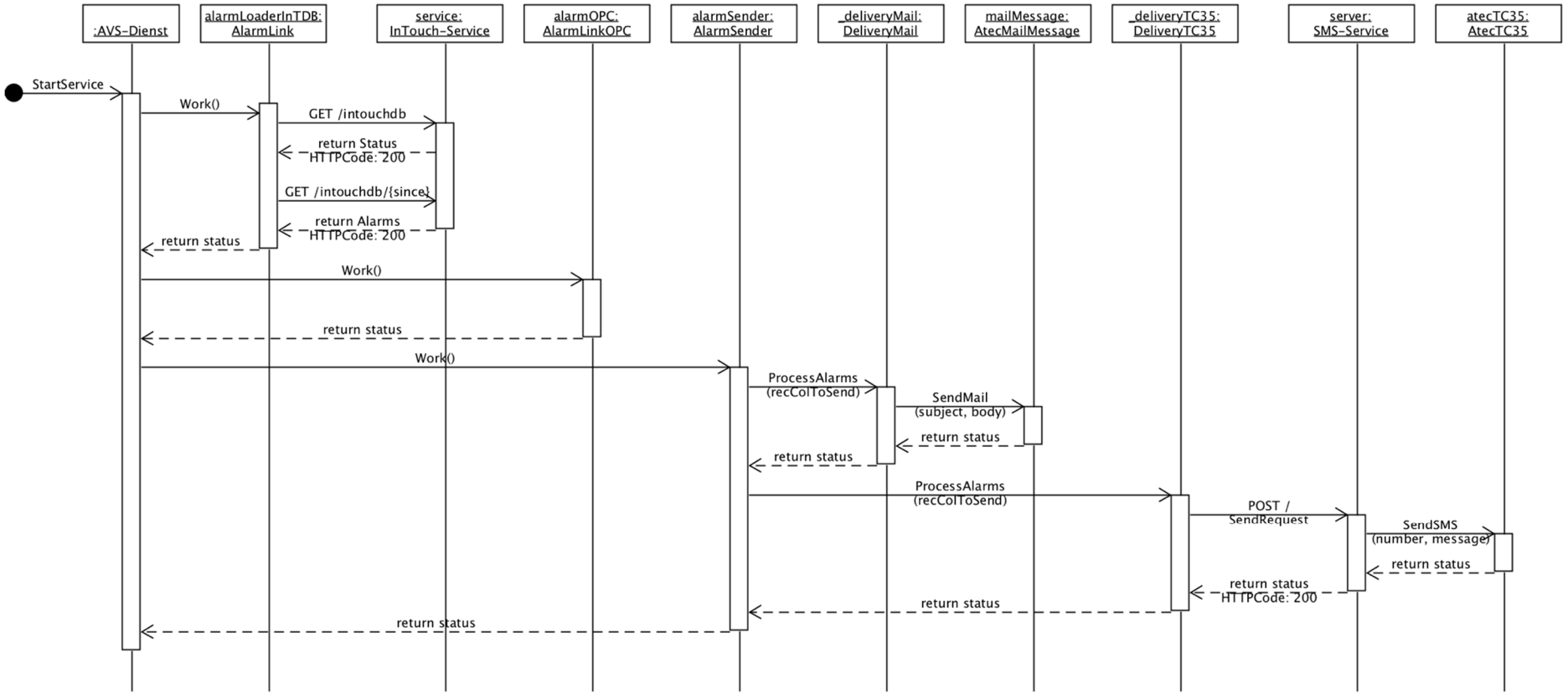


Abbildung 6-10: Laufzeitsicht des Zielsystems

6.2.3.4. Verteilungssicht

Wie aus den bisherigen Sichten zu entnehmen ist, sind die Microservices von der monolithischen Ausgangsanwendung abgespalten. Dies spiegelt sich auch in der Verteilungssicht wider. Die enthaltenen Artefakte dieser Sicht sind in der Tabelle 6-4 aufgelistet.

Artefakt	Enthaltene Bausteine
AVS-Client	AlarmLink, AlarmLinkOPC, AlarmSender, AVSClientViewer
AVS-Service	AlarmLink, AlarmLinkOPC, AlarmSender, AtecMail
InTouch-Service	-
SMS-Service	-
Modem	-
E-Mail-Server	-

Tabelle 6-4: Artefakte der Verteilungssicht des Zielsystems

In der Abbildung 6-11 ist eine Variante der Verteilung dargestellt. Das System kann in dieser Konfiguration nach wie vor auf zwei Rechnern betrieben werden, einem Client für die Darstellung der Benutzerschnittstelle, welche über eine Datenbankschnittstelle direkt mit der Datenbank des Systems kommuniziert, und einem Server für die Verarbeitung der Daten von den Quellsystemen und dem Versenden von ausgewählten Daten über die Versendesysteme.

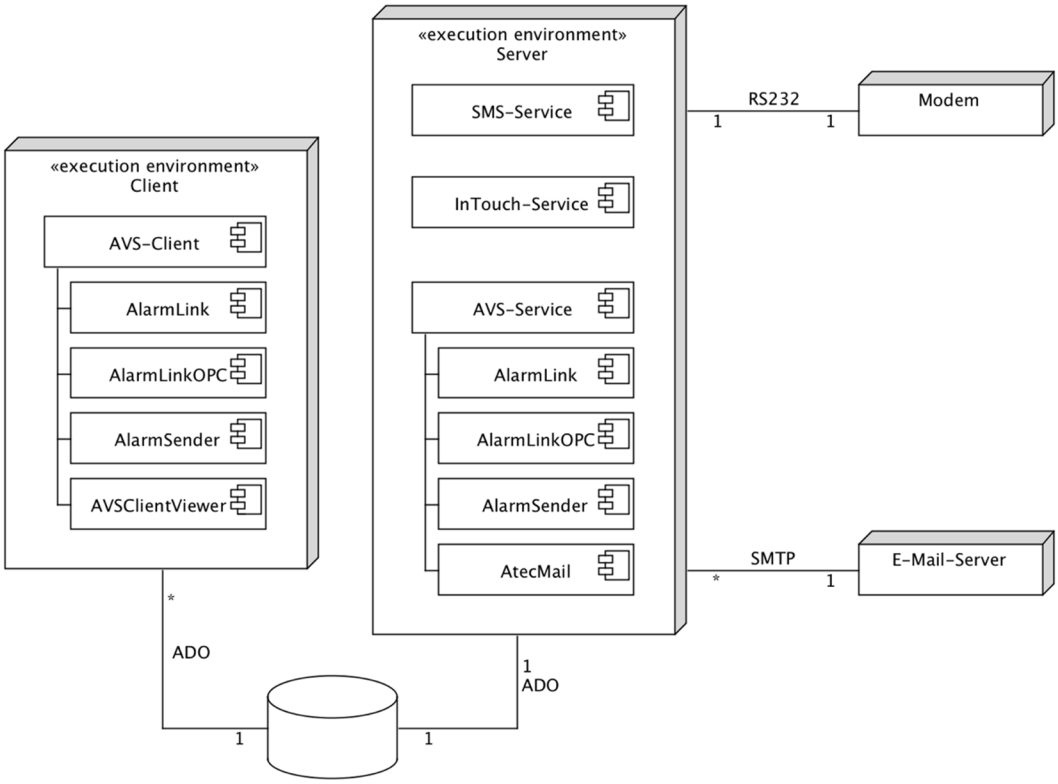


Abbildung 6-11: Verteilungssicht des Zielsystems Variante 1

Die Kommunikation mit dem Modem und dem E-Mail-System wird vom Server übernommen und ist im Falle des Modems durch die serielle Schnittstelle ortsgebunden. Die Kommunikation mit den Services wird über REST abgehandelt und verlässt den Server nicht.

Alternativ kann das System auch in Variante zwei (Abbildung 6-12) betrieben werden. Hierbei sind die Microservices auf externen Rechnern installiert und die Kommunikation zwischen Server und Services erfolgt über das Netzwerk. Dadurch wird die Ortsgebundenheit des Modems aufgehoben und die Möglichkeit geschaffen, mehr Modems im System zu verwenden als im ursprünglichen System serielle Schnittstellen am Server vorhanden waren.

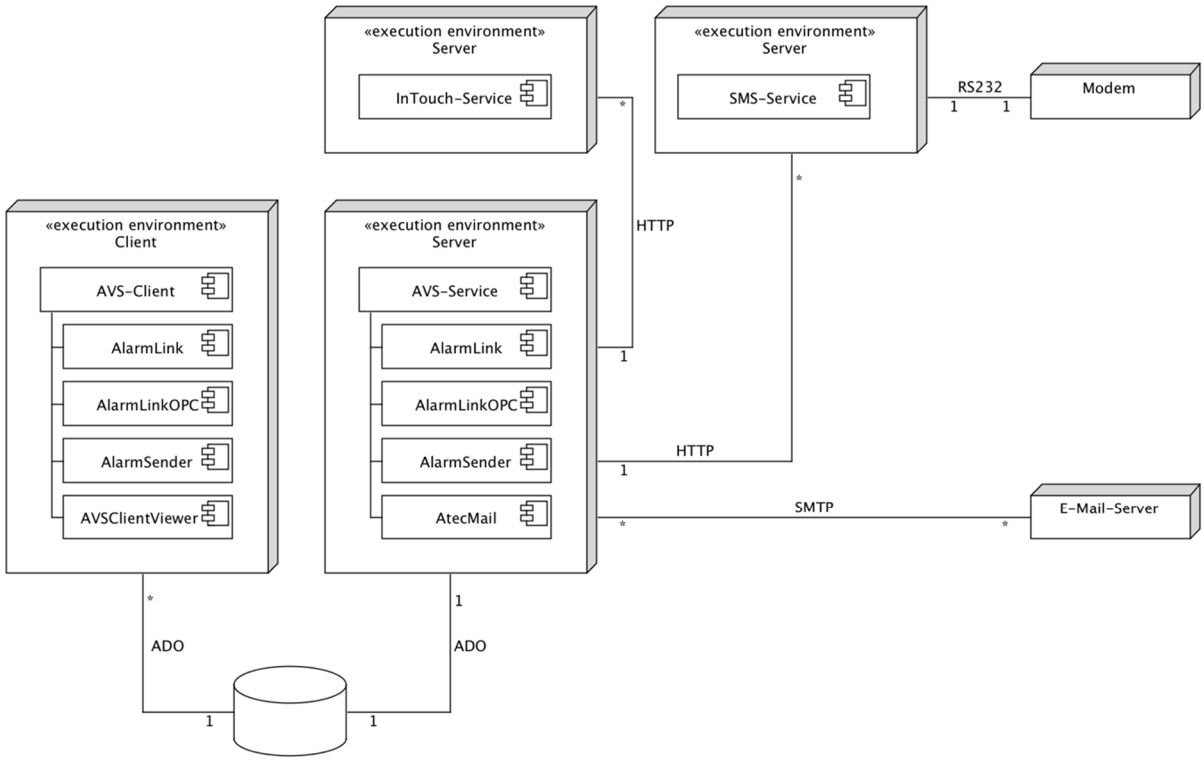


Abbildung 6-12: Verteilungssicht des Zielsystems Variante 2

6.3 Vergleiche

Dieses Kapitel widmet sich der Erklärung des Aufbaus der Vergleiche. Zu Beginn wird der Metrikvergleich zwischen Ausgangssoftware und Zielsystem erklärt und anschließend wird auf die Umgebung der Datenerhebung zur Änderungsgeschwindigkeit sowie auf die Interviews mit den EntwicklernInnen eingegangen.

6.3.1 Metriken

Zum Vergleich der Metriken wurden zwei verschiedene Erhebungssysteme herangezogen. Zum einen die Entwicklungsumgebung Visual Studio 2013 des Herstellers Microsoft in der Professional Edition und zum anderen die Software sonarqube des Herstellers sonarsource. sonarqube wurde in Version 6.1 eingesetzt.

Aus dem ersten Tool, Visual Studio, wurden die Ergebnisse zu folgenden Metriken auf Solution-Ebene, Projekt-Ebene, Klassen-Ebene und Datei-Ebene exportiert:

- Wartbarkeitsindex nach Microsoft (Kapitel 5.2.9),
- zyklomatische Komplexität,
- Vererbungstiefe,
- Klassenkopplung
- und die Anzahl der Codezeilen.

Die Metriken des Tools sonarqube sind ein Komplexitätsmaß, welches die Pfade durch eine Methode zählt, und die Größe des Quellcodes in Form der Anzahl von Codezeilen. Das Komplexitätsmaß der Firma sonarsource wird berechnet durch Schlüsselwörter im Quellcode, so startet jede Methode mit dem Wert 1 und bestimmte Anweisungen, wie IF, WHILE oder FOR, erhöhen dieses Maß um 1. (Racodon, 2016)

6.3.2 Änderungsgeschwindigkeit & Interviews

Die Messung Änderungsgeschwindigkeit am Quellcode und die vertiefenden Interviews zur Klärung der Fragestellung in der Hypothese 3 wurden in einer Sitzung mit den betroffenen EntwicklerInnen durchgeführt. Die Auswahl der Stichprobe, die zu lösenden Aufgabenstellungen sowie die Rahmenbedingungen dieser Erhebung sind in diesem Kapitel detailliert beschrieben.

6.3.2.1. Stichprobe

Die Stichprobe wurde aus der Grundgesamtheit aller ProgrammiererInnen mit der Einschränkung der Kenntnis der Programmiersprache C# gewählt. Aus diesen wurden vier Personen ohne Kenntnis der Ausgangs- oder Zielanwendung aus dem Großraum Graz in der Steiermark, Österreich, gewählt. Die Auswahl wurde eingeschränkt auf örtliche Verfügbarkeit für die Durchführung der Erhebung der Änderungsgeschwindigkeit. Es wurde Wert darauf gelegt, SoftwareentwicklerInnen aus verschiedenen Branchen und aus Betrieben mit unterschiedlicher Größe in der Stichprobe zu vereinen.

Es konnte ein Entwickler aus einem Unternehmen mit 25 MitarbeiterInnen, ein Entwickler aus einem Unternehmen mit 100 MitarbeiterInnen, ein Entwickler aus einem Unternehmen mit 400 MitarbeiterInnen und ein Entwickler aus einem Unternehmen mit 140 MitarbeiterInnen zur Erhebung der Änderungsgeschwindigkeit und für das Interview gewonnen werden.

6.3.2.2. Erhebung der Änderungsgeschwindigkeit

Zur Durchführung sind zwei Aufgaben vorgesehen. Diese sollen Anforderungen aus dem Unternehmensalltag widerspiegeln, um die Aussagekraft an Unternehmensstandards

anzupassen. Die erste Anforderung zur Umsetzung ist die Tatsache, dass das Versenden der Nachrichten über das Modem nicht immer zuverlässig funktioniert. Dies kann bedingt sein durch Probleme bei der Kommunikation über die serielle Schnittstelle oder durch Verbindungsschwierigkeiten mit dem Netzbetreiber. Dadurch ergibt sich die Anforderung 1:

- Das System muss in der Lage sein, bis zu fünf Modems für den Nachrichtenversand via SMS anzusprechen, welche der Reihe nach abgefragt werden, bis die Zustellung erfolgreich war.

Für die Lösung dieser Aufgabenstellung gibt es im Anwendungsfall der monolithischen Applikation, als auch der Microservices drei Lösungsmöglichkeiten. Diese sind im ersten Fall die Lösungsmöglichkeit 1, das Speichern einer Liste von Zustellungsobjekten des Types „DeliveryTC35“ in der Klasse „AlarmSender“ und dem iterativen Durchlaufen dieser bei dem Versuch, die zu verarbeitende Nachrichtenliste zuzustellen. Die zweite Lösungsmöglichkeit ist das Vorhalten einer Liste von „AtecTC35“ Objekten in der Klasse „DeliveryTC35“. Schließlich ist die dritte Lösungsmöglichkeit das direkte Ansprechen von bis zu fünf verschiedenen Modems im Modul „AtecTC35“ über fünf verschiedene serielle Schnittstellen. Im Quellcode Microservice-Anwendung kann das Problem durch dieselben Möglichkeiten wie in der Aufgabenstellung für die monolithischen Anwendungen gelöst werden, einzig der Zugriff auf mehrere „AtecTC35“-Objekte in der Klasse „DeliveryTC35“ ist nicht möglich, da hier der Zugriff durch den Einsatz der REST-Schnittstelle angepasst wurde. Diese Schnittstelle kann jedoch auf verschiedene Endpunkte angewendet werden, somit ergeben sich für den Einsatz der Microservices ebenfalls drei Lösungspfade.

Die zweite Aufgabe ist ein Fehler im Programm. Das Quellsystem InTouch zeichnet den/die quittierenden/quittierende BenutzerIn auf. Dieser/Diese wird im aktuellen System nicht übernommen und steht nicht zur Auswertung bereit. Für eine lückenlose Nachverfolgung von Alarmen ist diese Information jedoch unabdingbar. Die ursprüngliche Anforderung wurde nicht vollständig umgesetzt und die Fehlerquelle soll von der Testperson gefunden werden. Dadurch ergibt sich die Anforderung 2:

- Das System muss in der Lage sein, den/die quittierenden/quittierende BenutzerIn aus einem Quellsystem, sofern vorhanden, zu übernehmen und für spätere Auswertungen zur Verfügung zu stellen.

Die Lösungspfade für diese Aufgabenstellung sind in der monolithischen Anwendung und der Anwendung mit Microservices unterschiedlich. In ersterem Fall kann der Fehler durch die Korrektur eines Datenbankaufrufes behoben werden. Im Fall der Microservices muss jedoch zusätzlich die Schnittstelle angepasst werden. Dies kann erreicht werden durch das Erweitern der Datentransferklassen auf beiden Seiten, in der monolithischen Anwendung und dem Microservice.

6.3.2.3. Durchführung

Die Erhebung der Änderungsgeschwindigkeit wurde in einem abgeschlossenen Raum durchgeführt. Der Testperson wurde ein Notebook zur Verfügung gestellt, über welches

diese die monolithische Anwendung und die Microservice-Anwendung zum gegebenen Zeitpunkt analysieren konnte. Der Ablauf dieser Erhebung wurde mit allen vier Testpersonen gleichermaßen eingehalten:

- Begrüßung.
- Erklärung des Anwendungsfalles der Ausgangsanwendung mit typischen Problemfällen.
- Darstellung des Aufbaues der monolithischen Anwendung, sowie der Anwendung mit den Anpassungen der Microservices mithilfe der Baustein- und Laufzeitsichten aus den Kapiteln 6.2.1 und 6.2.3.
- Erklärung der Zeitnahme.
- Erklärung der Aufgabenstellung 1 und Bereitstellen des Quellcodes in der Entwicklungsumgebung.
- Freigabe für die Testperson zur Analyse und Behebung der Aufgabenstellung 1 mit Zeitmessung und Dokumentation der Ergebnisse.
- Erklärung der Aufgabenstellung 2 und Bereitstellen des Quellcodes in der Entwicklungsumgebung.
- Freigabe für die Testperson zur Analyse und Behebung der Aufgabenstellung 2 mit Zeitmessung und Dokumentation der Ergebnisse.

Die erhobenen Zeiten spiegeln die Dauer wider, welche die Testperson benötigte, um die richtige Quellcodestelle im Programm zu finden und die vollständige Erklärung des gesamten Lösungspfades darzulegen.

Im Anschluss der Erhebung zur Änderungsgeschwindigkeit wurden in einem direkten Einzelinterview vertiefende Fragen zu den Themen Microservices und Codequalität gestellt. Dieses durch den Leitfaden aus Anhang A geführte Interview wurde nach Zustimmung der Testperson aufgezeichnet. Das Interview wurde in die Einleitung, einführende unterstützende Fragestellungen und die Kernfrage der Veränderung der Codequalität in den analysierten Quellcodebeispielen eingeteilt. Zur Verabschiedung wurden soziodemographische Daten erhoben und der/die InterviewpartnerIn mit dem Hinweis entlassen, dass die Ergebnisse der Untersuchung in der Arbeit dokumentiert und öffentlich zugänglich sind.

7 ERGEBNISSE

In diesem Abschnitt sind die Ergebnisse der Forschung dokumentiert. In der Reihenfolge Metrikanalyse und -vergleich, Darstellung der Änderungsgeschwindigkeit und den Interviews sind die Vergleiche zwischen Ausgangssoftware und Zielsoftware textuell und grafisch dargelegt.

7.1 Metriken

Zum Vergleich der Softwaremetriken wurden zwei verschiedene Tools eingesetzt, welche zum Teil unterschiedliche Metriken erhoben. Die Detailergebnisse sind in diesem Kapitel gegliedert nach der Ebene, auf welcher die Metriken erhoben wurden.

7.1.1 Modulebene

Auf Ebene der Module wurde der Umfang der Anwendung und die Komplexität der Anwendung erhoben und mittels Summierung und Durchschnittsbildung analysiert und verglichen. Im Anschluss an die jeweilige Metrik sind die Ergebnisse auch grafisch dargestellt.

7.1.1.1. Lines of Code

Die Ergebnisse der Metrik Lines of Code der Modulebene aus der Entwicklungsumgebung Visual Studio sowie die Ergebnisse aus dem Tool sonarqube sind in der Tabelle 7-1 im Detail dargestellt. Diese wurde eingeteilt in zwei Spalten mit den berechneten Werten der eingesetzten Tools und weiter unterteilt für die Detailangaben zum Monolithen und den Microservices. Das Tool sonarqube setzt zur Berechnung der Lines of Code auf Zeilen in einer Datei inklusive generiertem Code von der Entwicklungsumgebung. Dadurch sind die Ergebnisse dieses Tools im direkten Vergleich zum Tool Visual Studio höher.

Im Vergleich von Ausgangssoftware zu Zielsystem hat sich der Umfang der Software in Summe im Tool Visual Studio um 1,23% und im Tool sonarqube um 3,91% gesteigert. Die Steigerung des Umfanges ist erklärbar durch den erhöhten Aufwand der Kommunikation, ein Umstand, welcher den Nachteilen der Microservices aus Kapitel 2.2.3 zuzuschreiben ist.

Modul	Lines of Code Visual Studio		Lines of Code sonarqube	
	Monolith	Microservices	Monolith	Microservices
AlarmLink	589	464	1264	1041
AlarmLinkOpc	1677	1677	1305	1305
AlarmSender	1217	1171	2050	2009
AtecMail	61	61	139	139
AtecTC35	320	-	622	-
AtecTemplateParser	509	509	971	971
AVS.NETService	2679	2679	205	205
AVSClient	2676	2676	4914	4914
AVSClientViewer	2271	2271	3754	3754
Atec.AVS.IntouchDB.Service	-	217	-	511
Atec.SMS.Service	-	421	-	970
Summe	11999	12146	15224	15819
Differenz in %		↑ 1,23		↑ 3,91

Tabelle 7-1: Detailergebnisse der Metrik Lines of Code auf Modulebene

Dem gegenüber steht eine Verringerung des Umfangs in den Modulen „AlarmLink“ und „AlarmSender“, aus welchen Funktionen extrahiert wurden. Die grafische Darstellung der Veränderungen in Abbildung 7-1 veranschaulicht diese Veränderung. Die unangetasteten Module des Programmierprozesses sind „AlarmLinkOpc“, „AtecMail“, „AtecTemplateParser“, „AVS.NETService“, „AVSClient“ und „AVSClientViewer“. Diese sind in der Abbildung nicht dargestellt, da sie keine Veränderung im Quellcodeumfang aufweisen.

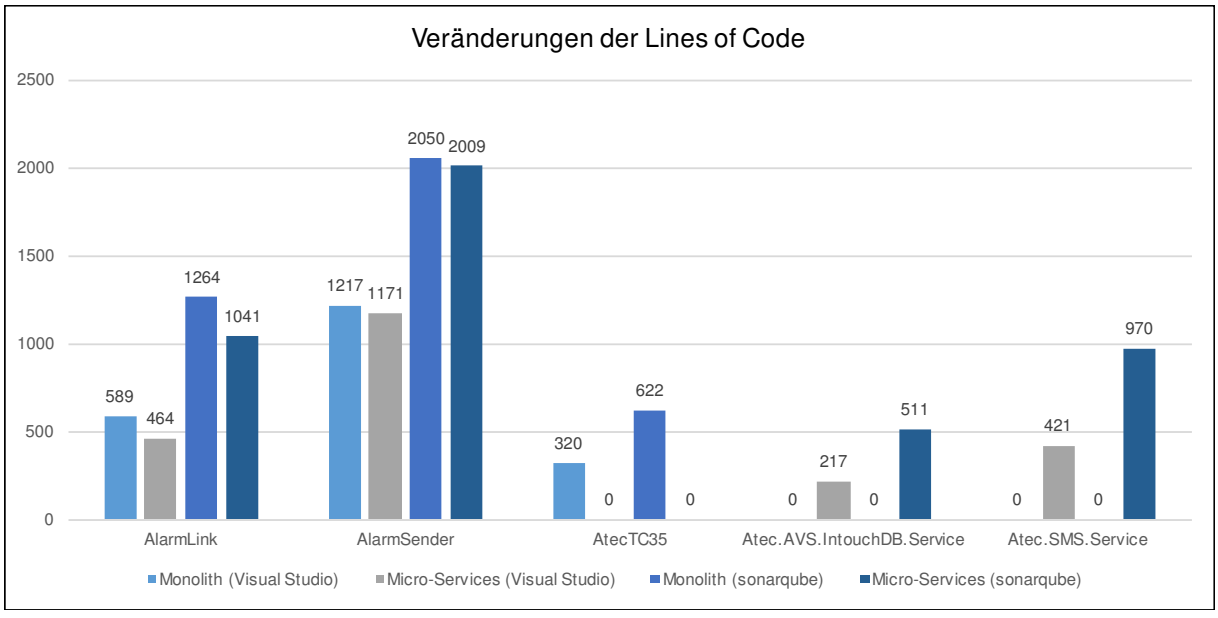


Abbildung 7-1: Vergleich der Metrik Lines of Code auf Modulebene

7.1.1.2. Komplexität

Die Komplexität konnte im Durchschnitt in beiden Anwendungen durch den Einsatz der Microservice-Architektur verringert werden. Beide Tools zur Berechnung der Metriken weisen in den bearbeiteten Modulen verringerte Komplexitäten auf und der durchschnittliche Wert konnte im Tool Visual Studio um 7,26% und im Tool sonarqube um 8,69% verringert werden. Die Ergebnisse sind in der Tabelle 7-2 aufgelistet.

Modul	Zyklomatische Komplexität Visual Studio		Komplexität sonarqube	
	Monolith	Microservices	Monolith	Microservices
AlarmLink	243	185	280	200
AlarmLinkOpc	600	600	118	118
AlarmSender	444	432	437	420
AtecMail	21	21	21	21
AtecTC35	116	-	127	-
AtecTemplateParser	202	202	209	209
AVS.NETService	1471	1471	21	21
AVSClient	849	849	892	892
AVSClientViewer	744	744	782	782
Atec.AVS.IntouchDB.Service	-	138	-	109
Atec.SMS.Service	-	191	-	157
Durchschnitt	521,11	483,30	320,78	292,90
Differenz in %		↓ 7,26		↓ 8,69

Tabelle 7-2: Detailergebnisse der Metrik Komplexität auf Modulebene

In der grafischen Aufbereitung in Abbildung 7-2 kann dies visuell nachvollzogen werden. In der Grafik sind nur jene Module dargestellt, in denen eine Veränderung von Monolith zu Microservice festgestellt wurde. Diese sind die Module „AlarmLink“ und „AlarmSender“ aus dem Monolithen, welche mit verringerter Komplexität im Microservice weiterexistieren, sowie die neuen Microservices „Atec.AVS.IntouchDB.Service“ und „Atec.SMS.Service“. Letzteres beinhaltet das Modul „AtecTC35“ vollständig. Trotz des gesteigerten Kommunikationsaufwandes und des gesteigerten Umfanges der Anwendung konnte die Komplexität im Durchschnitt gesenkt werden.

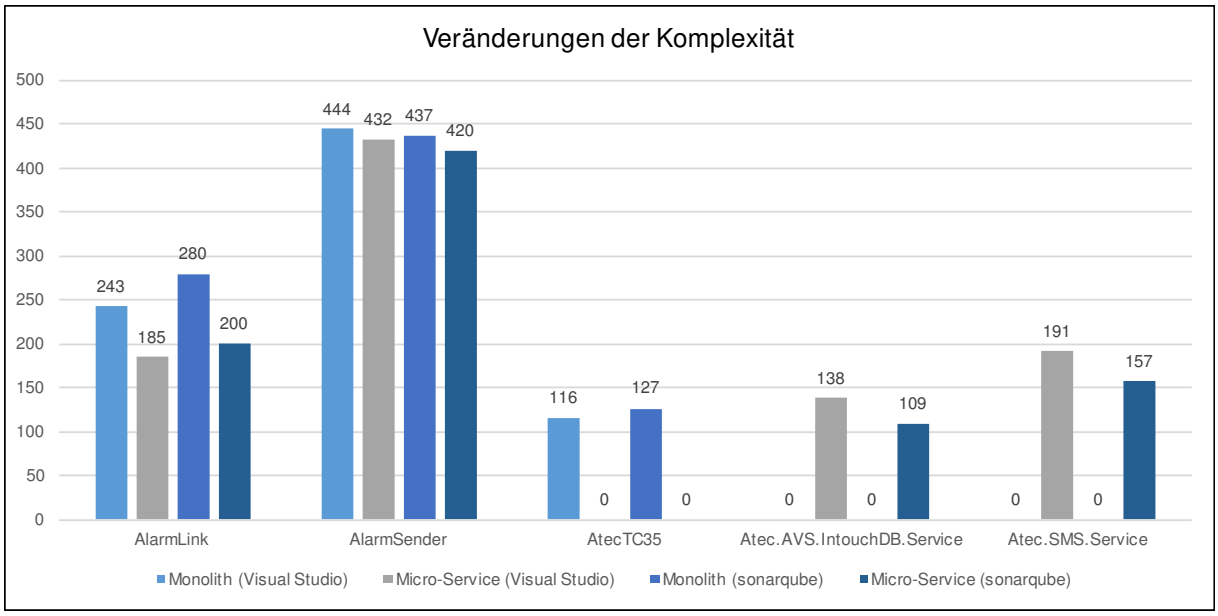


Abbildung 7-2: Vergleich der Metrik Komplexität auf Modulebene

7.1.1.3. Wartbarkeitsindex, Vererbungstiefe und Klassenkopplung

Zur Unterstützung der Komplexitätsmetrik wurden weitere im Tool Visual Studio verfügbare Metriken erhoben. Diese sind der Wartbarkeitsindex nach Microsoft, die Vererbungstiefe und die Metrik der Klassenkopplung. Die Detailergebnisse sind in der Tabelle 7-3 aufgelistet.

Modul	Wartbarkeitsindex		Vererbungstiefe		Klassenkopplung	
(µS = Microservice)	Monolith	µS	Monolith	µS	Monolith	µS
AlarmLink	83	82	8	8	47	56
AlarmLinkOpc	72	72	8	8	141	141
AlarmSender	77	77	8	8	94	107
AtecMail	70	70	1	1	8	8
AtecTC35	69	-	1	-	18	-
AtecTemplateParser	82	82	1	1	18	18
AVS.NETService	81	81	4	4	81	81
AVSClient	82	82	8	8	167	167
AVSClientViewer	84	84	7	7	155	155
Atec.AVS.IntouchDB.Service	-	85	-	3	-	35
Atec.SMS.Service	-	88	-	2	-	49
Durchschnitt	77,78	80,30	5,11	5,00	81,00	81,70
Differenz in %		↑ 3,24		↓ 2,17		↑ 0,86

Tabelle 7-3: Detailergebnisse der Metriken Wartbarkeitsindex, Vererbungstiefe und Klassenkopplung auf Modulebene

Daraus lassen sich keine relevanten Schlussfolgerungen ableiten. Die durchschnittliche Wartbarkeit konnte durch den Einsatz der Microservices gesteigert werden. Diese Metrik ist in eine Skala von 0% bis 100% eingeteilt. Das bedeutet, ein höherer Wert wird als wartbarer wahrgenommen. Jedoch kann nach näherer Betrachtung der Detailergebnisse festgestellt werden, dass sich die Veränderungen auf Modulebene nur unwesentlich unterscheiden. Der Unterschied von 3,24% kann direkt auf die größere Anzahl der Module zurückgeführt werden. Die Vererbungstiefe konnte im Gesamtergebnis im Durchschnitt verringert werden, wobei ein geringerer Wert das Ziel ist, aber auch diese Metrik weist bei näherer Betrachtung auf Modulebene zu geringe Unterschiede auf, als dass diese eine Aussagekraft für die Beantwortung der Forschungsfrage hätten. Ähnlich verhält es sich mit der Metrik der Klassenkopplung, welche bereits im Gesamtergebnis keine markante Veränderung aufweist. Zur Veranschaulichung sind die Ergebnisse der Metriken Wartbarkeitsindex und Klassenkopplung in der Abbildung 7-3 grafisch dargestellt. Daraus ist ersichtlich, dass der Wartbarkeitsindex in den Modulen „AlarmLink“ und „AlarmSender“ minimale Unterschiede aufweist und die durchschnittliche Verbesserung aus dem Wegfall des wenig wartbaren Moduls „AtecTC35“ und dem Hinzufügen der gut wartbaren Microservices stammt. Die Klassenkopplung wurde in allen Vergleichen verschlechtert, dies ist symbolisiert durch die höhere Metrikzahl in allen Modulen.

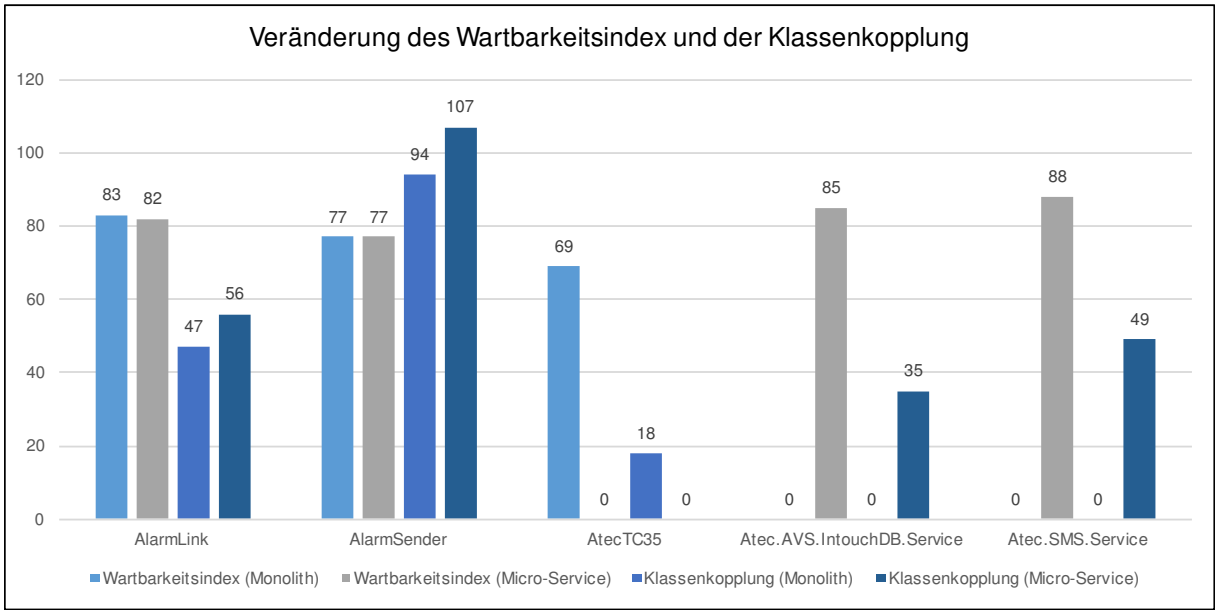


Abbildung 7-3: Vergleich der Metriken Wartbarkeitsindex und Klassenkopplung auf Modulebene

7.1.2 Klassenebene

Auf Klassenebene verhalten sich die Veränderungen gleich wie auf der höheren und aggregierten Modulebene. Eine vollständige Auflistung der gesamten Ergebnisse kann dem Anhang B entnommen werden.

7.1.3 Zusammenfassung

Zusammenfassend gibt es für die Hypothese 2, „Die Anwendung von Microservices reduziert die Code-Komplexität“, Indizien zur Bestätigung. Die Metriken des Softwareumfanges sowie die Metriken Wartbarkeitsindex, Vererbungstiefe und Klassenkopplung weisen zu geringe Veränderungen auf, um die Codequalität durch den Einsatz von Microservices zu beeinflussen. Dies wurde durch zwei Tools unterschiedlicher Hersteller erhoben.

7.2 Änderungsgeschwindigkeit

Die Erhebung der Änderungsgeschwindigkeit erfolgte mit vier Testpersonen an der monolithischen Ausgangsanwendung und dem Zielprodukt mit Microservices. Die Testpersonen wurden mit den Anforderungen aus dem Kapitel 6.3.2.2 konfrontiert und mit der Lösungsfindung beauftragt. Dazu wurde pro Testperson jeweils eine Anforderung für den Monolithen und die andere Anforderung für die Microservices gewählt. Im Detail wurden die Testpersonen A bis D mit den Anforderungen aus Tabelle 7-4 betraut. Nachfolgend sind die Beobachtungen des Interviewers festgehalten.

Testperson	Anwendung	Anforderung	Zeit 1	Zeit 2
A	Monolith	1	5:24	8:12
	Microservice	2	1:06	5:30
B	Microservice	1	8:55	10:40
	Monolith	2	6:54	7:17
C	Monolith	1	1:10	1:50
	Microservice	2	0:58	4:43
D	Microservice	1	3:02	3:43
	Monolith	2	3:21	3:24

Tabelle 7-4: Änderungsgeschwindigkeiten nach Anforderungen auf Anwendungen

Testperson A wurde am 27. 10. 2016 um 13:00 vor die Aufgabe zur Lösung der Anforderung 1 an der monolithischen Ausgansanwendung gestellt. Die Testperson fand sich gut in der Anwendung zurecht und erkannte zum Zeitpunkt 5 Minuten und 24 Sekunden die richtige Programmstelle zur Lösung des Problems. Der Lösungsweg wurde zum Zeitpunkt 8 Minuten und 12 Sekunden richtig und vollständig dargelegt. Die Testperson bediente sich des Lösungsweges 1 aus Kapitel 6.3.2.2. Die zweite Aufgabenstellung folgte unmittelbar darauf und wurde der Testperson zusammen mit dem Quellcode in der Entwicklungsumgebung gestellt. Für die korrekte Identifikation der Problemstelle wurden 1 Minute und 6 Sekunden benötigt. Die Darlegung des vollständigen Lösungsweges benötigte 4 Minuten und 24 Sekunden. Die Testperson hatte die Problemstelle gefunden, das Problem der Kommunikation jedoch im Verhältnis lange nicht lösen können. Die lose Kopplung der

Microservices und die damit einhergehende doppelte Ausführung der Datentransferobjekte wurde nach insgesamt 5 Minuten und 30 Sekunden korrekt identifiziert und erklärt.

Testperson B wurde am 28. 10. 2016 um 18:00 vor die Aufgabe zurLösung der Anforderung 1 an die Microservice-Anwendung gestellt. Die Einarbeitung dieser Person in den Quellcode nahm zu Beginn viel Zeit in Anspruch. Nach 8 Minuten und 55 Sekunden konnte die Testperson die korrekte Stelle zur Problembehebung im Quellcode identifizieren. Der gezeigte und vollständige Lösungsweg, bestehend aus dem Ansprechen von bis zu fünf Modems im Microservice, wurde nach 1 Minute und 45 Sekunden identifiziert. Darauf folgte die Herausforderung zur Lösung der Anforderung 2 in der monolithischen Anwendung. Es dauerte 6 Minuten und 54 Sekunden bis zur Identifikation der korrekten Quellcodestelle. Die Lösung der Anforderung war nach 7 Minuten und 17 Sekunden abgeschlossen. Diese Testperson nahm sich zu Beginn die Zeit, um die Anwendung zu analysieren und zu verstehen. Die Lösung der Anforderungen nahm danach im Durchschnitt 1 Minute und 4 Sekunden in Anspruch. In Relation zur erfassten Zeit 1 ein geringer Anteil.

Testperson C wurde am 4. 11. 2016 um 9:30 vormittags vor die Aufgabe zur Lösung der Anforderung 1 in der monolithischen Anwendung gestellt. Diese Aufgabenstellung wurde nach 1 Minute und 50 Sekunden vollständig gelöst, nachdem 40 Sekunden zuvor die korrekte Identifizierung der Problemstelle stattfand. Im Anschluss begann die Lösungsfindung der Anforderung 2 in der Microservice-Anwendung. Hierfür wurden 58 Sekunden bis zum Auffinden der richtigen Problemstelle benötigt. Wie die Testperson A hatte diese Testperson anfänglich Probleme, die Anforderung korrekt zu lösen. Der Lösungsweg über die Anpassung beider Datentransferobjekte war nach 4 Minuten und 43 Sekunden erfolgreich beschrieben. Gegenüber den vorangegangenen Testpersonen wählte diese die Vorgehensweise, sich direkt anhand der Module zu orientieren und die für die Anforderungslösung nicht benötigten Teile der Software nicht zu untersuchen.

Testperson D wurde am 4. 11. 2016 um 13:00 zur Analyse der beiden Softwareartefakte gebeten. Diese startete mit der Anforderung 1 in der Microservice-Anwendung und schloss die Suche der Stelle im Quellcode nach 3 Minuten und 2 Sekunden erfolgreich ab. Die Darstellung des Lösungsweges nahm in diesem Fall weitere 41 Sekunden in Anspruch. Der gewählte Lösungsweg war das Speichern von mehreren Objekten des Typs „DeliveryTC35“ in der „AlarmSender“-Klasse des gleichnamigen Moduls. In weiterer Folge wurde der Testperson die Anforderung 2 zur Lösung in der monolithischen Anwendung übergeben. Dies konnte erfolgreich gelöst werden nach einer Gesamtzeit von 3 Minuten und 24 Sekunden. Für das Auffinden der richtigen Quellcodestelle wurden 2 Minuten und 21 Sekunden benötigt. Die Testperson widmete sich ebenfalls mehr der Analyse der Anwendung, jedoch nicht im Quellcode, sondern mithilfe der zur Verfügung gestellten Diagramme der Unified Modeling Language (UML) aus den Bausteinsichten der jeweiligen Anwendungen (Abbildung 6-2 und Abbildung 6-9).

	Anforderung 1			Anforderung 2		
	Zeit 1	Zeit 2	Differenz	Zeit 1	Zeit 2	Differenz
Monolith	05:24	08:12	02:48	06:54	07:17	00:23
	01:10	01:50	00:40	03:21	03:24	00:03
Mittelwert	03:17	05:01	01:44	05:07	05:20	00:13
Microservice	08:55	10:40	01:45	01:06	05:30	04:24
	03:02	03:43	00:41	00:58	04:43	03:45
Mittelwert	05:58	07:11	01:13	01:02	05:06	04:04

Tabelle 7-5: Entwicklung der Veränderungsgeschwindigkeit

Betrachtet aus einer gesamtheitlichen Perspektive tritt in Erscheinung, dass in Bezug auf die Anforderung 1 die Lösungsgeschwindigkeit, von der Erkennung der richtigen Quellcodestelle bis zur vollständigen Darlegung des Lösungspfades, im Durchschnitt in der Microservice-Anwendung weniger Zeit in Anspruch nimmt als in der monolithischen Anwendung. Diese Erkenntnis wurde in der Anforderung 2 umgekehrt. Hier liegt der durchschnittliche Wert zur Lösungsdarlegung höher, bedingt durch den vermehrten Kommunikationsaufwand im Gesamtsystem. Gleichzeitig konnte die richtige Lösung in Summe jedoch schneller fertiggestellt werden als im monolithischen System. Die Umsetzung der Anforderung 1, welche in beiden Systemen ähnlich zu lösen war, konnte im monolithischen System schneller gelöst werden. In beiden Fällen sind Ausreißer erkennbar, wobei große Abweichungen nach unten, eine schnelle Lösung, beziehungsweise oben, eine langsame Lösung, vorhanden sind.

Als Ergebnis dieses Forschungsteiles kann die Hypothese 1, die Anwendung von Microservices beschleunigt die Adaptierung der Software durch Entwickler, weder bestätigt noch falsifiziert werden. Aus dem Betrachtungswinkel der Änderungsgeschwindigkeit, gleichbedeutend mit dem Programmieraufwand, sobald der/die EntwicklerIn im System eingearbeitet ist, können die Microservices mit der Anforderung 1 punkten. An den Ergebnissen aus der Tabelle 7-5, in der Spalte Anforderung 2, lässt sich jedoch auch ablesen, dass der Programmieraufwand bei Microservice übergreifenden Lösungen, im Vergleich zur monolithischen Anwendung, mehr Zeit in Anspruch nimmt.

7.3 Interviews

Zur Analyse der Interviews wird das Verfahren nach Mayring (Bortz & Döring, 2006) angewendet. Die Inhaltsanalyse mit induktiver Kategorienbildung ergibt sieben Kategorien und ist im Detail im Anhang C dargestellt. Die befragten Personen aus dem Bereich der Softwareentwicklung weisen eine große Bandbreite an Vorwissen zu den Themen Microservices und Codequalität auf. Dies lässt sich belegen durch die Angaben der Interviewten zur Frage der bereits gesammelten Erfahrung mit Microservices. Die Antworten reichen von Theoriekenntnissen über den Einsatz der Architektur in kleinen Umgebungen bis

zum produktiven Einsatz in der Produktentwicklung. Eine Person beschreibt den Einsatz des Musters, jedoch in einer entarteten Form. Das bedeutet, dass die Anwendung in kleine Teilbereiche gegliedert wurde, aber die Trennung nicht nach den Vorgaben des Domain Driven Design geschehen ist. Zum Einsatz in der Produktentwicklung gibt es zwei verschiedene Ansichten unter den Interviewten. Einerseits wurden die Vorteile bei der Anbindung von externen Diensten hervorgehoben, andererseits wurde der Vorteil der Unabhängigkeit als Nachteil der Wartung und Instandhaltung ausgelegt. Der Verwaltungsaufwand würde durch den Einsatz in nicht rentable Höhen steigen.

Die Codequalität wird von allen Befragten als wichtig hervorgehoben, auch wenn deren Einsatz bei den aktuellen ArbeitgeberInnen nicht immer forciert wird. Zwei der Interviewten spannten eine Verbindung der Codequalität zur Softwarequalität und in weiterer Folge hin zur grundlegenden Architektur einer Softwareanwendung. Sie beschreiben die Softwarequalität als Basis für die Architekturqualität. Des Weiteren soll eine korrekt implementierte Architektur die Codequalität fördern. Der Terminus Codequalität wurde durch die befragten Personen in die Kategorien Metriken, Codereviews, Syntax, Lesbarkeit, Dokumentation und Testabdeckung verfeinert. Hier wurde angeführt, dass zum Beispiel in der aktuellen Arbeitsumgebung automatisiert, mittels Softwareunterstützung, auf Metriken in der Softwareentwicklung geachtet wird. Dasselbe Unternehmen setzt zudem Codereviews zur Erhaltung der Qualität ein. Auf der anderen Seite versuchen die EntwicklerInnen in Unternehmen ohne Fokus auf Codequalität selbstständig die Lesbarkeit zu erhöhen oder die Dokumentation zu erstellen.

Zum Abschluss der Interviews wurde die Hauptfrage zur Falsifizierung der Hypothese 3, „Microservices erhöhen die Codequalität in Form von Lesbarkeit in einer Softwareanwendung“, gestellt. Alle teilnehmenden Personen bestätigen eine Verbesserung der Lesbarkeit hauptsächlich durch die Trennung der Codeteile anhand der Vorgaben des Domain Driven Design. Die Interviewpartner 1 und 2 führten zur Untermauerung die Wartbarkeit ins Feld. Sie definierten die Lesbarkeit als Merkmal zur schnellen Einarbeitung in den bestehenden Quellcode und der damit verbundenen geringen Anpassungszeit zum Verständnis des Quellcodes. Durch die vier befragten Personen kann somit die Hypothese 3 bestätigt werden und einer Erhöhung der Lesbarkeit durch den Einsatz von Microservices zugestimmt werden.

7.4 Zusammenfassung

Zusammenfassend konnten alle aufgestellten Hypothesen durch die Wahl des Forschungsdesigns überprüft werden. Die Daten der Erhebung der Änderungsgeschwindigkeit ließen keine Bestätigung oder Falsifizierung der Hypothese 1 zu. Die Ergebnisse der Umsetzung aus der Anforderung 2 würden die Ergebnisse aus der Erhebung der Anforderung 1 unterstützen, diese lassen jedoch keine eindeutige Beantwortung der Hypothese zu, denn der Umfang zur vollständigen Lösung der Anforderung 2 ist im Umfeld der Microservices höher als in der monolithischen Ausgangsanwendung. Somit sind die Ergebnisse dieser Untersuchung plausibel und

sprechen für die Qualität der Erhebung. Im Umfang der Anforderung 1 unterscheiden sich die beiden Anwendungen jedoch kaum und die Ergebnisse der Lösungsgeschwindigkeit sind im Mittelwert nicht aussagekräftig genug, um die Hypothese 1 als Gesamtes zu bestätigen oder zu falsifizieren.

Hingegen konnten für die Bestätigung der Hypothese 2 Indizien gefunden werden. Die Komplexität der Zielsoftware konnte im Vergleich zur Ausgangsanwendung verringert werden. Dies wurde erhoben durch zwei verschiedene Tools für Softwaremetriken und ist positiv unterstützt durch die Metriken der Wartbarkeit sowie der Vererbungstiefe. Beide Metriken weisen positive Veränderungen von der monolithischen Anwendung zu den Microservices auf. Diese stellen für sich jedoch keinen Anspruch auf eine signifikante Beeinflussung des Codequalitätsergebnisses durch Microservices. Gegenüber den positiven Veränderungen stehen zwei negative Veränderungen. Die Metrik der Lines of Code sowie die Metrik der Klassenkopplung haben sich in geringem Umfang verschlechtert. Damit halten sich, abgesehen von der Komplexität, die erhobenen Metriken die Waage, um in Summe als Indiz für die positive Veränderung der Codequalität im Bereich der Komplexität anerkannt zu werden.

Schließlich wurde durch die Interviews die Hypothese 3 zur Verbesserung der Lesbarkeit bestätigt. Alle befragten Personen, welche aus verschiedenen Branchen und Unternehmen mit unterschiedlicher Größe stammen, kamen nach der Analyse der beiden Anwendungen zu dem Schluss, dass sich die Lesbarkeit des Quellcodes nach der Umwandlung zu Microservices erhöht hat. Die Erhebung der Codequalität in den Interviews unterstützt die Metrikerhebung aus dem praktischen Forschungsteil. Ebenso wurde von den befragten Personen eine erhöhte Wartbarkeit der Microservices genannt. Dies ist im praktischen Forschungsteil in der Metrik „Wartbarkeitsindex“ ebenfalls abzulesen. Dagegen wurde die Größe des Quellcodes durch die Testpersonen anders wahrgenommen als dies die Ergebnisse der Erhebung zeigen.

Im Zusammenhang kann für den gewählten Anwendungsfall die Forschungsfrage „Wie wirkt sich der Einsatz des Architekturentwurfes Microservices auf die Codequalität aus?“, wie folgt beantwortet werden: Die Codequalität kann in den Bereichen Komplexität und Lesbarkeit durch den Einsatz der Softwarearchitektur Microservices verbessert werden.

8 DISKUSSION

Die reflektierten Erkenntnisse aus der Masterarbeit sowie eine kritische Würdigung der Methodenwahl und der Ergebnisse sind in diesem Kapitel dargestellt.

8.1 Methode

Wie ein Teil der interviewten Personen bestätigt, ist das Vorgehen der Metrikerhebung, im industriellen Umfeld der Softwareentwicklung, eine gängige Methode zur Erhebung von Codequalität. Dazu wurde versucht, durch den Einsatz weiterer Erhebungen die Varianz der Ergebnisse zu erhöhen, damit sich diese gegenseitig auch bei einfacher Ausprägung unterstützen und somit ein schlüssiges Gesamtfazit gezogen werden kann. Die Aussagekraft dieses Fazits wird geschmälert durch die geringe Anzahl der SoftwareentwicklerInnen, wodurch diese Erhebung als Laborexperiment tituliert werden muss. Weiterführende Forschung sollte im Feld betrieben werden.

8.2 Ergebnisse

Aus Sicht der Ergebnisqualität stellt sich die Berechnung der Durchschnittswerte als problematisch dar. Diese unterstützen das Ergebnis, da in den bearbeiteten Modulen von Monolith zu Microservices jeweils eine Verbesserung erzielt werden konnte. Durch die zusätzliche Einführung von weiteren Modulen wird jedoch der Durchschnittswert gedrückt und somit ein noch signifikanteres Ergebnis erzielt. Die Berechnungen der Metriken in den vorhandenen und eingesetzten Tools setzen jedoch ebenfalls auf den Durchschnitt als Gesamtergebnis aus mehreren Modulen und wurden vom Autor deshalb als valide eingestuft.

Allgemein wurden die Ergebnisse dieser Masterarbeit nach den wissenschaftlichen Grundsätzen erhoben, sind jedoch nur unter der Einschränkung der Anwendungsgröße für die Allgemeinheit anwendbar. Durch den Umfang von ungefähr 12.000 Zeilen Quellcode muss diese Einschränkung getroffen werden.

8.3 Ausblick

Die Beantwortung der Forschungsfrage und der damit verbundenen Bestätigung der Veränderung der Codequalität durch den Einsatz von Microservices konnte in dieser Arbeit in eingeschränkter Form erbracht werden. Die gewonnenen Erkenntnisse lassen durch die Größe der Untersuchung nur unter Einschränkungen einen Rückschluss auf eine allgemeine

Anwendbarkeit für alle Microservice-Anwendungen zu. Dazu bedarf es einer weiterführenden Forschung mit Großanwendungen, um zum Beispiel die Aussagekraft für Anwendungen, wie Amazon’s Webshop oder den Videodienst Netflix, bestätigen zu können. Hierfür bietet diese Masterarbeit einen Anknüpfungspunkt.

ANHANG A - Interviewleitfaden

Aufgabenstellung

Programmierung

InterviewpartnerIn Nr.: ____

	Anforderung 1	Anforderung 2	Lösungsweg	Zeit 1	Zeit 2
Monolith					
Microservices					

Zeit 1: Zeit bis zum Erkennen der richtigen Stelle im Programm.

Zeit 2: Zeit bis zur korrekten Darlegung der benötigten Schritte zur Erfüllung der Anforderung.

Interviewleitfaden

Einleitung

Nachdem Sie den praktischen Teil des Interviews nun bereits abgeschlossen haben, würde ich noch gerne vertiefende Fragen zu den Themen Microservices und Codequalität stellen.

Allgemeines

(1) Wie ist es Ihnen mit den Quellcodebeispielen ergangen?

Microservices

(2) Haben Sie bereits Erfahrungen mit Microservices gesammelt?

Codequalität

(3) Wie stehen Sie zu Codequalität?

(4) Wird in Ihrer aktuellen Arbeitsumgebung auf Codequalität geachtet?

Verbindung

(5) Wurde im Vergleich Monolith zu Microservices die Lesbarkeit des Quellcodes verändert?

Statistik

(6) Wie alt sind Sie,

(7) in welcher Position sind Sie im Unternehmen tätig,

(8) wie viele Mitarbeiter hat das Unternehmen, in dem Sie tätig sind?

Verabschiedung

Herzlichen Dank für die Teilnahme an meiner Untersuchung. Wenn Sie es wünschen, werde ich Sie über das Ergebnis informieren.

ANHANG B - Metriken auf Klassen- und Dateiebene

Visual Studio Modul	Typ	Lines of Code		Zyklomatische Komplexität	
		Monolith	Micro-services	Monolith	Micro-services
AlarmLink	AlarmLinkInTouchDB	82	41	24	10
	AlarmLinkInTouchDBConfi	22	22	7	7
	frmEditInTouchDBConfi	168	160	14	14
	TableAlarmSource	103	103	64	64
	TableModule	46	46	28	28
	ViewLastEvent	48	48	30	30
	ViewWWAlmDB	96		62	
	ViewWWAlmDB_LastEvent	24		14	
	ServiceAlarm		27		26
	ServiceClient		14		3
	ServiceStatus		3		3
AlarmLinkOpc	Alarm	134	134	72	72
	AlarmItemValues	27	27	19	19
	AlarmLinkOpc	234	234	83	83
	AlarmLinkOpcDBConfi	181	181	31	31
	AlarmSource	126	126	62	62
	AvsDbClassesDataContext	31	31	17	17
	ConfigObject	16	16	15	15
	frmDBConfiguration	136	136	13	13
	frmOPCTagSelection	208	208	15	15
	frmPrintPreview	45	45	7	7
	Group	100	100	49	49
	Item	141	141	69	69
	Module	62	62	30	30
	OpcAlarm	104	104	49	49
	Server	76	76	39	39
	Variable	56	56	30	30
AlarmSender	AlarmSender	46	46	17	17
	DBAVSDataContext	19	19	11	11
	Defines	3	3	3	3
	DeliveryMail	144	144	33	33
	DeliveryMain	12	12	6	6
	DeliveryTC35	122	126	42	39
	frmDeliveryMailConfig	210	210	14	14
	frmDeliveryTC35Config	132	82	19	10
	frmDeliveryTC35TestSMS	78	78	11	11

	SMSReceive	56	56	29	29
	vwAddresseePropertyExtended	43	43	32	32
	TableAlarm	106	106	69	69
	TableAlarmToSend	56	56	35	35
	TableDelivery	54	54	34	34
	ViewAlarmToSend	136	136	89	89
AtecMail	AtecMailMessage	61	61	21	21
AtecTC35	AtecTC35	173		60	
	AtecTC35Exception	12		8	
	clsSMS	135		48	
AtecTemplateParser	DoubleLinkedList	51	51	20	20
	DoubleLinkedList.Node	24	24	12	12
	Example	34	34	2	2
	IIterator	0	0	3	3
	ITemplate	0	0	2	2
	ITmplBlock	0	0	4	4
	ITmplLoader	0	0	1	1
	TemplatePool	72	72	23	23
	TmplParser	180	180	66	66
	TmplParser.BlockBound	38	38	15	15
	TmplParser.BlockParser	94	94	47	47
	TmplParser.TagBound	16	16	7	7
AVS.NETService	AvsNetService	67	67	24	24
	Program	5	5	2	2
	AccessFlag	52	52	27	27
	Addressee	100	100	49	49
	AddresseeProperty	106	106	50	50
	Alarm	142	142	76	76
	AlarmSource	126	126	62	62
	AlarmToSend	104	104	52	52
	AvsNetDataContext	105	105	49	49
	ColumnName	64	64	33	33
	ConditionGroup	64	64	33	33
	Delivery	70	70	36	36
	ErrorLog	29	29	22	22
	Filter	180	180	86	86
	Group	84	84	41	41
	GroupFilter	80	80	37	37
	Module	58	58	30	30
	OpcAlarms	104	104	49	49
	OpcGroup	82	82	42	42
	OpcItem	159	159	76	76

	OpcServer	68	68	35	35
	Operator	64	64	33	33
	ShiftSchedule	60	60	31	31
	ShiftScheduleProperty	80	80	39	39
	SMSReceive	56	56	29	29
	spFrequency_Ergebnis	29	29	24	24
	Variable	56	56	30	30
	VisFilter	110	110	56	56
	vwAddresseePropertyDelivery	29	29	22	22
	vwAddresseePropertyExtended	41	41	31	31
	vwAlarmChronologic	69	69	63	63
	vwAlarmSendReport	73	73	61	61
	vwAlarmToSendAddresseeProperty	81	81	70	70
	vwGroupFilter	61	61	49	49
	vwLastAlarmEvent	21	21	22	22
AVSClient	Loading	80	80	11	11
	frmMainWindow	353	353	64	64
	Program	5	5	5	5
	AddresseeManagement	24	24	7	7
	FormDeliveryData	64	64	12	12
	FormEditAddressee	282	282	27	27
	FormEditGroup	262	262	39	39
	FormInsertTimespan	94	94	19	19
	FormShiftSchedule	142	142	17	17
	GroupManagement	26	26	11	11
	ShiftScheduleManagement	20	20	7	7
	AVSDataClassesDataContext	8	8	7	7
	Module	50	50	26	26
	TableAddressee	89	89	52	52
	TableAddresseeProperty	55	55	34	34
	TableColumnName	53	53	34	34
	TableConditionGroup	53	53	33	33
	TableDelivery	50	50	31	31
	TableFilter	97	97	62	62
	TableGroup	50	50	30	30
	TableGroupFilter	36	36	22	22
	TableOperator	51	51	32	32
	TableShift	42	42	25	25
	TableShiftScheduleProperty	48	48	28	28
	ViewAddresseePropertyDelivery	56	56	35	35
	ViewGroupFilter	114	114	73	73
	ViewGroupFilterDelivery	80	80	41	41

	FilterManagement	46	46	20	20
	FormEditFilter	346	346	45	45
AVSClientViewer	TableAlarmSource	103	103	64	64
	TableModule	44	44	27	27
	AVSClientViewerCtrl	1156	1156	148	148
	frmPrintPreview	36	36	11	11
	PrintReport	148	148	35	35
	PrintReportCtrl	148	148	44	44
	Alarm	116	116	65	65
	AVSDataClassesDataContext	8	8	7	7
	SPAlarmFrequency	73	73	47	47
	TableAlarmChronologic	157	157	104	104
	ViewAlarmSendReport	151	151	104	104
	VisFilter	131	131	76	76
Atec.AVS.IntouchDB .Service	Program		15		14
	ServiceConfiguration		13		10
	Startup		4		2
	ViewWWAlmDB		96		62
	ViewWWAlmDB_LastEvent		24		14
	Alarm		27		26
	Status		5		5
	IntouchDBController		33		5
Atec.SMS.Service	ISMSService		0		4
	ISMSServiceStatus		0		3
	AtecTC35		142		46
	AtecTC35Exception		12		8
	clsSMS		165		54
	Program		39		29
	ShortMessage		13		13
	ShortMessageCollection		1		1
	ISendRequest		0		2
	ISendResponse		0		1
	SendRequest		5		5
	SendResponse		3		3
	Status		7		7
	RESTService		34		15

Tabelle B-1: Detailergebnisse der Metriken Lines of Code und Zykomatische Komplexität aus dem Tool Visual Studio

sonarqube		Lines of Code		Zyklomatische Komplexität	
Modul	Typ	Monolith	Micro-services	Monolith	Micro-services
AlarmLink	AlarmLinkInTouchDB	172	176	24	26
	AlarmLinkInTouchDBConfi	59	59	7	7
	frmEditInTouchDBConfi	75	75	10	10
	TableAlarmSource	238	238	78	78
	TableModule	121	121	32	32
	ViewLastEvent	125	125	35	35
	ViewWWAlmDB	221		75	
	ViewWWAlmDB_LastEvent	77		15	
	ServiceAlarm		26		1
	ServiceClient		42		7
	ServiceStatus		12		0
AlarmLinkOpc	AlarmLinkOpc	394	394	68	68
	AlarmLinkOpcDBConfi	162	162	18	18
	AvsDbClassesDataContext	11	11	0	0
	frmDBConfiguration	74	74	6	6
	frmOPCTagSelection	64	64	7	7
	frmPrintPreview	36	36	3	3
AlarmSender	AlarmSender	89	89	18	18
	DBAVSDataContext	10	10	1	1
	Defines	11	11	2	2
	DeliveryMail	257	257	40	40
	DeliveryMain	35	35	6	6
	DeliveryTC35	218	235	52	44
	frmDeliveryMailConfig	70	70	10	10
	frmDeliveryTC35Config	77	50	15	6
	frmDeliveryTC35TestSMS	43	43	7	7
	TableAlarm	242	242	84	84
	TableAlarmToSend	141	141	41	41
	TableDelivery	137	137	40	40
	ViewAlarmToSend	301	301	109	109
AtecMail	AtecMailMessage	124	124	21	21
AtecTC35	AtecTC35	313		79	
	AtecTC35Exception	37		8	
	clsSMS	257		40	
AtecTemplateParser	DoubleLinkedList	188	188	38	38
	Example	45	45	1	1
	TemplatePool	124	124	25	25
	TmplParser	599	599	145	145
AVS.NETService	AvsNetService	143	143	16	16

	Program	27	27	1	1
AVSClient	Loading	59	59	7	7
	frmMainWindow	306	306	51	51
	Program	29	29	2	2
	AddresseeManagement	54	54	8	8
	FormDeliveryData	47	47	8	8
	FormEditAddressee	150	150	23	23
	FormEditGroup	217	217	35	35
	FormInsertTimespan	97	97	15	15
	FormShiftSchedule	103	103	13	13
	GroupManagement	67	67	13	13
	ShiftScheduleManagement	53	53	7	7
	TableAddressee	195	195	59	59
	TableAddresseeProperty	139	139	40	40
	TableColumnName	141	141	39	39
	TableConditionGroup	137	137	38	38
	TableDelivery	129	129	36	36
	TableFilter	223	223	71	71
	TableGroup	130	130	34	34
	TableGroupFilter	101	101	25	25
	TableOperator	133	133	37	37
	TableShift	113	113	28	28
	TableShiftScheduleProperty	132	132	34	34
	ViewAddresseePropertyDelivery	141	141	41	41
	ViewGroupFilter	255	255	90	90
	ViewGroupFilterDelivery	181	181	51	51
	FilterManagement	92	92	17	17
	FormEditFilter	202	202	34	34
AVSClientViewer	TableAlarmSource	240	240	77	77
	TableModule	117	117	31	31
	AVSClientViewerCtrl	724	724	140	140
	frmPrintPreview	43	43	7	7
	PrintReport	158	158	30	30
	PrintReportCtrl	243	243	37	37
	SPAlarmFrequency	184	184	63	63
	TableAlarmChronologic	338	338	146	146
	ViewAlarmSendReport	343	343	144	144
	VisFilter	288	288	91	91
Atec.AVS.IntouchDB .Service	Program		39		3
	ServiceConfiguration		43		9
	Startup		21		1
	ViewWWAlmDB		221		75

	ViewWWAlmDB_LastEvent		77		15
	Alarm		26		1
	Status		13		0
	IntouchDBController		56		5
Atec.SMS.Service	ISMSService		28		0
	ISMSServiceStatus		14		0
	AtecTC35		271		55
	AtecTC35Exception		37		8
	clsSMS		304		45
	Program		82		17
	ShortMessage		50		12
	ISendRequest		14		0
	ISendResponse		12		0
	SendRequest		22		0
	SendResponse		13		0
	Status		27		0
	RESTService		81		20

Tabelle B-2: Detailergebnisse der Metriken Lines of Code und Zykomatische Komplexität aus dem Tool sonarqube

Visual Studio Modul	Typ	Wartbarkeitsindex		Vererbungstiefe		Klassenkopplung	
		Monolith	Microservices	Monolith	Microservices	Monolith	Microservices
AlarmLink	AlarmLinkInTouchDB	49	60	2	2	16	18
	AlarmLinkInTouchDBConfi	72	72	8	8	10	10
	frmEditInTouchDBConfi	53	54	8	8	29	27
	TableAlarmSource	80	80	3	3	6	6
	TableAlarmSource.COLUMN	100	100	1	1	0	0
	TableModule	81	81	3	3	6	6
	TableModule.COLUMN	100	100	1	1	0	0
	ViewLastEvent	81	81	3	3	6	6
	ViewLastEvent.COLUMN	100	100	1	1	0	0
	ViewWWAlmDB	80		3		6	
	ViewWWAlmDB.COLUMN	100		1		0	
	ViewWWAlmDB_LastEvent	82		3		6	
	ViewWWAlmDB_LastEvent.COLUMN	100		1		0	
	ServiceAlarm		91		1		1
	ServiceClient		73		1		15
	ServiceStatus		95		1		1
AlarmLinkOpc	Alarm	77	77	1	1	17	17
	AlarmItemValues	91	91	1	1	2	2
	AlarmLinkOpc	45	45	2	2	49	49
	AlarmLinkOpcDBConfi	53	53	8	8	69	69
	AlarmSource	78	78	1	1	19	19
	AvsDbClassesDataContext	88	88	2	2	21	21
	ConfigObject	93	93	1	1	0	0
	frmDBConfiguration	50	50	8	8	46	46
	frmOPCTagSelection	51	51	7	7	53	53
	frmPrintPreview	63	63	7	7	22	22
	Group	78	78	1	1	18	18

	Item	78	78	1	1	17	17
	Module	78	78	1	1	15	15
	OpcAlarm	74	74	1	1	17	17
	Server	82	82	1	1	15	15
	Variable	78	78	1	1	12	12
AlarmSender	AlarmSender	50	50	1	1	10	10
	DBAVSDataContext	88	88	2	2	15	15
	Defines	96	96	1	1	0	0
	DeliveryMail	50	50	2	2	24	24
	DeliveryMain	78	78	1	1	2	2
	DeliveryTC35	57	56	2	2	21	42
	frmDeliveryMailConfig	47	47	8	8	32	32
	frmDeliveryTC35Config	53	57	8	8	35	26
	frmDeliveryTC35TestSMS	55	55	8	8	39	39
	SMSReceive	79	79	1	1	11	11
	vwAddresseePropertyExtended	90	90	1	1	2	2
	TableAlarm	79	79	3	3	5	5
	TableAlarm.COLUMN	100	100	1	1	0	0
	TableAlarmToSend	80	80	3	3	6	6
	TableAlarmToSend.COLUMN	100	100	1	1	0	0
	TableDelivery	80	80	3	3	5	5
	TableDelivery.COLUMN	100	100	1	1	0	0
	ViewAlarmToSend	79	79	3	3	5	5
	ViewAlarmToSend.COLUMN	100	100	1	1	0	0
AtecMail	AtecMailMessage	70	70	1	1	8	8
AtecTC35	AtecTC35	58		1		10	
	AtecTC35Exception	90		1		0	
	clsSMS	60		1		11	
AtecTemplateParser	DoubleLinkedList	77	77	1	1	2	2
	DoubleLinkedList.Node	85	85	1	1	1	1

	Example	55	55	1	1	4	4
	IIterator	100	100	0	0	0	0
	ITemplate	100	100	0	0	1	1
	ITmplBlock	100	100	0	0	0	0
	ITmplLoader	100	100	0	0	1	1
	TemplatePool	66	66	1	1	9	9
	TmplParser	69	69	1	1	9	9
	TmplParser.BlockBound	78	78	1	1	0	0
	TmplParser.BlockParser	76	76	1	1	2	2
	TmplParser.TagBound	84	84	1	1	0	0
AVS.NETService	AvsNetService	63	63	4	4	18	18
	Program	71	71	1	1	4	4
	AccessFlag	82	82	1	1	15	15
	Addressee	78	78	1	1	18	18
	AddresseeProperty	77	77	1	1	19	19
	Alarm	77	77	1	1	19	19
	AlarmSource	78	78	1	1	19	19
	AlarmToSend	74	74	1	1	18	18
	AvsNetDataContext	81	81	2	2	46	46
	ColumnName	82	82	1	1	14	14
	ConditionGroup	83	83	1	1	14	14
	Delivery	82	82	1	1	15	15
	ErrorLog	91	91	1	1	3	3
	Filter	76	76	1	1	20	20
	Group	78	78	1	1	19	19
	GroupFilter	74	74	1	1	16	16
	Module	82	82	1	1	15	15
	OpcAlarms	74	74	1	1	17	17
	OpcGroup	82	82	1	1	15	15
	OpcItem	77	77	1	1	18	18

	OpcServer	78	78	1	1	11	11
	Operator	83	83	1	1	14	14
	ShiftSchedule	82	82	1	1	16	16
	ShiftScheduleProperty	78	78	1	1	16	16
	SMSReceive	79	79	1	1	11	11
	spFrequency_Ergebnis	90	90	1	1	2	2
	Variable	78	78	1	1	12	12
	VisFilter	78	78	1	1	11	11
	vwAddresseePropertyDelivery	91	91	1	1	3	3
	vwAddresseePropertyExtended	91	91	1	1	2	2
	vwAlarmChronologic	89	89	1	1	4	4
	vwAlarmSendReport	89	89	1	1	4	4
	vwAlarmToSendAddresseeProperty	89	89	1	1	4	4
	vwGroupFilter	90	90	1	1	3	3
	vwLastAlarmEvent	89	89	1	1	4	4
AVSClient	Loading	61	61	7	7	26	26
	frmMainWindow	56	56	7	7	82	82
	Program	93	93	1	1	3	3
	AddresseeManagement	73	73	8	8	9	9
	FormDeliveryData	66	66	8	8	23	23
	FormEditAddressee	49	49	8	8	41	41
	FormEditGroup	50	50	8	8	52	52
	FormInsertTimespan	69	69	8	8	21	21
	FormShiftSchedule	55	55	8	8	35	35
	GroupManagement	72	72	8	8	13	13
	ShiftScheduleManagement	76	76	8	8	10	10
	AVSDataClassesDataContext	91	91	2	2	8	8
	Module	79	79	1	1	11	11
	TableAddressee	80	80	3	3	8	8
	TableAddressee.COLUMN	100	100	1	1	0	0

	TableAddresseeProperty	80	80	3	3	5	5
	TableAddresseeProperty.COLUMN	100	100	1	1	0	0
	TableColumnName	81	81	3	3	5	5
	TableColumnName.COLUMN	100	100	1	1	0	0
	TableConditionGroup	81	81	3	3	5	5
	TableConditionGroup.COLUMN	100	100	1	1	0	0
	TableDelivery	81	81	3	3	6	6
	TableDelivery.COLUMN	100	100	1	1	0	0
	TableFilter	80	80	3	3	7	7
	TableFilter.COLUMN	100	100	1	1	0	0
	TableGroup	81	81	3	3	6	6
	TableGroup.COLUMN	100	100	1	1	0	0
	TableGroupFilter	81	81	3	3	5	5
	TableGroupFilter.COLUMN	100	100	1	1	0	0
	TableOperator	81	81	3	3	5	5
	TableOperator.COLUMN	100	100	1	1	0	0
	TableShift	81	81	3	3	6	6
	TableShift.COLUMN	100	100	1	1	0	0
	TableShiftScheduleProperty	82	82	3	3	5	5
	TableShiftScheduleProperty.COLUMN	100	100	1	1	0	0
	ViewAddresseePropertyDelivery	80	80	3	3	6	6
	ViewAddresseePropertyDelivery.COLUMN	100	100	1	1	0	0
	ViewGroupFilter	79	79	3	3	5	5
	ViewGroupFilter.COLUMN	100	100	1	1	0	0
	ViewGroupFilterDelivery	76	76	3	3	3	3
	ViewGroupFilterDelivery.COLUMN	100	100	1	1	0	0
	FilterManagement	69	69	8	8	16	16
	FormEditFilter	51	51	8	8	42	42
AVSClientViewer	TableAlarmSource	80	80	3	3	6	6
	TableAlarmSource.COLUMN	100	100	1	1	0	0

	TableModule	81	81	3	3	5	5
	TableModule.COLUMN	100	100	1	1	0	0
	AVSClientViewerCtrl	48	48	7	7	107	107
	AVSClientViewerCtrl.LoadFunction	100	100	1	1	2	2
	AVSClientViewerCtrl.LoadFunctionFinished	100	100	1	1	2	2
	frmPrintPreview	73	73	7	7	20	20
	PrintReport	61	61	7	7	43	43
	PrintReportCtrl	62	62	7	7	38	38
	PrintReportCtrl.PrintFinished	100	100	1	1	2	2
	Alarm	78	78	1	1	12	12
	AVSDataClassesDataContext	91	91	2	2	8	8
	SPAlarmFrequency	75	75	3	3	6	6
	SPAlarmFrequency.COLUMN	100	100	1	1	0	0
	TableAlarmChronologic	75	75	3	3	7	7
	TableAlarmChronologic.COLUMN	100	100	1	1	0	0
	ViewAlarmSendReport	74	74	3	3	6	6
	ViewAlarmSendReport.COLUMN	100	100	1	1	0	0
	VisFilter	74	74	3	3	6	6
	VisFilter.COLUMN	100	100	1	1	0	0
Atec.AVS.IntouchDB.Service	Program		75		1		12
	ServiceConfiguration		91		1		2
	Startup		82		1		6
	ViewWWAlmDB		80		3		6
	ViewWWAlmDB.COLUMN		100		1		0
	ViewWWAlmDB_LastEvent		82		3		6
	ViewWWAlmDB_LastEvent.COLUMN		100		1		0
	Alarm		91		1		1
	Status		94		1		1
	IntouchDBController		59		2		14
Atec.SMS.Service	ISMSService		100		0		6

	ISMSServiceStatus		100		0		0
	AtecTC35		62		1		9
	AtecTC35Exception		90		1		0
	clsSMS		59		1		19
	Program		80		1		16
	ShortMessage		93		1		0
	ShortMessageCollection		100		2		2
	ISendRequest		100		0		0
	ISendResponse		100		0		0
	SendRequest		94		1		1
	SendResponse		95		1		1
	Status		94		1		1
	RESTService		70		1		8

Tabelle B-3: Detailergebnisse der Metriken Wartbarkeitsindex, Vererbungstiefe und Klassenkopplung aus dem Tool Visual Studio

ANHANG C - Kategorienbildung

Kategorie Erfahrung mit Microservices

Kategorienkürzel	Kategorienname	Ankerbeispiel
EM1	Theorie	„... haben wir gebaut in allen varianten ...“
EM2	Kein Praxiseinsatz	„... ich habe damit noch nichts größeres gebaut ...“
EM3	Praxiseinsatz	„... der tarifkalkulator funktioniert mit micro-services ...“
EM4	Dienste	„... ein dienst hat bei uns oft mehrere micro-services ...“
EM5	Keine	„... noch gar nicht ...“

Tabelle C-1: Kategorie Erfahrung mit Microservices

Kategorie Einsatzgebiet von Microservices

Kategorienkürzel	Kategorienname	Ankerbeispiel
EG1	Dienste	„... ich finde es gut sachen wie webservices wie zum beispiel wetterdienste oder sowas ansprechen kannst ...“
EG2	Produkt	„... dienste auslagern würd ich bei einem produkt nicht machen weil dann hätte man zu viele systeme ...“

Tabelle C-2: Kategorie Einsatzgebiet von Microservices

Kategorie Architektur

Kategorienkürzel	Kategorienname	Ankerbeispiel
A1	Architekturqualtität	„... ist das um und auf ... im hinblick auf architekturqualität ...“
A2	Architektur	„... codequalität im sinne einer durchgezogenen architektur ist immer gut ...“

Tabelle C-3: Kategorie Architektur

Kategorie Codequalität

Kategorienkürzel	Kategorienname	Ankerbeispiel
CQ1	Metriken	„... es gibt metriken wir nutzen ein tool das heißt sonarqube und da gibt es glaube ich tausende regeln die man adjustieren kann wie man sie braucht ...“
CQ2	Codereviews	„... das andere thema der codqualität sind codereviews ... zumindest im softwarebereich ...“
CQ3	Syntax	„... bezogen auf syntax und so weiter ... also wir haben zwei sachen gehabt wo ein gewisser coding standard gegeben war ...“
CQ4	Lesbarkeit	„... ich schaue persönlich auch dass ich alles so leserlich wie möglich schreibe ...“
CQ5	Dokumentation	„... wie es mit den kommentaren aussieht beziehungsweise auch ob guidelines eingehalten wurden ...“
CQ6	Testabdeckung	„... die testabdeckung selbst ist natürlich ein wichtiges kriterium ...“

Tabelle C-4: Kategorie Codequalität

Kategorie Codequalitätsveränderung

Kategorienkürzel	Kategorienname	Ankerbeispiel
QV1	Trennung	„... hat sich auf jeden fall geändert weil die trennung ist halt strikter ...“
QV2	Ersetzbarkeit	„... wenn ich den intouch durch etwas anders austausche habe ich überhaupt nicht auch wenn ich den code nicht kenne kein schlechtes gewissen wenn ich den service einfach wegreiße und einfach nur die schnittstelle einhalte ...“
QV3	Wartbarkeit	„...der eindruck der wartbarkeit war bei den microservices besser ...“
		„... die grenze außen rundherum ist das ende der fachlichen komponente ... von dem ist es ein stück wartbarer für mich weil die zuordnung leichter erkennbar ist was wohin gehört“

QV4	Zeit	„... die zeit die ich am anfang investieren muss zum zurechfinden spare ich mir in weiterer folge ein ...“
QV5	Größe	„... es ist leichter einzugrenzen ... man tut sich leichter wenn man jeden bereich extra eingrenzen kann man findet so schneller zur ursache ...“

Tabelle C-5: Kategorie Qualitätsveränderung

Kategorie Lesbarkeit

Kategorienkürzel	Kategorienname	Ankerbeispiel
L1	Trennung	„... ja auf jeden fall die lesbarkeit ist besser aufgesplittet ... weil die trennung sauberer ist ...“
L2	Einarbeitung	„... am anfang tue ich mir immer schwer damit ...“
L3	Zusammenspiel	„... wenn ich die struktur einmal begriffen habe wie das zusammenspiel ist weiß ich auch wo ich im nächsten schritt hingehen muss ...“
L4	Troubleshooting	„... die lesbarkeit insgesamt im bezug auf troubleshooting finde ich besser ...“

Tabelle C-6: Kategorie Lesbarkeit

ABKÜRZUNGSVERZEICHNIS

Abb.	Abbildung
ADO	ActiveX Data Objects
API	Application Programming Interface
AVS	Alarmverwaltungssystem
COM	Component Object Model
CRM	Customer Relationship Management
CTO	Chief Technology Officer
DA	Data Access
DB	Datenbank
DCOM	Distributed Component Object Model
DDD	Domain Driven Design
DI	Diplomingenieur
DIP	Dependency Inversion Principle
DIT	Depth of Inheritance Tree
FH	Fachhochschule
GUI	Graphical User Interface
GQM	Goal-Question-Metric
HTML	Hypertext Markup Language
HTTP	Hypertext Transfer Protocol
IEC	International Electrotechnical Commission
IL	Intermediate Language
ISO	International Organization for Standardization
ISP	Interface Segregation Principle
IT	Informationstechnologie
JSON	JavaScript Object Notation
LOCM	Lack of Cohesion in Methods
LSP	Liskov Substition Principle
MI	Maintainability Index
MSc.	Master of Science
NOS	Number of Statements
OCP	Open Close Principle
OPC	Ole for Process Control
OWIN	Open Web Interface for .NET
PAAS	Platform as a Service
PIN	Persönliche Identifikationsnummer
PPS	Production Planning System
REST	Representational State Transfer
RFC	Request for Comments

RPC	Remote Procedure Call
SCS	Self-Contained Systems
SIM	Subscriber Identity Module
SMS	Short Message Service
SMTP	Simple Mail Transfer Protocol
SOA	Serviceorientierte Architektur
SOAP	Simple Object Access Protocol
SRP	Single Responsibility Principle
UI	User Interface
UML	Unified Modeling Language
URL	Uniform Resource Locator
URN	Uniform Resource Name
WebAPI	Web Application Programming Interface
WCF	Windows Communication Foundation
XML	Extensible Markup Language

ABBILDUNGSVERZEICHNIS

TABELLENVERZEICHNIS

LISTINGS

LITERATURVERZEICHNIS

Abbott, M. L., & Fisher, M. T. (2010). *The Art of Scalability.* New Jersey: Addison-Wesley.

Al Qutaish, R. E., & Abran, A. (2005). *An Analysis of the Design and Definitions of Halstead's Metrics.* Abgerufen am 17. 10. 2016 von Univserität von Québec: http://profs.etsmtl.ca/aabran/Accueil/AlQutaish-Abran%20IWSM2005.pdf

Ambler, S. W., & Sadalage, P. J. (2006). *Refactoring Databases: Evolutionary Database Design.* Boston: Addison Wesley Professional.

Balzert, H. (1998). *Lehrbuch der Software-Technik* (Bd. 1). Spektrum, Akad. Verlag.

Basili, V. R., Caldiera, G., & Rombach, H. D. (1994). The Goal Question Metric Approach. *Encyclopedia of Software Engineering*, 528-532.

Bernstein, P. A., Hadzilacos, V., & Goodman, N. (1987). *Concurrency Control and Recovery in Database Systems.* Reading, MA: Addison-Wesley Publishing Company.

Bortz, J., & Döring, N. (2006). *Forschungsmethoden und Evaluation für Human- und Sozialwissenschaftler* (4., überarbeitete Auflage Ausg.). Heidelberg: Springer Medizin Verlag.

Buschmann, F., Meunier, R., Rohnert, H., Sommerlad, P., & Stal, M. (1996). *Pattern-Oriented Software Architecture: A System of Patterns.* Chichester: Wiley.

Chidamber, S. R., & Kemerer, C. F. (1994). A Metrics Suite for Object Oriented Design. *IEEE Transactions on Software Engineering, 20*(6), 476-493.

Cockburn, A. (2005). *Hexagonal architecture*. Abgerufen am 17. 09. 2016 von Alistair.Cockburn.us: http://alistair.cockburn.us/Hexagonal+architecture

Conway, M. (1968). How Do Committees Invent. *Datamation*(April), 28-31.

Daya, S., Duy, N. V., Eati, K., Ferreira, C. M., Glozic, D., Gucer, V., . . . Vennam, R. (2015). *Microservices from Theory to Practice: Creating Applications in IBM Bluemix Using the Microservices Approach.* New York: IBM Redbooks.

Dräther, R., Koschek, H., & Sahling, C. (2013). *Scrum - kurz & gut.* Köln: O'Reilly.

Dumke, R., Foltin, E., Koeppe, R., & Winkler, A. (1996). *Softwarequalität durch Meßtools: Assessment, Messung und instrumentierte ISO 9000.* Braunschweig: Vieweg.

Evans, E. (2004). *Domain-Driven Design: Tackling Complexity in the Heart of Software.* Boston: Addison-Wesley.

Familiar, B. (2015). *Microservices, IoT, and Azure: Leveraging DevOps and Microservice Architecture to Deliver SaaS Solutions.* New York: Apress.

Fenton, N., & Bieman, J. (2015). *Software Metrics: A Rigorous and Practical Approach* (3. Auflage Ausg.). Florida: CRC Press.

Fielding, R. T. (2000). Architectural Styles and the Design of Network-based Software Architectures. *Dissertation*. Irvine: University of California.

Fildebrandt, U. (2016). *Solide Microservices bauen: Alte Prinzipien mit neuen Services*. Abgerufen am 17. 09. 2016 von JAXenter: https://jaxenter.de/solide-microservices-bauen-alte-prinzipien-mit-neuen-services-35451

Fowler, M. (2015). *Microservice Trade-Offs*. Abgerufen am 06. 07. 2016 von martinfowler.com: http://martinfowler.com/articles/microservice-trade-offs.html

Fowler, M., & Lewis, J. (01 2015). Microservices: Nur ein weiteres Konzept in der Softwarearchitektur oder mehr? *OBJEKTspektrum*, 14-20.

Gharbi, M., Koschel, A., Rausch, A., & Starke, G. (2015). *Basiswissen für Softwarearchitekten.* Heidelberg: dpunkt.verlag.

Gray, J. (1978). Notes on Data Base Operating Systems. *Lecture Notes in Computer Science*(60), 393-481.

Hanmer, R. (2013). *Pattern-Oriented Software Architecture For Dummies.* John Wiley & Sons.

innoQ. (2016). *SCS - Self-Contained Systems - SCS vs. Microservices*. Abgerufen am 16. 11. 2016 von SCS - Self-Contained Systems: http://scs-architecture.org/vs-ms.html

INTECO atec automation GmbH. (2016). *INTECO atec automation GmbH - Firmenprofil*. Abgerufen am 17. 10. 2016 von INTECO atec automation GmbH: http://www.inteco-atec.at/index_63_63_7_301_1_0__.html

ISO/IEC 25010:2011-03. (2011). *Systems and software engineering - Systems and software Quality Requirements and Evaluation (SQuaRE) - System and software quality models.* (International Organization for Standardization, Hrsg.) Genf, Schweiz.

Jay, G., Hale, J. E., Smith, R. K., Hale, D., Kraft, N. A., & Ward, C. (2009). Cyclomatic Complexity and Lines of Code: Empirical Evidence of a Stable Linear Relationship. *Journal of Software Engineering and Applications, 2*(3), 137-143.

Josuttis, N. M. (2007). *SOA in Practice: The Art of Distributed System Design.* Sebastopol: O'Reilly Media.

Kan, S. H. (2002). *Metrics and Models in Software Quality Engineering.* Boston: Addison-Wesley Professional.

Knoche, H. (2016). Sustaining Runtime Performance While Incrementally Modernizing Transactional Monolithic Software Towards Microservices. *Proceedings of the 7th ACM/SPEC on International Conference on Performance Engineering - ICPE '16*, 121-124.

Lewis, J., & Fowler, M. (2014). *Microservices: a definition of this new architectural term*. Abgerufen am 29. 05. 2016 von martinfowler.com: http://martinfowler.com/articles/microservices.html

Liggesmeyer, P. (2009). *Software-Qualität: Testen, Analysieren und Verifizieren von Software* (2. Auflage Ausg.). Heidelberg: Spektrum Akademischer Verlag.

Marshall, D. (1999). *Remote Procedure Calls (RPC)*. Abgerufen am 12. 09. 2016 von Cardiff University: http://www.cs.cf.ac.uk/Dave/C/node33.html

Martin, R. C. (2003). *Agile software development: principles, patterns and practices.* New Jersey: Pearson Education.

Martin, R. C. (2008). *Clean Code: A Handbook of Agile Software Craftsmanship.* Upper Saddle River, NJ: Prentice Hall.

McCabe, T. J. (1976). A Complexity Measure. *IEEE Transactions on Software Engineering, SE-2*(4), 308-320.

Microsoft. (2007). *Maintainability Index Range and Meaning - Code Analysis Team Blog*. Abgerufen am 18. 10. 2016 von Code Analysis Team Blog: https://blogs.msdn.microsoft.com/codeanalysis/2007/11/20/maintainability-index-range-and-meaning/

Microsoft. (2016). *Code Metrics Values*. Abgerufen am 28. 09. 2016 von Microsoft Developer Network: https://msdn.microsoft.com/en-us/library/bb385914.aspx

Newman, S. (2015). *Building Microservices.* Sebastopol: O'Reilly Media.

O'Hanlon, C. (2006). A Conversation with Werner Vogels. *Queue, 4*(4), 14-22.

Oman, P., & Hagemeister, J. (1992). Metrics for Assessing a Aoftware Aystem's Maintainability . *Software Maintenance, 1992. Proceerdings., Conference on* (S. 337-344). Orlando: IEEE.

Patterson, C., Smith, T., & Sellers, D. (2016). *Topshelf Key Concepts - Topshelf 3.0 documentation*. Abgerufen am 24. 10. 2016 von Topshelf 3.0 documentation: https://topshelf.readthedocs.io/en/latest/overview/faq.html

Racodon, D. (2016). *Metrics - Complexity*. Abgerufen am 09. 12. 2016 von SonarQube Documentation: http://docs.sonarqube.org/display/SONAR/Metrics+-+Complexity

Rau, K.-H. (2016). *Ausgewählte Aspekte der Einführungsphase.* Wiesbaden: Springer Fachmedien Wiesbaden.

Reenskaug, T. (1979). *MODELS - VIEWS - CONTROLLERS.* Abgerufen am 15. 06. 2016 von Universität von Oslo, Persönliche Hompages des Fachbereichs für Informatik: http://heim.ifi.uio.no/~trygver/1979/mvc-2/1979-12-MVC.pdf

RFC 2616. (1999). *Hypertext Transfer Protocol -- HTTP/1.1.* (IETF, Hrsg.) Abgerufen am 18. 10. 2016 von http://www.ietf.org/rfc/rfc2616.txt

Richardson, C. (2014). *Pattern: Monolithic Architecture*. Abgerufen am 05. 06. 2016 von Microservice architecture: http://microservices.io/patterns/monolithic.html

Richardson, C. (2016). *Refactoring a Monolith into Microservices*. Abgerufen am 20. 06. 2016 von NGINX: https://www.nginx.com/blog/refactoring-a-monolith-into-microservices/

Sneed, H. M., Seidl, R., & Baumgartner, M. (2010). *Software in Zahlen: Die Vermessung von Applikationen.* München: Hanser.

Spanneberg, B. (2015). Build-Automatisierung und Continuous Integration. In E. Wolff, *Continuous Delivery. Der pragmatische Einstieg* (S. 77-115). Heidelberg: dpunkt.verlag.

Starke, G., & Hruschka, P. (2016). *Struktur des arc42 Templates*. Abgerufen am 17. 10. 2016 von arc42: http://www.arc42.de/template/struktur.html

Strîmbei, C., Dospinescu, O., Strainu, R.-M., & Nistor, A. (2015). Software Architectures – Present and Visions. *Informatica Economică, 19*(4/2015), 13-27.

Toffetti, G., Brunner, S., Blöchlinger, M., Dudouet, F., & Edmonds, A. (2015). An architecture for self-managing microservices. *Proceedings of the 1st International Workshop on Automated Incident Management in Cloud*, 19-24.

Vernon, V. (2013). *Implementing Domain-Driven Design.* New Jersey: Addison-Wesley.

Vogel, O., Arnold, I., Chughtai, A., Ihler, E., Kehrer, T., Mehlig, U., & Zdun, U. (2009). *Software-Architektur Grundlagen - Konzepte - Praxis* (2. Auflage Ausg.). Heidelberg: Spektrum Akademischer Verlag.

Wolff, E. (2006). *JAOO 2006: Werner Vogels - CTO Amazon*. Abgerufen am 29. 05. 2016 von J and I and Me : http://jandiandme.blogspot.co.at/2006/10/jaoo-2006-werner-vogels-cto-amazon.html

Wolff, E. (2016). *Microservices: Grundlagen flexibler Softwarearchitekturen.* Heidelberg: dpunkt.

Young, C. (2014). *Hexagonal Architecture–The Great Reconciler?* Abgerufen am 17. 09. 2016 von Geeks With Blogs: http://geekswithblogs.net/cyoung/archive/2014/12/20/hexagonal-architecturendashthe-great-reconciler.aspx