Domänengetriebene Entwicklung von fortgeschrittenen Web-Anwendungen

Zur Erlangung des akademischen Grades eines

Doktors der Ingenieurwissenschaften

von der KIT-Fakultät für Informatik
des Karlsruher Instituts für Technologie (KIT)

genehmigte

Dissertation

von

Michael Schneider

aus Appenweier

Tag der mündlichen Prüfung: 06.02.2024

1. Referent: Prof. Dr. Sebastian Abeck
2. Referentin: Prof. Dr. Anne Koziolek

Inhaltsverzeichnis

Gender-Hinweis

In der vorliegenden Arbeit wird darauf verzichtet, bei Personenbezeichnungen sowohl die männliche als auch die weibliche Form zu nennen. Die männliche Form gilt in allen Fällen, in denen dies nicht explizit ausgeschlossen wird, für beide Geschlechter.

1 Einleitung

Dieses Kapitel motiviert und beschreibt die betrachtete Forschungsarbeit. Zu Beginn erfolgt in Abschnitt 1.1 die Motivation der Arbeit. In Abschnitt 1.2 wird die Einordnung der Arbeit in das Feld der Informatik erläutert. Die Problemstellungen werden durch ein betrachtetes Beispielszenario in Abschnitt 1.3 beispielhaft dargestellt. In Abschnitt 1.4 werden die Problemstellungen anschließend detailliert beschrieben. Abschnitt 1.5 beschreibt die Zielsetzungen der Arbeit, die sich aus den Problemstellungen ergeben und die Forschungsbeiträge erläutern. Die Prämissen (siehe Abschnitt 1.6) beschreiben die Rahmenbedingungen der Arbeit. Zuletzt wird in Abschnitt 1.7 der Aufbau der Arbeit beschrieben.

1.1 Motivation

Heutzutage spielt die Entwicklung von fortgeschrittenen Web-Anwendungen eine immer größere Rolle. Bei einer fortgeschrittenen Web-Anwendung handelt es sich im Kontext dieser Arbeit um eine modulare Microservice-basierte Anwendung, welche die kohäsiven Funktionalitäten bündelt und eine lose Kopplung der Softwarebausteine über Schnittstellen vorsieht. Insbesondere kommt hier die Microservice-Architektur zum Tragen, die eine Ausprägung einer serviceorientierten Architektur (SOA) ist und über die Komposition von mehreren Microservices die benötigte Funktionalität des Softwaresystems abbildet. Ein Microservice forciert durch das Prinzip der eindeutigen Verantwortlichkeit (engl. Single-Responsibility-Prinzip) die Aufteilung des Softwaresystems in eigenständige Softwarebausteine. Eine solche fortgeschrittene Web-Anwendung kann auch verschiedene Sensordaten des Internets der Dinge (Internet of Things, IoT) konsumieren. Ein Beispiel hierfür sind Anwendungen für Mobilitätslösungen [DK+18], welche über Schnittstellen bereitgestellte Sensorinformationen von Anbietern, wie Smart Cities [MK+17], konsumieren. Beispielsweise kann eine solche Anwendung mittels der Sensorinformationen die Suche nach freien Parkplätzen erleichtern. Eine Anforderung an eine solche Anwendung ist, dass diese flexibel ist und auch in der Zukunft um weitere Funktionalität erweitert werden kann. Weiterhin wird eine solche Anwendung mittels moderner Konzepte des DevOps (Devlopment and Operations) entwickelt. Hierbei werden die Prozesse zwischen den Softwareentwickler und den IT-Teams automatisiert und integriert. Hierbei kommt die Continuous Integration und Continuous Delivery (CI/CD) zum Einsatz, wobei die entwickelten Artefakte durch eine entsprechende CI/CD-Pipeline ausgeliefert werden.

Für die Entwicklung eines solchen Systems, welches flexibel für Erweiterungen ist und eine hohe Wartbarkeit aufweist, sind ein systematischer Entwicklungsprozess und eine durchdachte Architektur zwingend erforderlich. Der Microservice-basierte Ansatz, welcher ein System in verschiedene fachliche Services unterteilt, bietet sich für solche Anwendungen an [Ne15]. Um die Fachlichkeit zu bestimmen und den verschiedenen (Micro)Services zuzuordnen, ist der Einsatz des domänengetriebenen Entwurfs (engl. Domain-Driven Design, DDD) [Ev04] geeignet. Die strategische Aufteilung der Fachlichkeit wird gemäß DDD anhand der sogenannten Context Map modelliert. Die Context Map unterteilt die Domäne in verschiedene Unterdomänen und ordnet die Geschäftslogik in sogenannte Bounded Contexts ein. Jeder Bounded Context wiederum ist ein Kandidat für einen Microservice [Ne15]. Zwischen den Bounded Context können Beziehungen modelliert werden. Diese sind auf der strategischen und organisatorischen (und nicht auf der technischen) Ebene abgeordnet. Die Beziehungen zwischen den verschiedenen Bounded Contexts, die in der Context Map modelliert werden, bilden den Einfluss der Entwicklungsteams zwischen den Schnittstellen der verschiedenen Services ab. Kann ein Team stark bei der Gestaltung der Schnittstelle eines anderen Microservices mit entscheiden, so besteht eine enge Beziehung zwischen den Teams. Werden Services von außen konsumiert (externe Services), ist ein Einfluss meistens nicht möglich.

Die Context Map bildet diese Art der Beziehung (die Teamkommunikation) durch verschiedene Relationen ab. Soll eine neue Anwendung innerhalb der Domäne entwickelt werden, lassen sich die bereits etablierten Services der Domäne wiederverwenden. Bevor jedoch die verschiedenen Services wiederverwendet werden können, ist es wichtig, sich mit den Anforderungen der zu entwickelnden Anwendung zu beschäftigen. Schließlich muss bekannt sein, welche Funktionalitäten (und damit auch Services) für die Anwendung benötigt werden. Die Anforderungsspezifikation kann hierbei auf verschiedene Arten erfolgen [Po10]. Eine Möglichkeit der Anforderungserhebung ist die verhaltensgetriebene Softwareentwicklung (engl. Behavior-Driven Development, BDD) [Sm14]. Bei diesem Ansatz werden die Anforderungen in sogenannten Features spezifiziert. Das Domänenwissen formt die Basis und legt die ubiquitäre Sprache fest. Durch ein systematisches Vorgehen lässt sich aus den Anforderungen zusätzliche Domänenlogik extrahieren. Dieses aufgespürte Domänenwissen wird, insofern kein passender Bounded Context existiert, in einen oder mehrere neue Bounded Contexts abgebildet. Zusätzlich müssen die neuen, fachlichen Bounded Contexts in die existierende Context Map eingegliedert werden, da die notwendige Fachlichkeit für die entstehende Anwendung bisher noch nicht betrachtet wurde, aber für weitere Anwendungen aus der Domäne relevant sein können.

Bei der Entwicklung einer solchen fortgeschrittenen Anwendung sind auch die auftretenden Querschnittsdomänen zu berücksichtigen. Ein Beispiel hierfür sind die Sensordaten, welche durch die Querschnittsdomäne Internet der Dinge (Internet of Things, IoT) bereitgestellt werden. Aber auch andere Querschnittsfunktionalitäten wie die Identitäts- oder Zugriffsverwaltung (Identity and Access Management, IAM) sind bei der Entwicklung zu berücksichtigen. Hierbei ist es wichtig, dass diese Funktionalitäten aus den Microservices der Anwendung herausgezogen werden und so erfasst werden,

dass diese auch bei der Entwicklung von anderen Anwendungen wiederverwendet werden können. Hierzu ist die Strukturierung der Fachlichkeit der Querschnittsdomänen notwendig, welche keinerlei Inhalte der Geschäftsdomäne enthält, sondern die benötigte Funktionalität generisch anbietet.

Ein weiterer wichtiger Aspekt des systematischen Entwicklungsprozesses ist das Testen. Beim Testen des Systems handelt es sich um dessen Untersuchung auf die Erfüllung der gewünschten Anforderungen (Akzeptanzkriterien) und auf die Fehlerfreiheit des Systems. Dieses Testkonzept soll nicht alleinstehend, sondern die Entwicklung der fortgeschrittenen Web-Anwendungen unterstützen. Das Testkonzept soll hierbei die verschiedenen Arten von Tests berücksichtigen und die systematische Entwicklung der verschiedenen Tests ermöglichen. Hierunter fallen unter anderem Unit Tests, Komponententests, Integrationstests, Vertragstests und Ende-zu-Ende-Tests. Erste Ideen für eine Testarchitektur werden bereits in der Literatur behandelt [RG15]. Bei einem verteilten Software-System, wie bei einer Microservice-Architektur, ergeben sich weitere Herausforderungen an das Testen, da die benötigten Funktionalitäten auf verschiedene Services verteilt sind und über Schnittstellen (z. B. über REST-APIs) kommunizieren. Sobald die Schnittstelle eines anderen Microservices benötigt wird, stellt sich die Frage, wie diese Abhängigkeiten verfügbar gemacht werden können. Die bereitgestellte Funktionalität eines einzelnen Microservices lässt sich unabhängig von der Anwendung testen. In diesem Kontext spielt auch der Begriff DevOps eine wichtige Rolle. Um einen Microservice nutzen zu können, muss dieser bereitgestellt werden und zugänglich sein. Mittels CI/CD können Tests auf Basis von Code-Änderungen vollautomatisch ausgeführt und bei erfolgreichem Testdurchlauf direkt ausgeliefert werden.

1.2 Einordnung der Arbeit

Das Gebiet der Arbeit lässt sich im Bereich der Softwaretechnik einordnen. Die Softwaretechnik selbst umfasst verschiedene Wissensgebiete [BH+14]. Für die Entwicklung von Anwendungen werden Prozessmodelle genannt, die verschiedene Phasen wie die Analyse, den Entwurf und die Implementierung vorsehen. Diese Arbeit beschäftigt sich mit den Wissensgebieten der Anforderungsanalyse, dem Entwurf und der Implementierung und dem Testen. Die Anforderungsanalyse selbst erfasst funktionale Anforderungen, um das gewünschte Systemverhalten zu bestimmen [MB+01]. Neben funktionalen Anforderungen werden ebenfalls nicht-funktionale Anforderungen erfasst [CN+12]. Ein wichtiger Punkt bei der Anforderungsanalyse ist, dass diese systematisch in den Entwurf überführt werden kann.

Neben der genannten Anforderungsanalyse bildet der Entwurf einen Schwerpunkt dieser Arbeit. Als Architektur-Stil wird die Microservice-Architektur vorgesehen, die in der Geschäftswelt im Trend liegt [FM+17].

Beim Entwurf einer solchen verteilten Architektur sind die Prinzipien der hohen Kohäsion und der losen Kopplung zu berücksichtigen [Ka03]. Bei der betrachteten Microservice-Architektur kann ein Microservice als eine Softwarekomponente eingestuft werden [Ne15]. Diese Softwarekomponente hat Schnittstellen nach außen, die die gewünschte Funktionalität bereitstellen. Ein wichtiger Punkt zur Aufstellung der einzelnen Software-Komponenten ist ein systematisches Vorgehen, bei dem die zusammengehörende Funktionalität ermittelt wird. Für die Extraktion des notwendigen Wissens und zur Einteilung der Fachlichkeit wird auf den domänengetriebenen Entwurf von Evans [Ev04] zurückgegriffen. In diesem Rahmen fällt der Begriff des Bounded Contexts [Ev04], der auf der Modellierungsebene die Fachlichkeit einer funktionalen Einheit absteckt. Durch die Extraktion der Fachlichkeit und Separierung dieser in eigene Softwarekomponenten lässt sich die Wiederverwendbarkeit der Softwarebausteine erhöhen.

1.3 Betrachtetes Szenario

Die verfolgten Problemstellungen werden anhand eines Szenarios tiefer beschrieben. Das Szenario erfasst die aktuelle Ausgangssituation bei der Softwareentwicklung. Die im Szenario aufgegriffenen Defizite sollen durch das in dieser Arbeit behandelte Vorgehen zur domänengetriebenen Entwicklung gelöst werden und die Zielsetzung der Arbeit weiter motivieren.

1.3.1 Ausgangssituation

Eine große Problematik bei der Softwareentwicklung ist, dass Anwendungen isoliert voneinander entwickelt werden. Dies führt dazu, dass entwickelte Softwarebausteine sich nur bedingt oder nicht wiederverwenden lassen. Als Resultat werden ähnliche Softwarebausteine (oder Bausteine mit der gleichen Funktionalität) immer wieder neu entwickelt [MT+20]. Insbesondere Anwendungen aus der gleichen Domäne können Funktionen aus bereits entwickelten Anwendungen wiederverwenden. Dies ist aber nicht möglich, da diese Funktionalitäten fest in die Anwendung kodiert sind. Im betrachteten Szenario werden verschiedene Anwendungen aus der Domäne ConnectedCar entwickelt, die auf dem gleichen Domänenwissen aufbauen. Bei dieser Entwicklung wird keine Extraktion des Domänenwissens vorgenommen, sondern jede dieser Anwendungen wird separat voneinander entwickelt. Dadurch entsteht der Nachteil, dass für jede dieser Anwendungen das benötigte Domänenwissen aus der gleichen Domäne (in diesem Fall der Domäne ConnectedCar) erneut extrahiert, entworfen und implementiert wird. Dies führt dazu, dass für die Entwicklung jeder dieser Anwendungen das bisherige Wissen "verloren" geht und ein höhere zeitlicher Faktor für die Entwicklung benötigt wird.

Um dieses Defizit zu beheben, wird in dieser Arbeit deshalb das geteilte Domänenwissen strukturiert extrahiert und festgehalten, damit die Entwicklung weiterer Anwendungen aus der Domäne beschleu-

nigt wird und sprichwörtlich das Rad nicht immer neu erfunden werden muss. In Abbildung 1.1 wird die oben beschriebene grundlegende Problematik, die bei der Anwendungsentwicklung auftritt, skizziert und mit dem Lösungsansatz verbunden.

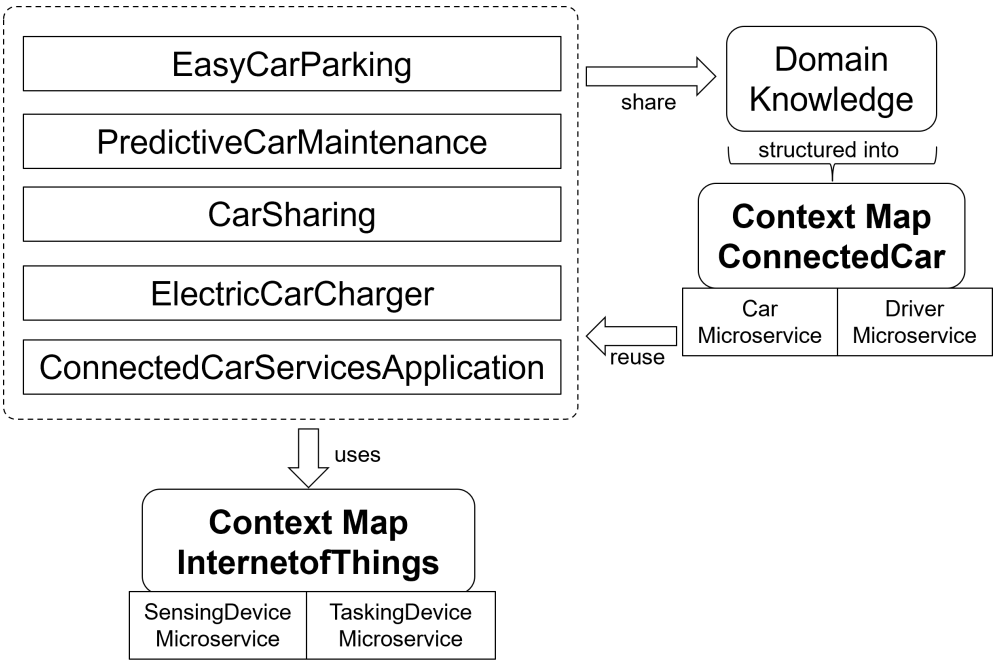


Abbildung 1.1: Szenario verschiedener Anwendungen unter Einbezug der Geschäftsdomäne und einer Querschnittsdomäne

Das Domänenwissen selbst wird anhand der Context Map strukturiert. Die konkrete Modellierung der einzelnen Fachlichkeitsbereiche erfolgt durch die strategische Aufteilung innerhalb der Context Map [Ev04] statt. Die Strukturierung erfolgt so, dass die verschiedenen Fachlichkeiten einsortiert werden. Das hier befindliche strukturierte Domänenwissen liefert die Grundlage für die wiederverwendbaren Softwarebausteine. Analog erfolgt dies für die benötigte Funktionalität der Querschnittsdomäne. So kann das hier etablierte Wissen ebenfalls für die verschiedenen Microservices (wieder)verwendet werden.

Bei der Implementierung weiterer Anwendungen aus der Domäne werden die bereits entworfenen Softwarebausteine in Betracht gezogen. Hierbei kann auf die bestehende Modellierung sowie auf die bestehende Implementierung zurückgegriffen werden. Hierbei kann ein verteilter Softwarebaustein von mehreren Anwendungen genutzt werden, aber auch für mehrere Anwendungen mehrfach verteilt sein. Dies kommt auf den Einsatzzweck und die Datengrundlage der Anwendungen an. Nutzen zwei Anwendungen die gleiche Datengrundlage, dann können diese die gleiche Microservice-Instanz nutzen. Alternativ können zwei Anwendungen aber auch unterschiedliche Datensätze verwenden. In diesem Fall muss der Microservice mehrfach verteilt werden. Beispielsweise kann der Microservice

"Driver" zweimal mit einer unterschiedlichen Datenbank verteilt sein. Einmal für die Anwendung EasyCarParking und einmal für die Anwendung CarSharing. Um die oben angesprochene Ziele zu erreichen, wird ein systematischer Engineering-Ansatz benötigt.

1.3.2 Entwicklung der Microservice-Architektur

Das primäre Ziel ist es, die systematische Entwicklung von fortgeschrittenen Web-Anwendungen zu unterstützen. Abbildung 1.2 zeigt den fachlich-inhaltlichen Überblick auf einem hohen Abstraktionsniveau. Eines der wichtigsten Ziele hierfür ist, dass eine klar strukturierte Microservice-Architektur bei der Entwicklung von fortgeschrittenen Web-Anwendungen entsteht. Damit dieses Ziel jedoch erreicht werden kann, ist ein iterativer und strukturierter Entwicklungsprozess für diese Anwendungen notwendig. Der Prozess selbst besteht, analog zu Vorgehensmodellen [Sc02] aus der Softwareentwicklung, aus verschiedenen Phasen. In den jeweiligen Phasen werden Artefakte erzeugt, welche wiederum als Eingabe für weitere Schritte verwendet werden. Zu Beginn des Vorgehens steht die Analysephase, welche bereits die ersten Artefakte generiert. Das Ziel hierbei ist, das gewünschte Verhalten der Software durch funktionale Anforderungen zu erhalten. In dieser Phase werden die Anforderungen anhand von Anforderungsfällen erhoben und spezifiziert.

Nachdem durch die Analysephase die wichtigsten Anforderungen der Anwendung erfasst und spezifiziert wurden, folgt im nächsten Schritt der Entwurf. Hierbei wird der Entwicklungsansatz so aufgestellt, dass eine möglichst strukturerhaltende Überführung möglich ist. Während des Entwurfs werden die Analyseartefakte systematisch in eine die klar strukturierte Microservice-Architektur überführt. Dies beinhaltet die Aufstellung der Software-Architektur, sowohl als auch die benötigten Artefakte zur Etablierung der Microservices und deren Schnittstellen. Einer der wichtigsten Punkte hierbei ist die Separierung der Geschäftsdomäne und der Funktionalitäten der Querschnittsdomänen. Beispielsweise bedeutet dies für die IoT-Domäne, dass keine Geschäftsobjekte in die Domäne IoT integriert werden und umgekehrt auch keine Geschäftsobjekte in die Domäne IoT einfließen. Als Resultat werden verschiedene Domänen erhalten - eine fachliche Domäne, welche das Geschäft strukturiert und beschreibt und eine oder mehrere Querschnittsdomänen, die sich um die weiteren Funktionalitäten (bspw. Identity and Access Management, IoT) kümmern und nur indirekt mit der Geschäftsdomäne agieren (aber trotzdem wichtig sind).

Um die strategische Aufteilung der Fachlichkeit vorzunehmen, wird in diesem Schritt des Entwicklungsprozesses DDD [Ev04] eingesetzt. Das wichtigste und essenzielle Artefakt hierbei ist die Context Map. Hierzu zählt auch die Eingliederung neuer Bounded Contexts in eine Context Map, sowie der Umgang mit der Context Map im Entwicklungsprozess. Durch die strategische Aufteilung und Separierung der Geschäfts- und Querschnittsdomäne werden die verschiedenen Bounded Contexts der jeweiligen Domäne bestimmt und in die zugehörige Context Map der Domäne integriert. Beispielsweise wir die Domäne IoT in mehrere funktionale Dienste unterteilt, welche entsprechend in die

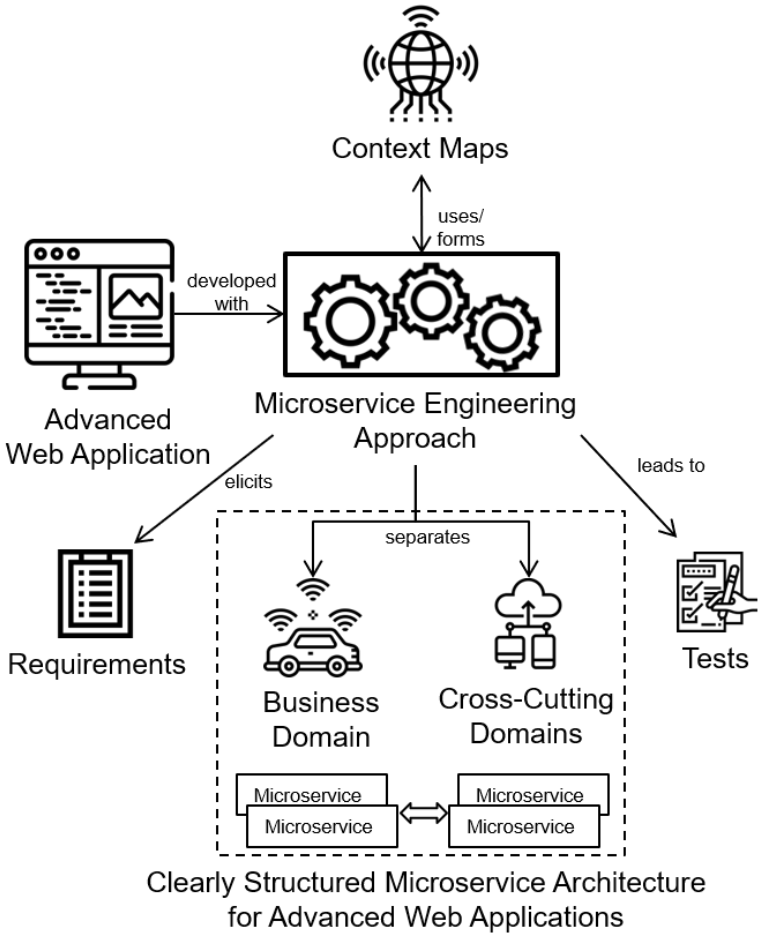


Abbildung 1.2: Überblick über den Inhalt und das Zusammenspiel der Artefakte

Context Map für IoT eingegliedert werden. Daraus ergibt sich die Microservice-Architektur für die Domäne IoT, welche für verschiedene IoT-Anwendungen wiederverwenden lässt, da die Services die benötigten Funktionalitäten erfüllen. Bei diesem Schritt ist darauf zu achten, dass fachlichen Inhalte nicht in die Services der Querschnittsdomäne (bspw. die Domäne IoT) aufgenommen werden. Dies gilt dies auch für die fachliche Domäne, da schließlich keine Querschnittsfunktionalitäten (bspw. IoT-spezifischen Funktionalitäten) in diesen Services platziert werden sollte.

Sobald die strategische Einordnung der Services erfolgt ist, wird die taktische Modellierung der noch nicht etablierten/ vorhandenen Services vorgenommen. Bei der taktischen Modellierung sind zusätzlich Standards (wie beispielsweise die OGC SensorThings API [OGC-STA-Sen] bei der Domäne IoT) berücksichtigt. Anschließend erfolgt die Implementierung und das Testen der benötigten Microservices und der Benutzerschnittstelle, sowohl auch als der gesamten Anwendung mittels Ende-zu-Ende-Tests. Um eine verteilte Microservice-Architektur zu Testen ist ein entsprechendes Testkonzept notwendig, welches in den Entwicklungsprozess integriert wird. Dieses Testkonzept beschriebt den Einsatz der

verschiedenen Testarten während der Entwicklung und führt systematisch zu den verschiedenen Tests während der Implementierung. Als Resultat ergeben sich neben den Unit-Tests, Integrationstests und Schnittstellentests, sowie Ende-zu-Ende-Tests.

1.4 Problemstellungen

Das Szenario umfasst verschiedene Problemstellungen, die im Folgenden näher betrachtet werden. Diese Problemstellungen bilden die Grundlage für die Ziele und Beiträge dieser Arbeit. Die einzelnen Problemstellungen sind mit einer Kennung versehen (P1-P4).

1.4.1 P1: Fehlende Systematik zur Aufstellung der Microservice-Architektur aus den Analyseartefakten

Bei der Entwicklung einer Microservice-Architektur ist es essenziell, dass die Anforderungen in den Entwurf überführt werden können. Gemäß den Konzepten aus DDD und deren Bezug zu Microservices [Ev04, Ne15] lassen sich aus den sogenannten Bounded Contexts Kandidaten für die Microservices gewinnen. Hieraus entsteht die Problemstellung, wie die Analyseartefakte gestaltet werden können, um eine systematische und möglichst strukturerhaltende Überführung der Analyseartefakte in den Entwurf mit verschiedenen Softwarebausteinen gestaltet werden kann. Die bestehenden Arbeiten bieten Lösungen in Form von Ansätzen und Prozessbeschreibungen an, eine ganzheitliche strukturerhaltende Systematik für eine domänengetriebene Entwicklung ist jedoch nicht gegeben. Für die Entwicklung eines solchen Systems, welches flexibel für Erweiterungen ist und eine hohe Wartbarkeit aufweist sowie wiederverwendbare Softwarebausteine hervorbringt, sind ein systematischer Entwicklungsprozess und eine durchdachte Architektur zwingend erforderlich. Ein Microservice-basierter Entwicklungsansatz, welcher ein System in verschiedene fachliche Services unterteilt, bietet sich für solche Anwendungen an. Um die Fachlichkeit zu bestimmen und den verschiedenen Microservices zuzuordnen, ist der Einsatz des domänengetriebenen Entwurfs geeignet.

1.4.2 P2: Separierung der anwendungsagnostischen Funktionalitäten

Bei der Anforderungsspezifikation wird festgehalten, welche Anforderung eine Anwendung erfüllen soll. Beim Entwurf müssen die verschiedenen Geschäfts- und Querschnittsfunktionalitäten, welche anfallen, voneinander separiert werden. Weiterhin ist wichtig, dass anwendungsagnostische Funktionalitäten herausgezogen werden. Als Resultat sollen kohäsive Softwarebausteine entstehen, die die anwendungsagnostischen Funktion bündeln. Die Separierung dieser Funktionalitäten wird als zweite Problemstellung gesehen. Hierzu wird eine Systematik und Richtlinien benötigt, um eine Trennung vorzunehmen.

1.4.3 P3: Fehlende Wiederverwendbarkeit der Softwarebausteine

Bei der Entwicklung von fortgeschrittenen Web-Anwendungen wird in aktuellen Arbeiten zu wenig auf die Trennung von geschäftlichen und Querschnittsfunktionalitäten geachtet. Infolgedessen werden die Softwarebausteine einer solchen fortgeschrittenen Web-Anwendung mit den Querschnittsaufgaben vermischt. Dies führt dazu, dass die Softwarebausteine nicht oder nur sehr bedingt von anderen Anwendungen wiederverwendet werden können, da die Anwendungsspezifika mit in den Softwarebaustein einfließt. Daher wird die Querschnittsfunktionalität extrahiert, um von mehreren Anwendungen wiederverwendet werden zu können. Beispielsweise werden IoT-Daten und Funktionalitäten direkt bei der Anwendung platziert. Diese Funktionalität wird aber ebenfalls von weiteren Anwendungen benötigt, weshalb eine Auslagerung in eigene Softwarebausteine zu bevorzugen ist. Daher bezieht sich diese Problemstellung auf die fehlende Wiederverwendbarkeit der Services. Standards wie die OGC SensorThings API [OGC-STA-Sen] und deren Implementierung durch den FROST®-Server [Fra-Fro] bilden hier erste Ansätze. Diese müssen aber bei der Entwicklung der Anwendung berücksichtigt werden.

1.4.4 P4: Testen der verteilten Anwendung

Microservices können die anstehenden (Teil-)Aufgaben eines Softwaresystems autonom durchführen [FL14]. Beim Testen einer Microservice-basierten Anwendung entstehen nun verschiedene Anforderungen [GW+19], welche durch ein systematisches Testkonzept abgedeckt werden sollen. Hierbei stehen nicht nur die verschiedenen Testarten im Fokus, sondern auch die systematische Ableitung der Testfälle aus den aufgestellten Artefakten ist von großer Bedeutung. Der Entwickler, bzw. Tester sollte hierbei auf die Artefakte des Engineerings-Ansatzes zurückgreifen können und gezielt die Tests aufstellen können.

1.5 Zielsetzungen und Beiträge

Die bestehenden in der Literatur bestehenden Arbeiten (siehe Kapitel 3) bieten Lösungen in Form von Ansätzen und Prozessbeschreibungen an, eine ganzheitliche strukturerhaltende Systematik für eine domänengetriebene Entwicklung ist jedoch (gemäß dem Handlungsbedarf, siehe Abschnitt 3.3) nicht gegeben. Für die Entwicklung eines solchen Systems, welches flexibel für Erweiterungen ist und eine hohe Wartbarkeit aufweist sowie wiederverwendbare Softwarebausteine hervorbringt, sind ein systematischer Entwicklungsprozess und eine durchdachte Architektur zwingend erforderlich. Ein Microservice-basierter Entwicklungsansatz, welcher ein System in verschiedene fachliche Services unterteilt, bietet sich für solche Anwendungen an. Um die Fachlichkeit zu bestimmen und

den verschiedenen Microservices zuzuordnen, ist der Einsatz des domänengetriebenen Entwurfs geeignet.

Basierend auf den angesprochenen Problemen werden nun die Ziele dieser Arbeit genauer erläutert. Der zentrale Beitrag dieser Arbeit ist ein systematischer Engineering-Ansatz zur Entwicklung solcher fortgeschrittenen Web-Anwendungen. Hierbei gilt es, eine Systematik zu definieren, welche die Wiederverwendbarkeit der Softwarebausteine ermöglicht. Ein Ziel dieser Arbeit ist es, die Querschnittsdomänen in Form einer strategischen Modellierung zu erfassen und mittels des domänengetriebenen Entwurfs in mehrere, wiederverwendbare Microservices aufzuteilen. Durch ein systematisches Vorgehen sollen sich diese Microservices leicht bei der Entwicklung von Anwendungen verwenden lassen. Ein weiterer Beitrag in Bezug auf das Gesamtkonzept ist ein durchgängiges Testkonzept. Dieses Testkonzept berücksichtigt hierbei die verschiedenen Arten von Tests wie Unit-, Komponenten-, Integrations-, Vertrags- und Ende-zu-Ende-Tests und wird in den Gesamtprozess eingeordnet. Hierzu werden in der Arbeit die folgenden Zielsetzungen und Beiträge (B1 - B3) im Rahmen des Engineering-Ansatzes festgehalten.

1.5.1 B1 - Systematischer Engineering-Ansatz zur Aufstellung der Anwendungsarchitektur

Ein Ziel dieser Arbeit ist es, die Architektur der Anwendung zu erfassen und mittels des domänengetriebenen Entwurfs in mehrere, wiederverwendbare Microservices aufzuteilen. Die zentrale Zielsetzung dieser Arbeit ist ein systematischer Engineering-Ansatz zur Entwicklung solcher fortgeschrittenen Web-Anwendungen. Hierzu ist eine Analyse der Domäne und der strategischen Aufteilung der Fachlichkeit gemäß DDD [Ev14] anhand der sogenannten Context Map notwendig. Die Context Map strukturiert die Domäne in verschiedene Unterdomänen und ordnet die Geschäftslogik in sogenannte Bounded Contexts ein. Jeder etablierte Bounded Context ist ein Kandidat für einen Microservice. Soll eine neue Anwendung innerhalb der Domäne entwickelt werden, lassen sich die bereits etablierten Services der Domäne wiederverwenden. Ein weiterer Punkt ist der Bezug der etablierten Domäneninhalte zu der eigentlichen Anwendungsentwicklung. Hier müssen die verschiedenen Services in die Architektur der Anwendung eingegliedert werden. Neben der Trennung der Fachlichkeit ist auch die Etablierung der Schnittstellen wichtig. Der Entwurf ist so gestaltet, dass die Schnittstellen systematisch aus den erstellten Diagrammen abgeleitet werden können. Für die anstehende Implementierung soll eine systematische Strukturierung eingeführt werden.

1.5.2 B2 - Separierung der Geschäfts- und Querschnittsdomänen

Bei der Entwicklung von fortgeschrittenen Web-Anwendungen besteht die Gefahr, dass Querschnittsinhalte, wie beispielsweise IoT in die Geschäftsdomäne aufgenommen werden. In diesem Fall können

die modellierten Geschäftsobjekte mit einer hohen Wahrscheinlichkeit nicht mehr in anderen Domänen bzw. Geschäftsbereichen wiederverwendet werden, da diese rein für den Anwendungszweck zugeschnitten sind. Analog gilt dies auch für weitere Querschnittsdomänen. Dementsprechend müssen diese Services generisch sein und dürfen keine geschäftsspezifischen Teile enthalten, damit diese von anderen Domänen wiederverwendet werden können. Ein wichtiger Punkt hierbei ist, wie die Services für die Anwendung orchestriert werden, so dass von der Anwendung die notwendige Funktionalität erfüllt werden kann. Hierbei werden auf der architekturellen Ebene entsprechende Diagramme benötigt, die diesen Umstand aufzeigen. Daher ist es wichtig, dass ein Vorgehen erarbeitet wird, wie eine solche Verknüpfung zwischen den beteiligten Domänen hergestellt werden kann. Als praktische Erprobung dient die Entwicklung verschiedener Anwendungen aus unterschiedlichen Domänen, wie beispielsweise die Anwendung ClinicsAssetManagement (CAM) aus der Domäne des Gesundheitswesens und die Anwendung PredictiveCarMaintenance aus der Domäne ConnectedCar. Diese Anwendungen benötigen von der Aufbereitung bis hin zur Bereitstellung der Sensordaten die Services aus der betrachteten Domäne IoT, aber auch die fachlichen Services der Domäne ConnectedCar.

1.5.3 B3 - Testkonzept für Microservice-basierte Anwendungen

Bei einem verteilten Software-System sind die benötigten Funktionalitäten auf verschiedene Services verteilt und es wird über Schnittstellen (z. B. über REST-APIs) kommuniziert. Um sicherzustellen, dass die Anwendung fehlerfrei funktioniert und die geforderte Funktionalität erfüllt, ist die Aufstellung eines Testkonzepts für die systematische Entwicklung fortgeschrittener Web-Anwendungen essenziell. Das Ziel ist es, dass der Entwickler ein Vorgehen und Werkzeuge an die Hand bekommen, welches die systematische Umsetzung der verschiedenen Tests ermöglicht. Dies beinhaltet die Ableitung der Tests und der notwendigen Testdaten. Die Tests und Teststrukturen sollen hierbei die agile Entwicklung der fortgeschrittenen Web-Anwendungen unterstützen und die verschiedenen Tests wie Unit-Tests, Integrationstests, Vertragstests und Ende-zu-Ende-Tests berücksichtigen. Gerade bei den Ende-zu-Ende-Tests oder Akzeptanztests [Sm14], welche die gesamte Anwendung betrachten stellt sich die Frage, wie das Testumfeld aufgebaut werden muss, damit eine Microservice-basierte Anwendung sinnvoll getestet werden kann. Hierzu müssen die Services (oder zumindest Platzhalter-Services in Form von sogenannten Mocks oder Mock-Objekten) für die Tests bereitstehen, da Funktionalitäten einer Anwendung oftmals mehrere Services benötigen. Die Integration in die CI/CD-Pipeline spielt hier eine bedeutende Rolle, da die Tests automatisiert ablaufen sollen. Als Ziel sollten "Best Practices" für eine Entwicklungsumgebung entstehen, die die Entwicklung und Ausführung der Tests für den Entwickler erleichtert.

1.5.4 Tragfähigkeit der Beiträge

Anhand praktischer Beiträge und die Umsetzung der Konzepte bei der Entwicklung von Anwendungen, erfolgt die Demonstration der Tragfähigkeit. Hierbei werden verschiedene Anwendungen aus der Domäne ConnectedCar entwickelt, um die Wiederverwendbarkeit der Domänen-Microservices zu demonstrieren. Anwendungen die entwickelt wurden sind die PredictiveCarMainenenance zur präventiven Wartung von Fahrzeugen und die ConnectedCarServices zur Verwaltung und Vermietung/-Mietung von Fahrzeugen. Zusätzlich wird gemeinsam mit Kooperationspartnern eine Anwendungen mit der Bezeichnung CustomerPortal entwickelt, die der Zugang für die Produkte eines Kunden darstellt.

Die entwickelten Anwendungen dienen als Nachweis der Anwendbarkeit der erzielten Forschungsergebnisse im industriellen Bereich. Die Beiträge haben das Ziel, eine systematische Entwicklung zu ermöglichen und anhand des eigenen Entwicklungsprozesses zur Domäne den Wiederverwendungsgrad der entwickelten Services innerhalb der Domäne zu erhöhen. Weiterhin ist das Ziel die Querschnittsfunktionalität auszulagern, so dass diese ebenfalls von verschiedenen Services wiederverwendet werden kann. Dies reduziert auf einen längeren Zeitraum die Kosten, da die gleichen Inhalte nicht immer wieder neu entwickelt werden müssen.

1.6 Prämissen der Arbeit

Im folgenden Abschnitt wird die Abgrenzung und Schärfung der Forschungsschwerpunkte vorgenommen. Hierbei wird der Problembereich der Arbeit eingegrenzt.

1.6.1 Prämisse 1: Microservice-basierte Anwendung

Das Hauptziel ist die Unterstützung von Entwicklern bei der Entwicklung von Microservice-basierten Anwendungen aus verschiedenen Domänen. Ein Softwaresystem lässt sich durch verschiedene Paradigmen und Architekturstile umsetzen. Diese Arbeit konzentriert sich dabei auf die Microservice-Architektur und zeigt anhand von Engineering-Artefakten ein konkretes Vorgehen zur Erstellung und Anwendung von Artefakten für diesen Architekturstil.

1.6.2 Prämisse 2: REST-basierte Schnittstellen

Die Schnittstelle ist der Dreh- und Angelpunkt eines Microservices. Im Hinblick auf die Literatur zu den Microservices beschränkt sich die Auswahl der Schnittstellenkonzepte insbesondere auf REST-basierte Schnittstellen. Andere Arten von Schnittstellen, wie z.B. Remote Procedure Calls

mittels gRPC [IK20] oder der Einsatz von Event-Bus-basierten Technologien werden nur teilweise angesprochen, aber nicht weiter diskutiert.

1.6.3 Prämisse 3: Domänengetriebener Entwurf

Um eine Microservice-Architektur für eine Querschnittsdomäne (wie die Querschnittsdomäne IoT) bereitzustellen, muss die Domäne in mehrere funktionale Dienste unterteilt werden. Die Aufteilung der Fachlichkeiten erfolgt mit dem domänengetriebenen Entwurfs (Domain-Driven Design, DDD) [Ev04]. Dadurch werden wichtigen Konzepte einer Domäne erfasst und modelliert. Weiterhin wird die von DDD nicht betrachtete Formalisierung anhand der vorhandenen Literatur als Quelle hinzugezogen.

1.6.4 Prämisse 4: Abgrenzung zu einer Referenzarchitektur

Die aufgestellte Microservice-Architektur ist von einer Referenzarchitektur abzugrenzen, da diese nur bedingt auf technische Details wie Protokolle eingeht. Die aufgestellte Architektur dient als Servicelandschaft für Anwendungen innerhalb einer Domäne. Das Ziel ist die fachliche Aufstellung der Querschnittsdomäne und die Aufstellung eines Vorgehens zur Verknüpfung mit fachlichen Domänen, so dass sich die Querschnittsdomäne bei der Anwendungsentwicklung leicht integrieren lässt.

1.6.5 Prämisse 5: Berücksichtigung von Standards

Gerade bei der Ableitung der Domäninhalte lassen sich Standards heranziehen. Insbesondere für die Querschnittsdomänen wie IoT lassen sich mehrere Quellen heranziehen. Diese Literatur umfasst Referenzarchitekturen, Standards und vieles mehr. Vor allem bekannte Standards wie die SensorThings API [OGC-STA-Sen] werden bei der Aufstellung der Architektur und der konkreten Services berücksichtigt und haben einen Einfluss auf die Arbeit.

1.6.6 Prämisse 6: Beschränkung auf bestimmte funktionale Tests

In dieser Arbeit liegt der Fokus auf einem Testkonzept für die Entwicklung von Microservicebasierten Anwendungen, das entsprechend der Testpyramide die verschiedenen Testarten von Ende-zu-Ende-, Integration- und Consumer-Driven-Contract-Tests bis zu den sogenannten Unit-Tests abdeckt. Ausgeklammert werden hierbei Tests, welche sich auf nicht-funktionale Anforderungen beziehen, wie beispielsweise Tests zur Verfügbarkeit und Performance. Ziel ist es, den Entwicklern eine Systematik bereitzustellen um gezielt Tests basierend auf den Anforderungen und der Microservice-Architektur umzusetzen.

1.7 Strukturierung der Arbeit

In Kapitel 1 wird die Arbeit motiviert und auf die Einordnung der Forschungsbeiträge eingegangen. Weiterhin wird die Zielsetzung und die wissenschaftlichen Forschungsbeiträge formuliert und die Prämissen festgelegt.

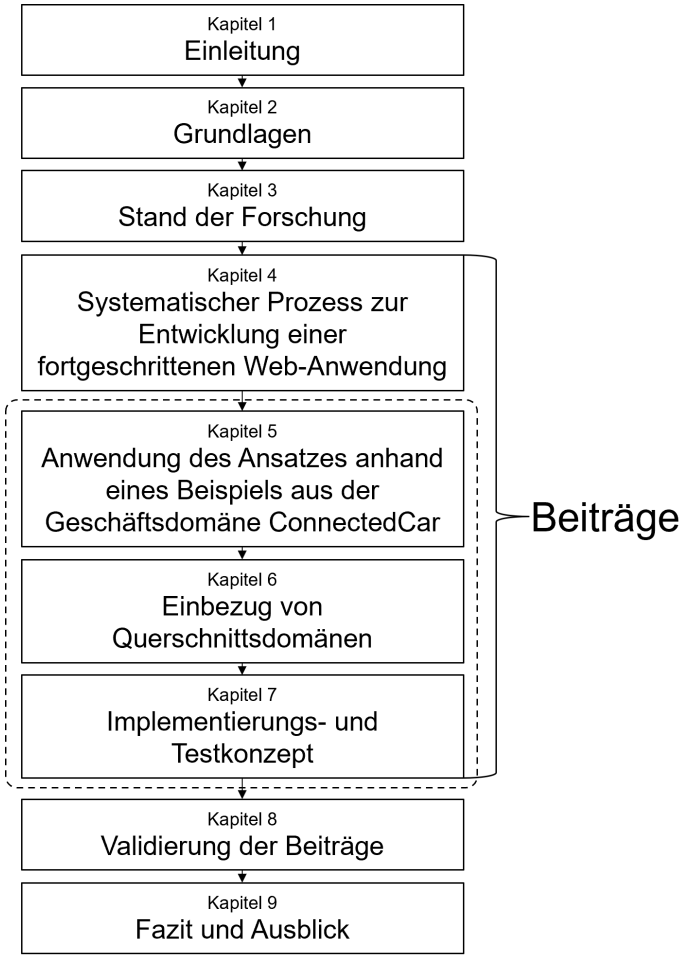


Abbildung 1.3: Struktur der Arbeit

Kapitel 2 befasst sich mit den benötigten Grundlagen der Arbeit. Hierbei werden benötigt Konzepte erfasst und kurz dargestellt. Weiterhin werden weiterführende Referenzen aufgezeigt. Der Forschungsstand wird in Kapitel 3 beleuchtet und bewertet. Basierend auf den Ergebnissen, wird der Handlungsbedarf aufgestellt, der in den nachfolgenden Kapiteln behandelt wird.

In Kapitel 4 wird das Gesamtkonzept vorgestellt. Hierbei wird der Ansatz in zwei verschiedene Prozesse aufgeteilt. Der Domänenentwicklungsprozess zur Erfassung des anwendungsagnostischen Domänenwissens sowie der dem Anwendungsentwicklungsprozess zum Entwurf der Anwendungslogik. Durch die Software-Architektur werden die hierbei entstandenen Microservices verknüpft.

Exemplarisch wird der Ansatz in Kapitel 5 anhand der ConnectedCarServiceApplication (CCSApp) aus der Domäne ConnectedCar demonstriert. Zudem werden die eingeführten Konzepte weiter vertieft. In Kapitel 6 wird am Beispiel von IoT behandelt, wie Querschnittsdomänen in die Entwicklung mit einbezogen werden, bzw. wie die Funktionalität aus der Anwendung herausgezogen wird. Die Überführung der aufgestellten Artefakte aus der Analyse und dem Entwurf wird in Kapitel 7 in die Implementierung und Tests überführt. Kapitel 8 demonstriert die Tragfähigkeit der Konzepte anhand verschiedener Anwendungen, die mittels dem Ansatz umgesetzt wurden. Zum Abschluss erfolgt in Kapitel 9 eine Zusammenfassung der Arbeit und ein Ausblick über weitere Arbeiten, die auf dieser Arbeit aufsetzten können wird gegeben.

2 Grundlagen

Dieses Kapitel erfasst die Grundlagen, welche für das Verständnis der Arbeit erforderlich sind. Es beginnt in Abschnitt 2.1 grundlegend mit den Inhalten der Softwareentwicklung. Darauf aufbauend sind die wichtigsten verwendeten Artefakte bzgl. der Analyse und den Anwendungsfällen in Abschnitt 2.2.1 weiter erläutert. Die Architektur und Entwurfsartefakte, die bei der Überführung in den Entwurf verwendet werden, sind in Abschnitt 2.3 beschrieben. Bei der Modellierung der Domäne spielt insbesondere das Modell eine wichtige Rolle, da das Domänenmodell das Geschäftswissen abbildet. Um das Modell zu erhalten, wird der domänengetriebenen Entwurf genutzt. Hierzu werden auch die verwendeten Architekurbegriffe in Abschnitt 2.4 geklärt. Insbesondere die modellgetriebene Architektur und die Microservice-Architektur bilden eine wichtige Grundlage. Der Bezug zur IoT-Domäne wird in Abschnitt 2.5 mit dem IoT-Standard der SensorThingsAPI erklärt. Für die anschließende Implementierung und Tests werden in Abschnitt 2.6 auf die Grundlagen der verschiedenen Tests eingegangen, die in dieser Arbeit in der Systematik verwendet werden.

2.1 Softwareentwicklung

Der in dieser Arbeit vorgestellte systematische domänengetriebene Ansatz zur Entwicklung von fortgeschrittenen Web-Anwendungen baut auf den bekannten Phasen der Softwareentwicklung auf. Die Softwareentwicklung umfasst hierbei verschiedene Vorgehensmodelle [BK15], die den Ablauf des Projekts beschreiben. Grundsätzlich sind die Phasen der Analyse, des Entwurfs, die Implementierung und dem Testen, sowie der Betrieb die gängigen Phasen. Die in dieser Arbeit wichtigste Phase ist der Entwurf, bei der das Domänenmodell entworfen wird und wiederverwendbare Microservices etabliert werden. Für die gezielte Extraktion des notwendigen Domänenwissens spielen auch die Anforderungen der zu entwickelnden Anwendung eine Rolle, weshalb die verwendeten Konzepte zu dieser ebenfalls kurz behandelt werden.

2.2 Analyse

Zu der Analyse gehören neben der Erfassung der Anforderungen weitere Tätigkeiten [Po10]. Neben dem Erfassen der Anforderungen ist auch deren Dokumentation von zentraler Bedeutung. Ein weiterer wichtiger Punkt ist, dass die Anforderungen von den verschiedenen Stakeholdern akzeptiert und

verwaltet werden, so dass bei Änderungen Anpassungen gut durchführbar sind. In dieser Arbeit ist insbesondere die Erfassung und Dokumentation der Anforderungen mittels Anwendungsfälle wichtig.

2.2.1 Anwendungsfälle

Anwendungsfälle (engl. use cases) spezifizieren die funktionalen Anforderungen. Diese geben an, was das System zu erfüllen hat [La04]. Prinzipiell wird ein Vertrag zwischen dem Benutzer und dem System festgelegt, wobei das System als gesamtes (sprich als Blackbox) betrachtet wird. Ein Anwendungsfall besteht hierbei aus verschiedenen Inhalten, die in der Literatur tiefer behandelt wird [Co01].

Ein wichtiger Bestandteil ist ein primäre Akteur ist ein Akteur, dessen Ziel der Anwendungsfall zu erfüllen versucht, mit anderen Worten, der Akteur, dessen Aktion den Anwendungsfall initiiert. Es kann mehrere Akteure geben, die dasselbe Ziel in Bezug auf einen bestimmten Anwendungsfall verfolgen.

Sekundäre Akteure können andere menschliche oder institutionelle Akteure sein, die bei der Ausführung des Anwendungsfalls unterstützende Rollen einnehmen. Diese Akteure können nicht-menschliche Akteure sein, wie z. B. externe Systeme wie Systeme, welche die Daten über Schnittstellen bereitstellen. Die Interaktionen mit den sekundären Akteuren werden nicht explizit spezifiziert, wenn die Interaktionen für den primären Akteur nicht direkt relevant oder sichtbar sind.

Eine Vorbedingung ist der Zustand des Systems und seiner Umgebung, der erforderlich ist, bevor der Anwendungsfall durchgeführt werden kann. Eine Nachbedingung ist der Zustand des Systems, nachdem der Anwendungsfall durchlaufen ist. Zusammen definieren diese Bedingungen den Vertrag zwischen dem primären Akteur und dem System.

Durch die Abläufe (engl. flows) werden die notwendigen Sequenzen beschrieben welche benötigt werden um das gewünschte Ergebnis zu erreichen. Neben dem Standardablauf werden oftmals auch alternative Abläufe betrachtet, die ebenfalls zum Ziel führen. Zusätzlich werden aber auch Fehlerfälle betrachtet, die zum Abbruch des Ablaufs führen.

2.3 Architektur und Modellierung

Dieser Abschnitt beschäftigt sich mit den Grundlagen zur Architektur in Richtung der verteilten Systeme, wie der Microservice-Architektur.

2.3.1 Begriff des Modells

In den betrachteten Ansätzen und dem Entwurf wird ein Modell in den Vordergrund gestellt. Ein Modell hat laut Stachowiak drei grundlegende Eigenschaften [St73], die in dieser Arbeit ebenfalls zutreffen. Die erste Eigenschaft ist, dass ein Modell eine Abbildung von etwas ist. Hierzu kommt, dass die Abbildung des Modells nicht vollständig ist, sondern nur die betrachteten Eigenschaften modelliert werden. Diese Eigenschaft wird als Verkürzung bezeichnet. Die dritte Eigenschaft ist der Pragmatismus. Hierbei geht es darum, dass ein Modell für bestimmte Intentionen ausgelegt ist. Im Sinne der Erschließung des Domänenwissens in dieser Arbeit ist das Modell immer auf eine bestimmte Domäne zugeschnitten.

2.3.2 Domänengetriebener Entwurf

Der domänengetriebene Entwurf (engl. Domain-Driven Design) beschreibt Konzepte und Ansätze zur Modellierung von Software-Systemen. Bekannt wurde der domänengetriebene Entwurf (DDD) durch Evans. In seinem Buch: "Domain-Driven Design: Tackling Complexity in the Heart of Software" zeigt Evans, wie komplexe Domänen zu bewältigen sind. DDD liefert hierbei eine Palette an Mustern, um eine Domäne zu verstehen und sie in einem Modell zu manifestieren. Im Kontext dieser Arbeit wird eine Teilmenge der Muster und Begriffe aus DDD verwendet. Eine Domäne beschreibt ein abgrenzbares Problemfeld, welches in einem weiteren Schritt durch ein Software-System unterstützt werden soll [DG+13]. Der Schwerpunkt des domänengetriebenen Entwurfs liegt bei der Analyse der Domäne und deren Konzepte und bezieht sich auf eine Geschäftsdomäne. Ohne ein tieferes Verständnis der Domäne besteht die Gefahr, dass das entwickelte System nicht den Anforderungen des Kunden entspricht, da Konzepte falsch verstanden werden und die Kommunikation aufgrund sprachlicher Unterschiede zu unzureichendem Verständnis führen. Hierzu bietet der domänengetriebene Entwurf Prinzipien, Muster und Aktivitäten an, welche es ermöglichen ein tiefes Verständnis der komplexen Domänen bei den Softwareentwicklern zu etablieren.

Ein weiteres Ziel des domänengetriebenen Entwurfs besteht darin, eine enge Kollaboration zwischen Entwicklern und Domänenexperten zu etablieren und den Wissensaustausch zwischen ihnen zu fördern. Bei einem Domänenexperten handelt es sich um eine Person, die sich in seiner Domäne bestens auskennt, da sich die Person in ihrem täglichen Umfeld ständig mit der Domäne beschäftigt und dadurch die Abläufe und Strukturen kennt. Um den Wissensaustausch zu fördern, wird eine ubiquitäre Sprache etabliert, welche von allen Beteiligten gesprochen werden soll. Die ubiquitäre Sprache ist notwendig, da Entwickler und Domänenexperten ihre eigenen sprachlichen Ausdrücke haben. Ein weiteres Problem ist, dass Entwickler mitunter kein oder kaum Verständnis über die Domäne haben. Das unterschiedliche Vokabular stellt hier ein weiteres Hindernis dar, da die Domänenkonzepte erst übersetzt werden müssten. Bei den Domänenkonzepten handelt es sich um die Strukturen und Abläufe

die innerhalb der Domäne geschehen. Um ein gemeinsames Verständnis über die Domäne zu erhalten, ist es unabdingbar, dass die Entwickler die gleiche Sprache wie die Domänenexperten sprechen, da sonst Missverständnisse über Domänenkonzepte entstehen können. Die ubiquitäre Sprache wird aber nicht nur von allen Beteiligten gesprochen, sondern fließt in das Modell und die Implementierung mit ein. Dies bedeutet, dass das Modell und der Code an die ubiquitäre Sprache gebunden werden und die Domänenkonzepte in der Implementierung ausgedrückt werden. Die ubiquitäre Sprache hat somit von der Domäne bis hin zur Implementierung einen großen Einfluss. Die enge Kopplung der ubiquitären Sprache an das Modell hat zur Folge, dass Änderungen an der ubiquitären Sprache auch zu Änderungen am Modell und somit auch an der Implementierung führen. Damit eine ubiquitäre Sprache aufgebaut werden kann, ist ein Wissensaustausch zwischen Entwicklern und Domänenexperten notwendig. Der Wissensaustausch erfolgt beim domänengetriebenen Entwurf durch das sogenannte "Knowledge Crunching". Typischerweise wird das Knowledge Crunching von den Entwicklern geleitet. Durch Fragen an die Domänenexperten oder durch die Nutzung anderer Techniken wird das Domänenwissen gewonnen.

In Abbildung 2.1 ist der Aufbau der Schichtenarchitektur dargestellt. In dieser werden die verschiedenen Aspekte des Software-Systems durch sogenannte Schichten voneinander abgetrennt. Auf der obersten Ebene befinden sich die Aspekte der Benutzerschnittstelle. Die Darstellung der Funktionalitäten für die Benutzer und die möglichen Eingabeoptionen wird in dieser Präsentationsschicht gehandelt. Die Schichtenarchitektur wird insbesondere zur Einordnung der Domänen- und Anwendungs-Microservices in Abschnitt 4.3.2 verwendet.

Eine Ebene tiefer befindet sich die Anwendungsschicht. Die Anwendungsschicht dient zur Koordination der Domänenschicht. Bei einer serviceorientierten Architektur geschieht in der Anwendungsschicht die Orchestrierung der Domänenschicht. Darunter ist die Domänenschicht angesiedelt. Auf der untersten Ebene befindet sich die Ebene der Infrastruktur. Diese Schicht ist für die Datenhaltung zuständig. Ein wichtiger Punkt an dieser Architektur ist, dass die Belange der unterschiedlichen Schichten von der Domäne abgegrenzt werden. Die Domänenschicht realisiert die fachlichen Konzepte der Domäne. Dadurch wird die Domäne von den anderen Belangen isoliert. Eine lose Kopplung des Systems wird erreicht, da die Abhängigkeiten innerhalb des Systems reduziert werden. Innerhalb der Schichten wird eine hohe Kohäsion erreicht. Weiterhin lässt sich die Domäne extrahieren und eine Wiederverwendbarkeit des Domänenwissens wird ermöglicht. Bei der Schichtenarchitektur die Evans vorsieht, ist nur die Kommunikation innerhalb einer Schicht und die Kommunikation von höheren Schichten zu tieferen Schichten gestattet.

Modellierung der Domäne

Ein wichtiger Aspekt des domänengetriebenen Entwurfs ist die Fachlichkeit und die Fachlogik der Domäne auszudrücken. Dieses geschieht anhand eines Modells, welches als Domänenmodell

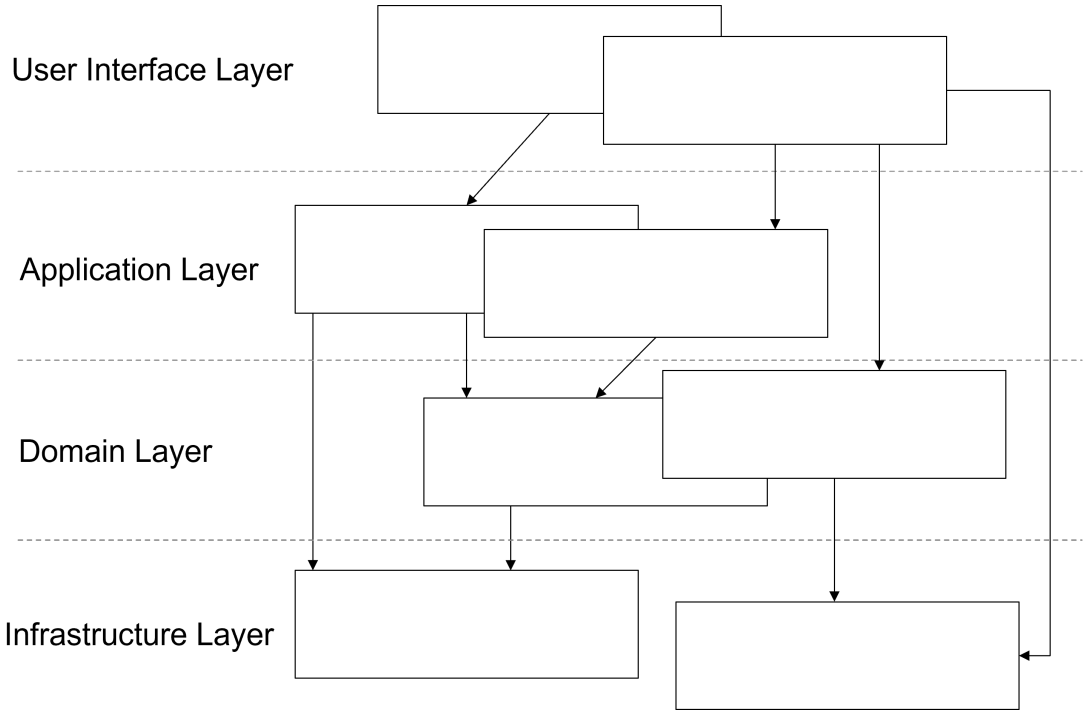


Abbildung 2.1: Layered Architecture aus DDD [Ev04]

bezeichnet wird. Es beschreibt die strukturellen und dynamischen Aspekte der Domäne, die für das zu entwickelnde Software-System benötigt werden. Das Domänenmodell ist nach Evans nicht nur eine bestimmte Art von Diagramm, sondern es geht um die Idee, welche durch die Diagramme ausgedrückt wird. Wie das Domänenmodell dargestellt wird, ist nicht festgelegt. Jegliche Diagramme und Skizzen sind geeignet, um die Domänenkonzepte auszudrücken, solange sie zum besseren Verständnis der Domäne dienen.

> "A domain model is not a particular diagram; it is the idea that the diagram is intended to convey." [Ev04]

Der domänengetriebene Entwurf geht auf verschiedene Bestandteile, Konzepte und Muster ein, die das Domänenmodell formen. Hierbei wird keine konkrete Formalisierung angestrebt. Domänenobjekte beschreiben die Elemente, die zur Domäne gehören. Durch Assoziationen zwischen den Domänenobjekten kann der Bezug des Domänenwissens dargestellt werden. Ein Domänenobjekt kann ein Wertobjekt (eng.: value object), eine Entität (engl. entity) oder ein Service sein. Bei Entitäten handelt es sich um Objekte des Modells, welche durch ihre eindeutige Identität definiert werden. Dadurch können die Entitäten eindeutig referenziert werden. Hat ein System den Bedarf einer Person eine Identität zuzuweisen, so lässt sich in diesem Fall eine Person als Entität abbilden. Attribute einer Entität können sich ständig ändern, aber die Identität einer Entität bleibt den ganzen Lebenszyklus die gleiche. Zwei Entitäten mit den gleichen Attributen unterscheiden sich dennoch auf Grund ihrer

Identität. Wertobjekte hingegen besitzen keine konzeptionelle Identität. Dies bedeutet, dass diese nur durch ihre Eigenschaften definiert werden. Dies bedeutet, Wertobjekte haben keine Kennung (engl. Identifier). Sie sind zur Beschreibung domänenrelevanter Attribute von Entitäten gedacht. Wertobjekte sind unveränderlich und wiederverwendbar und sollten (bis auf sehr wenige Ausnahmen) unveränderbar sein, damit die Wiederverwendbarkeit gewährleistet wird. Da Wertobjekte keine Kennung besitzen, können diese einfach erstellt und gelöscht werden. Bei der Modellierung des Domänenmodells muss entschieden werden, ob ein Domänenobjekt als Entität oder als Wertobjekt modelliert wird.

Strategischer Entwurf

Durch Entwurfstechniken wie den Bounded Contexts werden die Grenzen des Kontexts bestimmt. Bounded Contexts beschränken somit den Gültigkeitsbereich des Kontexts auf einen bestimmten Bereich ein [Ve13]. So ist die ubiquitäre Sprache nur im Rahmen des definierten Bounded Context gültig. Über die Context Map lässt sich ein Überblick der einzelnen Bounded Contexts gewinnen. Sie liefert den Beteiligten ein Verständnis über das Gesamtsystem, indem alle Bounded Contexts und deren Beziehung zueinander dargestellt werden. Verschiedene Bounded Contexts können unabhängig voneinander entwickelt werden, da diese jeweils einen anderen Kontext definieren und anhand von Schnittstellen miteinander kommunizieren. Bounded Contexts können Schnittstellen bereitstellen, die wiederum von anderen Bounded Contexts genutzt werden können. Die Kommunikation zwischen den Bounded Contexts erfolgt lose gekoppelt über diese Schnittstellen, die beispielsweise mit den Prinzipien wie REST (Representational State Transfer) oder SOAP (Simple Object Access Protocol) umgesetzt werden.

2.3.3 Modellgetriebene Architektur

Bei der modellgetriebenen Architektur (engl. Model-Driven Architecture, MDA) steht das Modell im Vordergrund. Diese Modelle werden im Rahmen der Softwareentwicklung zur Beschreibung des zu entwickelnden Softwaresystems verwendet. Hierbei werden verschiedene Abstraktionsebenen eingesetzt, welche durch Transformationen ineinander überführt werden. Jede Ebene stellt eine bestimmte Sichtweise auf das Systems dar [MS+02]. Weiterhin lassen sich die Modelle über ein Metamodell formalisieren. Abbildung 2.2 zeigt die verschiedenen Modelle und deren Zusammenhang. Oben befindet sich das Computation Independent Model (CIM), welches die Anforderungen an das umzusetzende System festhält.

Im zweiten Schritt wird das Modell der Anforderungen anhand der Transformationsregeln auf ein plattformunabhängiges Modell (engl. Platform Indipendent Model, PIM) überführt. Für das PIM eignet sich beispielsweise die Modellierungssprache UML als Repräsentation für die Modelle [SS09].

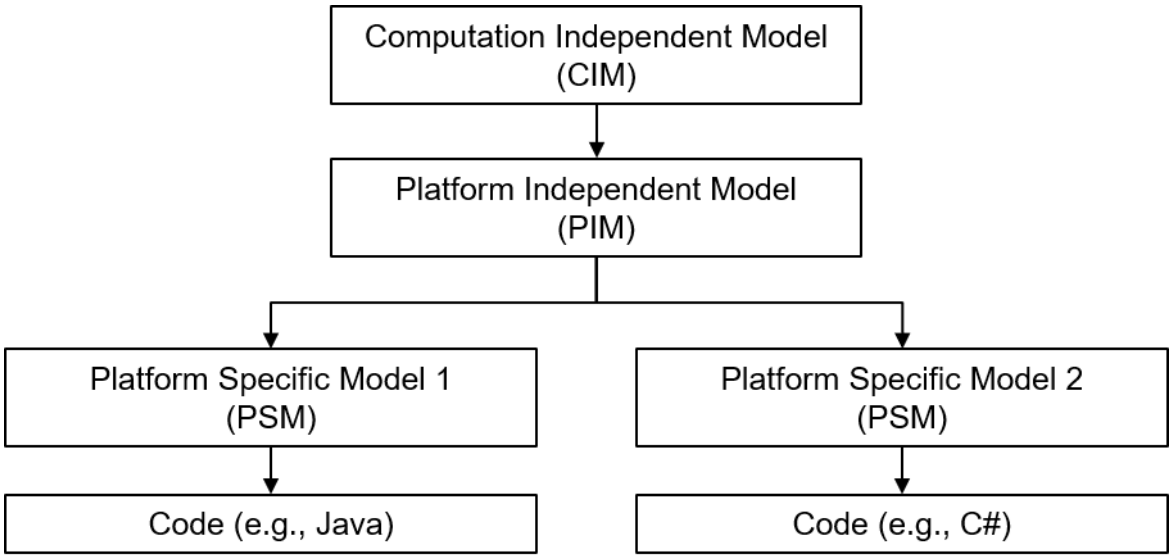


Abbildung 2.2: Modell-getriebene Architektur und die Transformationen [Ev04]

Die Transformation an sich muss nicht automatisiert durchgeführt werden, sondern kann auch teilautomatisiert oder manuell durchgeführt werden [KM05]. Anschließend wird dieser Entwurf in das entsprechende plattformabhängiges Modell für die Architektur (engl. Platform Specific Model, PSM) überführt, bevor dieser durch eine spezifische Implementierung umgesetzt wird. Die Implementierung findet hierbei in der gewünschten Implementierungssprache statt.

2.4 Klärung der Architekturbegriffe

Da es keine standardisierten Begriffsdefinitionen gibt, werden in diesem Abschnitt die verwendeten Begriffe kurz erläutert. Als Beispiel listet [CMU-SA] mehr als dreißig verschiedene Definitionen für eine Softwarearchitektur auf.

Architekturstile sind anwendungsunabhängige Lösungsprinzipien, die im gesamten Projekt verwendet werden. Die Unified Modeling Language (UML) bietet verschiedene Arten von Diagrammen, um die verschiedenen Arten von Architekturen formal als Modelle zu spezifizieren [BD09].

Die Softwarearchitektur betrachtet das System aus logischer Sicht, indem sie es in logische Komponenten unterteilt. Sie physikalische Sicht wird durch die Systemarchitektur auf ein System eingenommen und beschreibt dessen Struktur, die aus Netzwerk- und Hardwarekomponenten besteht. Darüber hinaus werden ihre Eigenschaften und Beziehungen zueinander sowie zu ihrer Umgebung und anderen Systemen dargestellt. Eine mögliche Modellierungsart mit UML ist das Verteilungsdiagramm.

Die Makroarchitektur dient dazu, die Prinzipien der Teilsysteme, z. B. Microservices oder Module, auf einer Blackbox-Ebene zu beschreiben. Die Mikroarchitektur befasst sich mit der internen Struktur eines einzelnen Teilsystems. Die Implementierungsarchitektur beschreibt die Struktur des Systems aus technischer Sicht. Dazu gehören Pakete, Bibliotheken und Rahmenwerke.

2.4.1 Microservice-Architektur

Bei einer Microservice-Architektur handelt es sich um einen Architekturstil, der laut Zimmermann [Zi17] eine Ausprägung der Service-orientierten Architektur (SOA) ist.

Eine erste Definition dieses Architekturstils haben Fowler und Lewis [FL14] veröffentlicht:

> "The microservice architectural style is an approach to developing a single application as a suite of small services, each running in its own process and communicating with lightweight mechanisms, often an HTTP resource API. These services are built around business capabilities and independently deployable by fully automated deployment machinery. There is a bare minimum of centralized management of these services, which may be written in different programming languages and use different data storage technologies."

Bei einer Microservice-Architektur werden hauptsächlich Microservices als Systembausteine eingesetzt. Diese Microservices werden diese Systembausteine eingesetzt um eine funktionierende Anwendung zu erstellen [DG+17]. Dabei sind Microservices kleine, autonome Services, die sich gezielt auf eine Aufgabe konzentrieren und diese gut lösen. Sie sind deshalb dafür geeignet, das Single Responsibility Principle umzusetzen [Ma09]. Microservices können von einem kleinen Entwicklungsteam entwickelt und verwaltet werden. Ein Microservice ist autonom, was bedeutet, dass er als lose gekoppelter Service eingesetzt werden kann. Da ein Microservice sich gezielt auf eine Sache fokussiert, ist die Wartung und Anpassung dieses Microservice deutlich einfacher als bei einem System, welches als unstrukturierter Monolith entwickelt wurde. Soll das System um neue Funktionalitäten erweitert werden, so kann dies problemlos durch die Implementierung eines oder mehrerer Microservices geschehen. Ein Microservice muss so entwickelt werden, dass sich bei Erweiterungen oder Änderungen an diesem, keine Änderungen an anderen Microservices nachziehen. Die Abhängigkeiten der Microservices werden durch die API definiert. Bei einer API handelt es sich um eine Schnittstelle zur Anwendungsprogrammierung. Wegen der Verteilung der Funktionalität auf verschiedene Microservices ist es wichtig, dass die API systematisch entworfen wird, da die Microservices über diese kommunizieren. Die API gibt nur die Schnittstellen eines Microservices preis und versteckt dessen Implementierungsdetails.

Ein Microservice implementiert im Sinne des domänengetriebenen Entwurfs einen Bounded Context, da ein Bounded Context eine bestimmte Fachlichkeit aus der Geschäftsdomäne darstellt [Ne15].

Jeder Bounded Context lässt sich durch unabhängige Teams implementieren. Dadurch lässt sich die Entwicklung des Systems durch den Einsatz mehrerer Teams skalieren. Die Implementierung einzelner Microservices bringt zusätzlich den Vorteil der kontinuierlichen Auslieferung (engl. Continuous Delivery). Jede implementierte Geschäftsfunktion kann dadurch einzeln ausgeliefert werden. Ggf. müssen die notwendigen Schnittstellen zu anderen Geschäftsfunktionen vorhanden sein. Eine Schnittstelle kann hier Upstream oder Downstream sein. Ein Upstream bietet Funktionalität an, während ein Downstream Funktionalität konsumiert. Ändert sich etwas im Downstream, bleibt der Upstream davon unberührt. Wird hingegen am Upstream etwas geändert, so ist nicht sichergestellt, dass der Downstream nicht betroffen ist.

2.4.2 Integrationsplattformen und Integrationsarchitektur

Cloud-Dienstleister stellen Plattformen zur Integration von bereits bestehenden Systemen bereit. Bei den sogenannten Integrationsplattformen (Integration Platform as a Service, iPaaS) [EW+17] werden verschiedene Dienste für die Anbindung der externen Systeme, die Aufbereitung der Daten und die Bereitstellung der Daten an eine entsprechende Benutzerschnittstelle vorgesehen. Durch eine entsprechende Architektur wird dieses Konzept unterstützt. Beispielsweise wird beim konkreten iPaaS-Produkt von MuleSoft der sogenannte API-led Connectivity-Ansatz [Mul-API-led] publiziert. Abbildung 2.3 zeigt einen Überblick über die Architektur.

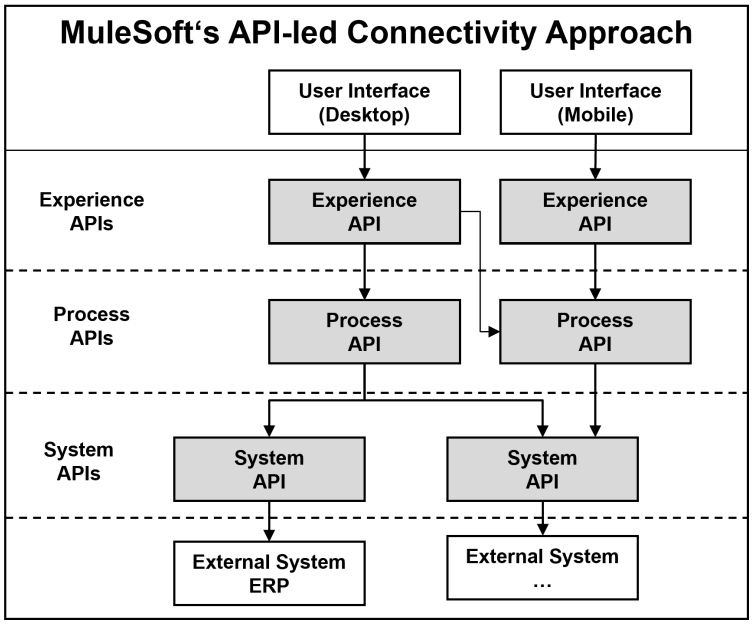


Abbildung 2.3: Architektur des API-led-Connectivity-Ansatzes [Ev04]

Die verschiedenen Bausteine sind einzelne Services, die anhand einer REST API die gewünschte

Funktionalitäten bereitstellt. Es wird zwischen System APIs, Process APIs und Experience APIs unterschieden:

- **System API**
  Die System APIs dienen zur Anbindung der externen Systeme. Damit wird eine Isolierung der Komplexität oder den Änderungen an den zugrunde liegenden Systemen vorgenommen. Die externen Datenquellen werden auf REST APIs abgebildet, so dass die Anbindung der Daten vereinfacht wird. Dadurch erlaubt die System API den Entwicklern auf die benötigten Daten zugreifen, ohne die Details der zugrunde liegenden Systeme kennen zu müssen. System-APIs sind in der Regel feinkörnig, unabhängig vom Geschäftsprozess und wiederverwendbar. Ein Beispiel ist hier eine Anbindung eines ERP-Systems über eine System-API, die die Daten bereitstellt.

- **Process API**
  Die Process APIs wiederum setzen die Geschäftslogik um, indem die darunterliegenden Daten aus den System APIs gemäß dem Geschäftsprozess orchestriert und verarbeitet werden. Hierbei wird die benötigte Logik ergänzt.

- **Experience API**
  Die Experience APIs ähneln den Prozess-APIs insofern, als sie die Inhalte, Merkmale und Funktionen mehrerer anderer APIs zusammenfassen. Im Gegensatz zu Prozess-APIs sind Experience APIs jedoch spezifischer an einen bestimmten Geschäftskontext gebunden und bereiten die Daten für eine bestimmte Benutzerschnittstelle (Desktop, Mobil) auf.

2.4.3 Schnittstelle, Web-API und API-Spezifikation

Eine Schnittstelle (engl. Application Programming Interface, API) beschreibt im Allgemeinen eine Schnittstelle, die eine Softwareanwendung so zur Verfügung stellt, dass sie in eine andere Softwareanwendung integriert werden kann. Dieser Begriff existierte bereits vor der Ära der Microservices, beispielsweise bei ESDL oder SOAP [MP+10]. Ein bekanntes Paradigma zur Beschreibung von Web-APIs ist der REpresentational State Transfer (REST). Neben REST existieren auch weitere Ansätze für Web-APIs, die sich basierend auf der eingesetzten Architektur und das verbundenen Paradigma einsetzen lassen [SS+19]. Dies sind zum einen die sogenannten Remote Procedure Calls (RPCs), die funktionsorientiert angelegt sind. Zum Weiteren existiert das Paradigma der ereignisgesteuerten Architektur. Im Zusammenhang mit Microservices werden die APIs über das Web zugänglich. Für die Definition von Web-APIs sind die Konzepte von REST weit verbreitet und sind für die Gestaltung der Web-APIs in dieser Arbeit im Fokus. REST baut auf den folgenden fünf Konzepten auf [Fi00]:

- Adressierbarkeit von Ressourcen: Ressourcen sind das Hauptkonzept und die grundlegende Abstraktion von Informationen in REST. Jede Ressource wird durch einen Uniform Resource Locator (URL) angesprochen.

- Einheitliche Schnittstelle: Es wird nur eine Anzahl definierter Methoden zur Bearbeitung der Ressourcen zur Verfügung gestellt.

- Zustandslose Kommunikation: Zustandslos bedeutet, dass der Server keine Sitzungsinformationen hält. Das bedeutet, dass der Client alle benötigten Informationen in der Anfrage mit übertragen muss, die der Server zur Bearbeitung dieser Anfrage benötigt. Der Vorteil der sich hiermit ergibt ist die zustandslose Kommunikation und eine höhere Skalierbarkeit.

- Repräsentationen von Ressourcen: Eine Ressource kann je nach Plattform durch verschiedene Formate repräsentiert werden. Beispiele für Formate und entsprechende Plattformen sind unter anderem HTML für den Browser und JSON für JavaScript-Anwendungen.

- Formatgesteuerter Zustandstransfer (Hypermedia-Konzept): Dieses Prinzip steht im Zusammenhang mit Hypermedia und dem sogenannten "Hypermedia as the Engine of Application State" (HATEOAS). Dabei wird das Konzept der Hypermedia (insbesondere der Hypermedia-Links) zur Steuerung der Zustandsübergänge einer Anwendung verwendet.

Für die Spezifikation einer Web-API ist OpenAPI [OAI-Wha] ein akzeptiertes Beschreibungsformat. Dieses dient der standardisierten Beschreibung von Web-APIs. Üblicherweise wird die Spezifikation in JSON- oder YAML-Dokumenten festgehalten. Jedes dieser Dokumente besteht das aus drei Teilen: Im ersten Teil " info" werden Metainformationen ausgedrückt. Der zweite Teil "paths" spezifiziert die Endpunkte und zugehörigen Methoden. Der dritte Teil "definitions" enthält wiederverwendbare Objekte, die über die API-Spezifikation hinweg genutzt werden können. Insbesondere im Zusammenhang mit der API-Spezifikation (siehe Abschnitt 4.3.2) wird die API nach diesem Beschreibungsformat spezifiziert.

2.5 Verknüpfung zu IoT und Standards

Bei der Erfassung der Querschnittsdomäne wird das Thema "Internet der Dinge" fokussiert. Hierbei spielt insbesondere der Standard der sogenannten SensorThings API eine wichtige Rolle, weshalb dieser kurz erläutert wird.

2.5.1 SensorThings API

Die SensorThings API (Application Programming Interface) ist ein offener Standard des OGC (Open Geospatial Consortium) [OGC-STA-Sen, OGC-STA-Tas]. Der Ziel des Standards ist es, IoT-Geräte und Anwendungen über das Web miteinander zu verbinden. Der Standard befasst sich mit der syntaktischen und semantischen Interoperabilität.

Die Schnittstelle des SensorThings-API-Standards basiert auf dem Hypertext Transfer Protocol (HTTP) und REST und wird zum Austausch der Daten über das Internet verwendet. REST (REpresentational State Transfer) ist ein Konzept, das die Definition von Webdiensten auf der Grundlage der HTTP-Operationen ermöglicht. Durch die Verwendung von REST können IoT-Geräte und IoT-Anwendungen IoT-Daten und Metadaten erstellen, lesen, aktualisieren und löschen.

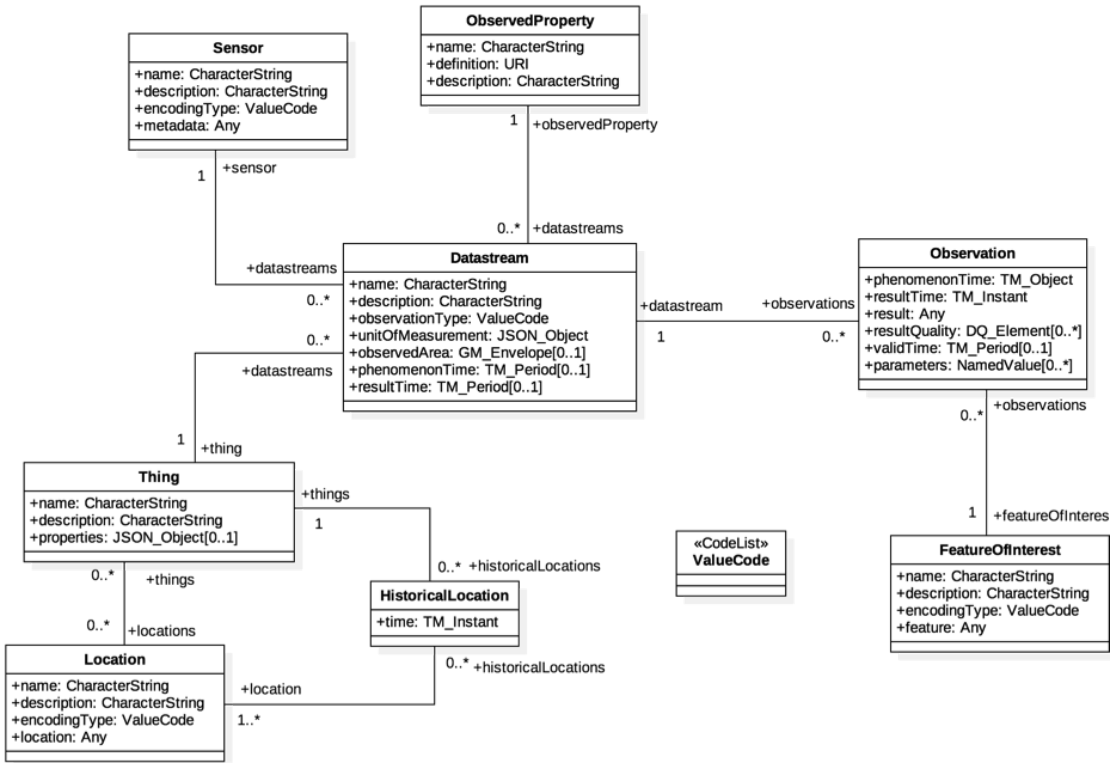


Abbildung 2.4: Die Entitäten der SenorThings API der OGC [OGC-STA-Sen]

Das Datenmodell kann als der Kern der SensorThings API angesehen werden. Abbildung 2.4 zeigt eine Übersicht des Datenmodells. Dieses definiert, wie Sensorinformationen in einer eindeutigen und herstellerunabhängigen Weise beschrieben werden sollten. Das Datenmodell besteht aus verschiedenen Entitäten (wie beispielsweise Thing, Datastram und Observation).

2.6 Tests und Testarten

Testen ist eine Tätigkeit, mit der sichergestellt werden soll, dass eine Software frei von Fehlern ist und sich die Software so verhält, wie es die Anforderungen vorgeben [Wh00]. Es ist wichtig, dass die Fehler eines Softwaresystems durch Tests aufgedeckt werden, um Abstürze und Ausfälle zu verhindern. Neben möglichen Fehlern ist das korrekte Verhalten der Software zu Testen, damit die Anforderungen des Kunden zu erfüllt werden. Daher werden beim Testen nicht nur Fehler erfasst, sondern auch die spezifizierten Anforderungen geprüft. Das Etablieren solcher automatisierter Tests erfordert Kenntnisse über das Softwaresystem. Es ist wichtig, dass sich die Entwickler der Probleme bewusst sind, die im Softwaresystem auftreten können. Insbesondere dürfen die Tests keine Experimente sein, bei denen mögliche Fehler in der Implementierung der Anwendung erraten werden. [RJ20]. Das Testen von Microservices unterscheidet sich vom Testen monolithischer Anwendungen [Cl14]. Eine Microservice-basierte Anwendung erfordert aufgrund der autonom einsetzbaren Microservices eine andere Teststrategie. Insbesondere die Integrations- und Ende-zu-Ende-Tests geraten in einen stärkeren Fokus [FS19].

2.6.1 Testpyramide

Die verschiedenen Testebenen unterscheiden sich durch den Umfang und die Ebenen, die ein Testfall abdeckt. Unit-Tests testen meist ein einzelnes Stück Code, während Ende-zu-Ende-Tests (E2E-Tests) die ganze Anwendung durchlaufen können. Da in der Literatur die Begrifflichkeiten unterschiedlich genutzt werden erfolgt eine Klärung der Begrifflichkeiten, wie diese in dieser Arbeit im Rahmen des adressierten Testkonzeptes (siehe Kapitel 7 verwendet werden.

Die in Abbildung 2.5 dargestellte, sogenannte Testpyramide [Co10] verdeutlicht die verschiedenen Test-Stufen. E2E-Tests sind teurere Tests, da die Ausführung dieser am längsten benötigt [Fo12]. Im Vergleich hierzu sind Unit-Tests deutlich schneller und kostengünstiger.

2.6.2 Unit-Tests

Mit Unit-Tests werden die kleinsten Teile eines Softwaresystems auf ihre Funktionalität getestet. Damit sind unter anderem einzelne Einheiten des Quellcodes, wie Funktionen, Module oder Objekte gemeint. Unit-Tests sind eine Form von White-Box-Tests, die darauf abzielen, einen Code-Abschnitt auf seine Korrektheit hin zu überprüfen. Hierdurch soll unter anderem sichergestellt werden, dass die bisherige Funktionalität durch Änderungen am Code, sowie die Implementierung neuer Funktionalität nicht beeinträchtigt wird [Kh20]. Beliebte Metriken für Unit-Tests sind die Pfadabdeckung und Anweisungsabdeckung [Re19]. In der Microservice-Architektur sollen diese Tests die interne Funktionalität

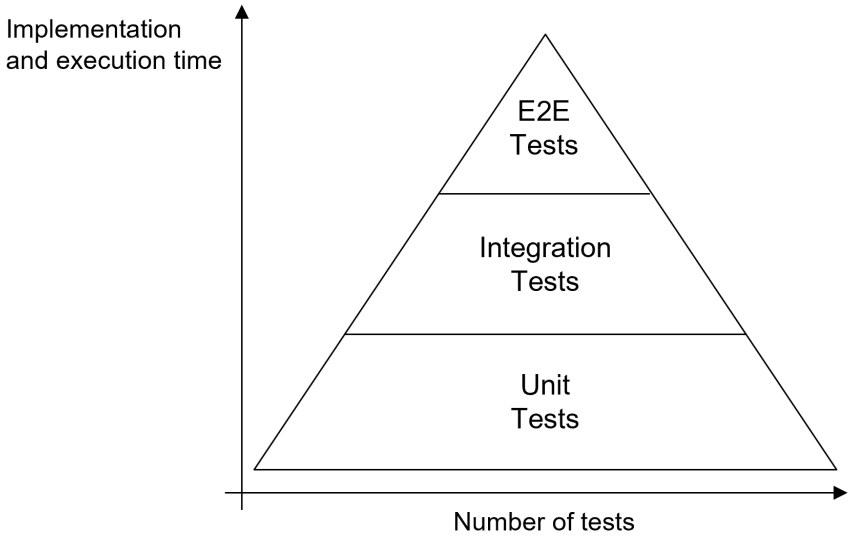


Abbildung 2.5: Testpyramide

eines Microservices überprüfen. Die Unit-Tests eines Microservices sind unabhängig von externen Komponenten wie bspw. andere Microservices. In der CI/CD-Pipeline werden die Unit-Tests direkt nach der Build-Phase im Test-Schritt ausgeführt.

2.6.3 Integrationstests

Integrationstests verifizieren die angebotenen Schnittstellen und Interaktionen zwischen Komponenten, um Defekte darin zu erkennen [Cl14]. Integrationstests fassen im Gegensatz zu Unit-Tests Software-Einheiten zusammen und testen diese als Subsystem, um fehlerhafte Interaktionen zu erkennen. Im Rahmen der Microservice-Architektur werden hier die Schnittstellen der einzelnen Microservices angesprochen.

2.6.4 Ende-zu-Ende-Tests

Bei Ende-zu-Ende-Tests wird der komplette Satz von Benutzerabläufen getestet, um sicherzustellen, dass sich Ihre Software wie vorgesehen verhält. Hierbei ist das Ziel, die Software aus der Perspektive eines Benutzers zu testen. Typischerweise beginnen die Tests mit einem sauberen System und führen die Interaktionen eines Benutzers durch. Gerade in einer Microservice-Architektur sind diese Tests besonders wichtig, da hierdurch auch das Zusammenspiel der verschiedenen beteiligten Services überprüft wird. Dadurch ist sichergestellt, dass eine systemübergreifende Überprüfung durchgeführt wird [Fr15b].

2.6.5 Weitere Typen von Softwaretests

Die Literatur umfasst in Bezug auf die Microservice-Architektur verschiedene weitere Testarten, die in dieser Arbeit nicht tiefer behandelt werden. Die Studie von Waseem et al. [WL+20] umfasst, welche konkreten Tests tiefer verfolgt werden. Im Folgenden werden kurz weitere Testarten beispielsweise angedeutet.

Regressionstests

Erweiterungen und Anpassungen an der Software sollten nicht zu Änderungen an der bestehenden Funktionalität führen. Regressionstests stellen sicher, dass Änderungen das bisherige Softwareverhalten nicht beeinflussen.

Smoke Testing

Der Name Smoke Testing leitet sich von Wasserinstallationen ab. Er steht für die Prüfung, ob nach einem Einsatz Rauch aus den Rohren kommt. Wie bei der Wasserinstallation wird das Smoke Testing in Softwareprojekten nach dem Einsatz durchgeführt.

Fuzz Testing

Beim Fuzz Testing werden zufällige und ungültige Daten zum Testen der Software verwendet. Das Ziel ist die Analyse von Abstürzen und Ausnahmen.

3 Stand der Forschung

Dieses Kapitel erfasst den Handlungsbedarf gestützt auf dem aktuellen Forschungsstand. Hierzu wird in Abschnitt 3.1 ein Anforderungskatalog aufgestellt. Dieser zeigt, welche Anforderung die domänengetriebene Entwicklung von fortgeschrittenen Web-Anwendungen in der vorliegenden Arbeit aufwirft. In Abschnitt 3.2 werden basierend auf den Anforderungen Publikationen betrachtet, die sich mit gleichen oder ähnlichen Fragestellungen beschäftigen. Im Rahmen des Anforderungskatalogs werden diese Publikationen bewertet. Zuletzt wird in Abschnitt 3.3 eine Bewertung hinsichtlich des Handlungsbedarfs gegeben. Hierbei fließt die Bewertung der existierenden Literatur mit ein. Anschließend erfolgt eine Zuordnung der Anforderungen zu den verschiedenen Kapiteln.

3.1 Aufstellung eines strukturierten Anforderungskatalogs

Die Anforderungen an die domänengetriebene Entwicklung von fortgeschrittenen Web-Anwendungen werden aus den Problemstellungen (siehe Abschnitt 1.4), sowie aus den Zielsetzungen und Beiträgen dieser Arbeit (siehe Abschnitt 1.5) abgeleitet. Der entsprechende Handlungsbedarf wird anhand des Stands der Forschung diskutiert und ermittelt.

Für die Bewertung bestehender Arbeiten wurden sieben Anforderungen (A1-A7) identifiziert. Die Anforderungen A1 bis A5 beschäftigen sich vorrangig mit der Analyse und dem Entwurf. Hier wird auf die Problematik der fehlenden Systematik zur Überführung in die gewünschte Architektur sowie die fehlende Wiederverwendbarkeit der Softwarebausteine eingegangen. Ein weiterer Punkt ist die Separierung der anwendungsagnostischen Funktionalitäten, um diese Wiederverwendbarkeit zu ermöglichen. Die Anforderungen A6 und A7 beschäftigen sich insbesondere mit der Implementierung und dem Testen einer solchen Anwendung. Hierbei steht insbesondere eine durchgängige Systematik im Rahmen des Engineerings im Vordergrund.

A1 - Strukturerhaltende Überführung der Analyseartefakte in den Entwurf

Eine nachvollziehbare und nachverfolgbare Dokumentation der Anforderungen ist ein wichtiger Aspekt [GF94]. Hierbei ist wichtig, dass nachvollzogen werden kann, woher die Anforderungen kommen und wie diese gezielt in den Entwurf und die anschließende Implementierung überführt werden

können. Hierzu soll auch jederzeit eine Rückverfolgung ermöglicht werden. Bei einer Microservice-Architektur werden Konzepte wie die Bounded Contexts aus DDD verwendet [Ne15] um die Microservices zu schneiden. Eine gezielte und strukturerhaltende Anforderungsanalyse kann hierbei helfen, die gewünschte Zielarchitektur zu erreichen. Im Rahmen der Arbeit ist die Ziel-Architektur eine Microservice-Architektur vorgesehen, welche dem Entwurfsmuster der "Layerd Architecture" [Ev04] entspricht. Eine wichtige Eigenschaft an die Anforderungsanalyse ist, dass sich die Analyseartefakte eignen sich in die gewünschte Entwurfsarchitektur überführen zu lassen. Eine Anforderung an die Systematik ist daher, dass die aufgestellten Artefakte möglichst strukturerhaltend in den Entwurf überführt werden können. Hierbei liefert die modellgetriebene Architektur (engl. Model-Driven Architecture, MDA) einen modellgetriebenen Softwareentwicklungsansatz um die verschiedenen Modelle (Computation Independent Models, Platform Independent Models und Platform Specific Models) mittels Transformationen ineinander zu überführen.

A2 - Unterstützung beim Entwurf von Microservice-basierten Anwendungen

Beim Entwurf der Microservice-Architektur müssen die kohäsiv zusammenhängenden Softwarebausteine identifiziert werden. Kohäsiv zusammenhängende Funktionalität wird daher in einem Microservice gebündelt, während eine lose Kopplung für einen weiteren Microservice spricht. Die Kommunikation erfolgt durch entsprechende Schnittstellen. Diese Autonomie wird von der Microservice-Architektur angestrebt [LF14]. Ein wichtiges Ziel dieser Anforderung ist es, dem Entwickler eine Systematik, Werkzeuge und Best Practices an die Hand zu geben, mit deren Hilfe er diese Trennung während des Entwurfs vornehmen kann. Als Resultat sollen die verschiedenen Geschäfts- und Querschnittsfunktionalitäten voneinander getrennt werden. Dies entspricht auch den Prinzipien der losen Kopplung und der hohen Kohäsion [NM+16].

A3 - Etablierung des Domänenwissens

Viele Softwareprojekte scheitern aufgrund mangelnder Verständnis der Geschäftsdomäne einer Organisation [Ev04]. Um ein solches Verständnis aufzubauen, ist der domänengetriebene Entwurf [Ev04, Ve13] geeignet. Das angeeignete Wissen soll zudem kohäsiv gebündelt werden. Passend zu der Auftrennung der Fachlichkeit wird DDD in Verbindung mit Microservices eingesetzt [NM+16]. Hierbei stecken die sogenannten Bounded Contexts die jeweiligen Grenzen ab. Dies schließt eine Taxonomie sowie die Aufteilung in kohäsive Softwarebausteine mit ein. Ein wichtiger Punkt hierbei ist es, dass die kohäsiv zusammengehörenden Softwarebausteine möglichst systematisch erfasst und etabliert werden. Dadurch wird die Wiederverwendbarkeit des Softwarebausteins gewährleistet. Für eine systematische Überführung in die Implementierung ist hierzu zudem eine Formalisierung des Domänenwissens notwendig.

A4 - Separierung von Domänenlogik und Anwendungslogik

Ob ein Microservice wiederverwendbar ist, hängt davon ab, wie er gestaltet ist und welche Funktionalität er bereitstellt. In der Microservice-Architektur ist es daher notwendig, zwischen der fachlichen Logik und der Anwendungslogik zu unterscheiden. Daher werden anwendungsagnostische Anteile isoliert, die separat von der Domäne betrachtet werden. Anwendungsbezogene Inhalte sind im Vergleich zu den fachlichen Microservices auf der Domänenebene in der Regel nicht wiederverwendbar. Daher werden Teile, die nicht anwendungsagnostisch sind, durch zusätzliche anwendungsorientierte Microservices abgedeckt. Eine strategische Aufteilung und Modellierung der verschiedenen Domänen ist hier erforderlich. Dies bezieht sich auch auf die Separierung der Geschäftslogik und Querschnittsfunktionalität. Hierzu müssen die kohäsiv zusammengehörenden Softwarebausteine der Querschnittsdomäne identifiziert und klassifiziert werden. Eine wichtige Unterscheidung sind hierbei Inhalte, welche zur fachlichen Domäne gehören und Inhalte die zu Domänen gehören, die Querschnittsfunktionalität erbringen.

A5 - Systematische Ableitung und Strukturierung der Schnittstelle

Ein wichtiger Punkt ist die Spezifikation der Schnittstelle zwischen den verschiedenen Softwarebausteinen. Die Entwicklungsartefakte sollen hierbei so aufgestellt werden, dass die API systematisch abgeleitet lässt. Für die Gestaltung der API-Richtlinien werden von verschiedenen Quellen publiziert [MA+17]. Ein wichtiger Schritt beim Engineering ist, dass nicht nur die Gestaltung der API systematisch erfolgt, sondern dass die Ableitung der API ebenfalls systematisch und strukturiert erfolgt. Dies schließt die benötigten Datentypen mit ein. Gerade bei einer Microservice-Architektur stellt die API eine sehr wichtige Grundlage dar, weswegen auch die systematische Herleitung dieser betrachtet werden sollte. Das Vorgehen sollte hierbei pragmatisch durchführbar sein.

A6 - Einbezug des Testens in das Engineering

Die korrekte Funktionalität der einzelnen Softwarebausteine, aber auch des gesamten Softwaresystems muss gegeben sein. Hierbei ist die interne Funktionalität und die bereitgestellte Schnittstelle eines Microservices entscheidend. Am Ende entscheidet die Vorgehensweise bei der Softwareentwicklung und das Testen über die Qualität der Software [Re19].

Um die Korrektheit der Schnittstelle weiter sicherzustellen, ist das Testen der Interaktion zwischen den Softwarebausteinen essenziell notwendig. Für die Integrationstests ist das Vorhandensein aller benötigten Microservices erforderlich.

Die Akzeptanz basierend auf den gestellten Anforderungen ist ein weiterer wichtiger Punkt. Hierbei ist das Testen der gesamten Software wichtig, was ein gesamtheitliches Testkonzept erfordert, welches in den Engineering-Ansatz integriert wird. Als Resultat sollen die Tests nicht alle unabhängig neu entwickelt werden müssen, sondern basierend auf den Testdaten und der Systematik ineinander greifen, um den Aufwand der Testaufstellung zu reduzieren.

A7 - Systematische und nachvollziehbare Ableitung von Tests und Testdaten

Eine Anforderung ist die systematische Ableitung der verschiedenen Tests und der Testdaten aus den Analyse- und Entwurfsartefakten. Hierdurch soll bestimmt werden, welche Tests benötigt werden und welche Testdaten konkret benötigt werden. Eine geeignete Wahl der Testdaten ist erforderlich, um gewisse Grenzfälle abdecken zu können. Die Definition der Eingaben und der entsprechenden Ergebnisse der Tests sind ein wichtiges Kriterium für die erfolgreiche Durchführung von Softwaretests [Wi16].

3.2 Bewertung bestehender Arbeiten

Dieser Abschnitt beschäftigt sich mit bestehenden Arbeiten, welche wichtige inhaltliche Konzepte beinhalten, die den Anforderungskatalog betreffen. Neben der Analyse der Arbeiten wird basierend auf den zuvor aufgestellten Anforderungen eine Bewertung im Hinsicht auf die Nutzbarkeit für die Arbeit gegeben.

3.2.1 Towards a Model-Driven Architecture Process for Developing Industry 4.0 Applications [BN+19]

Binder et al. [BN+19] stellen ein Prozessmodell auf Basis des Referenzarchitekturmodells für die Industrie 4.0 (Reference Architecture Model for Industry, RAMI 4.0) [Do15] vor um die fehlende Formalisierung zu konsolidieren. Die Architektur von RAMI sieht die Architektur-Ebenen Buisness, Functional, Information, Communication, Integration und Asset vor. Ein solches System ist Service-orientiert und wird aus mehreren verteilten Services zusammengesetzt. Der vorgestellte Ansatz durchläuft verschiedene bekannte Engineering-Aufgaben und setzt sich aus insgesamt sechs Teil-Prozessen zusammen. Die Ergebnisse der einzelnen Prozessschritte werden auf die verschiedenen Ebenen und Konzepte der modellgetriebenen Architektur (engl. Model-Driven Architecture, MDA) überführt. Neben der Einordnung in die Ebenen der MDA wird zusätzlich die Einordnung in das RAMI-Modell vorgenommen. Zur Einordnung in das RAMI-Modell wird der Prozess wird durch ein

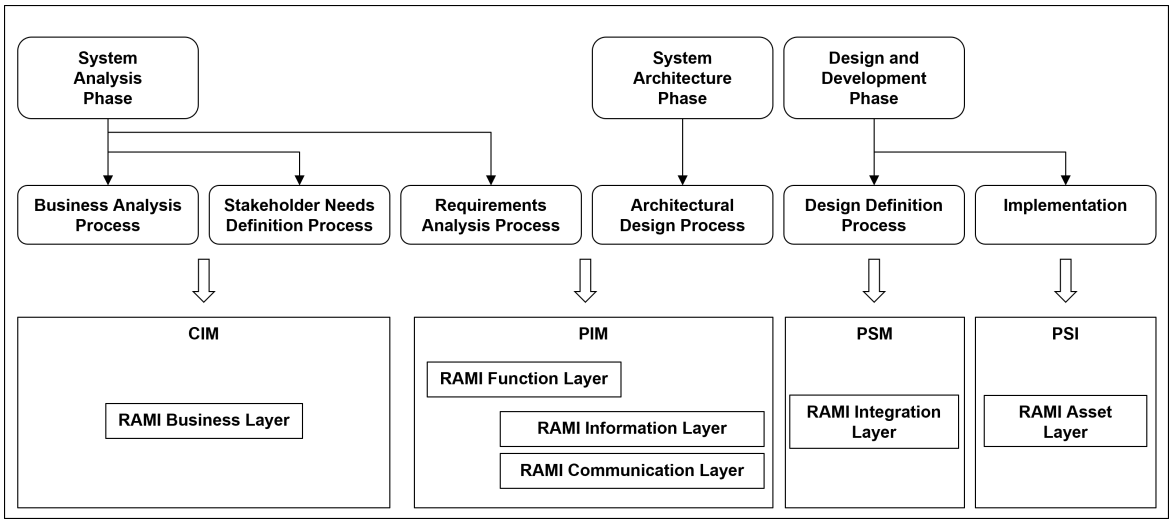


Abbildung 3.1: MDA-Prozess für die Entwicklung von Industrieanwendungen nach [BN+19]

Werkzeug mit der Bezeichnung "RAMI-Toolbox" unterstützt. Abbildung 3.1 zeigt die Überführung der einzelnen Teilprozesse in die MDA.

Zuerst werden die Anforderungen an das System ermittelt. Die Anforderungsanalyse wird aufgeteilt in die Geschäftsanalyse und die Ermittlung der Bedürfnisse der Stakeholder. Als Ergebnis ergibt sich das Computation Independent Model (CIM), welches die Ergebnisse der Anforderungsanalyse festhält. Dieses enthält Modelle zur Beschreibung von Geschäftsfällen, sowie Geschäftsprozesse und Anforderungen, um einen Überblick über das gewünschte System zu erhalten. Hierbei wird sich noch von der Implementierung distanziert.

Im zweiten Schritt wird die Anforderungsanalyse durchgeführt. Gemeinsam mit dem Prozess zum Entwurf der Architektur wird das plattformunabhängige Modell (engl. Platform Indipendent Model, PIM) erstellt. Hierbei wird MDA für die Strukturierung der Inhalte eingesetzt, aber nicht für die automatische Generierung der Entwurfs- und Implmentierungsartefakte. Als Ergebnis wird eine übergeordnete Architektur des Systems inklusive übergeordneter Funktionalitäten und generischer Akteure erhalten.

Anschließend wird dieser Entwurf in das plattformabhängiges Modell für die Architektur (engl. Platform Specific Model, PSM) überführt. Hierbei werden bei der Modellierung verschiedene technologische Aspekte mit integriert. Der Architekturentwurf wird letztendlich in die plattformspezifische Implementierung überführt. Die verschiedenen Artefakte, die durch den Prozess entstehen, werden auf die verschiedenen Ebenen des RAMI-Referenzmodells abgebildet.

Bewertung

Die Autoren stellen einen Rahmen vor, um eine Überführung der verschiedenen MDA-Artefakte in RAMI zu ermöglichen. RAMI selbst setzt auf eine Service-orientierte Architektur. Hierbei werden verschiedene Prozesse angeführt, die die Überführung unterstützen. Die Anforderungen selbst werden durch das RAMI-spezifische Modell festgehalten und anschließend in ein PIM überführt. Hierbei werden weitere Ebenen von RAMI genutzt. Die Überführung selbst erfolgt modellgetrieben anhand der MDA. Wie aber die Fachlichkeiten anhand der Anforderungen gebündelt und in verschiedenen Services aufgeteilt werden, wird nicht genauer erklärt. Daher ist die Anforderung A1 zur systematischen Überführung der Anforderungen nur teilweise erfüllt. Für die Aufteilung der funktionellen und strukturellen Komponenten und deren Beziehungen wird eine domänenspezifische Sprache (engl. Domain-Specific Language, DSL) aufgestellt. Eine Unterstützung des Entwurfs (Anforderung A2) ist dadurch zwar gegeben, aber eine Extraktion des Domänenwissens wird nicht explizit vorgenommen. Zudem wird die Wiederverwendbarkeit der Services nicht erwähnt. Daher sind die Anforderungen A2 und A3 nur teilweise erfüllt. Der Ansatz sieht keine explizite Domänenmodellierung vor, weshalb der Ansatz auch keine Trennung der Anwendungs- und Domänenlogik vornimmt. Entsprechend ist die Anforderung A4 nicht erfüllt. Der nächste Punkt ist die Anforderung A5 an der systematischen Ableitung der Schnittstelle. Die Schnittstellen der entstehenden Services werden als sogenannte Ports im Modell festgehalten. Für eine systematische Ableitung der API-Spezifikation fehlt hier aber die Vorgehensweise im Engineering. Für das Testen der resultierenden Implementierung wird in der Publikation nicht eingegangen. Daher sind die Anforderungen A6 und A7 nicht anwendbar.

3.2.2 Challenges of Domain-Driven Microservice Design: A Model-Driven Perspective [RS+18]

Rademacher et al. [RS+18] geben einen Überblick über die Herausforderungen, die DDD für die modellgetriebene Entwicklung mit sich bringt. Für die Problematik von zu informell gestalteter DDD-Modelle stellen die Autoren erweiterte UML-Diagramme vor, die durch ein erweitertes UML-Profil [RS+17] beschrieben werden. Zur Modellierung der Domäne werden drei Modelle vorgestellt: (i) das Domänenmodell, welches als Klassendiagramm inklusive der verschiedenen Domänenentitäten betrachtet werden kann; (ii) ein Zwischenmodell für die Microservice-Schnittstellen, das auf SoaML basiert; (iii) und ein Modell für die Verteilung der Microservices. Diese Modelle werden der Reihe nach etabliert, damit der Entwickler sich immer auf das Wesentliche des Entwicklungsschrittes konzentrieren kann. Dies inkludiert nach den Autoren- die Schnittstellen und Operationen, die Operationsparameter und Rückgabetypen, sowie die Endpunkte, Protokolle und Nachrichten-Formate.

Das Hauptziel ist die Formalisierung des Domänenmodells, so dass dieses auch durch den modellgetriebenen Ansatz automatisiert umgesetzt werden kann. Für die Modellierung der verschiedenen

Modelle wird ein Eclipse-basiertes Werkzeug namens AjiL verwendet. Zur Formalisierung sieht die Arbeit eine Erweiterung des UML-Metamodells für das Klassendiagramm vor, welches die Autoren zudem in einer weiteren Publikation adressieren [RS+17]. Ein UML-Profil umfasst Erweiterungen von UML-Metaklassen (z. B. einer Klasse) in Form von Stereotypen. Instanzen von erweiterten Metaklassen können mit spezifischen Stereotypen eines Profils semantisch angereichert werden. Ein UML-Profil kann auch formale Einschränkungen definieren, die eine automatische Validierung von der Modellen ermöglicht. So kann beispielsweise überprüft werden, ob Stereotypen korrekt eingesetzt werden. Hierbei werden nun konkret die Muster aus DDD übertragen. Abbildung 3.2 stellt die hinzu-

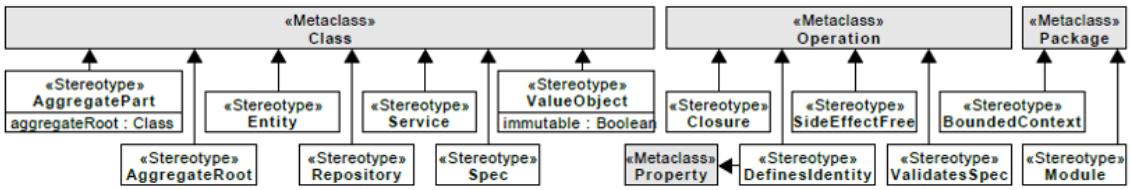


Abbildung 3.2: UML-Profil mit der Bezeichnung Domain-Driven MSA Modeling (DDMM) nach [RS+17]

gekommenen Stereotypen zum Klassendiagramm vor. Neben der Klasse werden weitere Stereotypen für die Operationen, Pakete und Eigenschaften eingeführt. Bei der Modellierung der Domäne wird anschließend auf diese Klassenprofile zurückgegriffen. Gemäß der Fachlichkeit und der Kohäsion werden die Bounded Contexts geschnitten.

Die API der Microservice wird aus dem Klassendiagramm und ihren Beziehungen zu den anderen Bounded Contexts abgeleitet. Für die Modellierung der Schnittstellen wird auf die SoaML (engl. Service-Oriented Architecture Modeling Language) zurückgegriffen. Die Autoren verdeutlichen ihr Konzept anhand eines Cargo-Beispiels, bei dem ein Domänenmodell erstellt wird, welches anhand der UML-Profile erweitert wird. Dieses ist im obigen Teil der Abbildung (Teil a) dargestellt. Das weitere Modell (Teil b) modelliert das Zwischenmodell für die Schnittstellen. Im konkreten Fall erhält jeder beteiligt Microservice eine Schnittstellendefinition, welche ggf. noch um geteilte Information erweitert wird.

Bewertung

Die Autoren stellen einen modellgetriebenen Ansatz zum Entwurf einer Microservice-Architektur vor. Neben der Beschreibung der Fachlichkeit gehen die Autoren nicht auf die Beschreibung der Anforderungen und deren Überführung in den Entwurf ein, weshalb die Anforderung A1 nicht erfüllt ist. Für die Unterstützung des Entwurfs und der Anforderung A2 ist die Formalisierung des Domänenwissens ist ein wichtiger Schritt. Eine Erweiterungen des UML-Profils ist hierbei ein geeignetes Konzept, dass für die Modellierung der Fachlichkeit in einem erweiterten Klassendiagramm einschlägig ist.

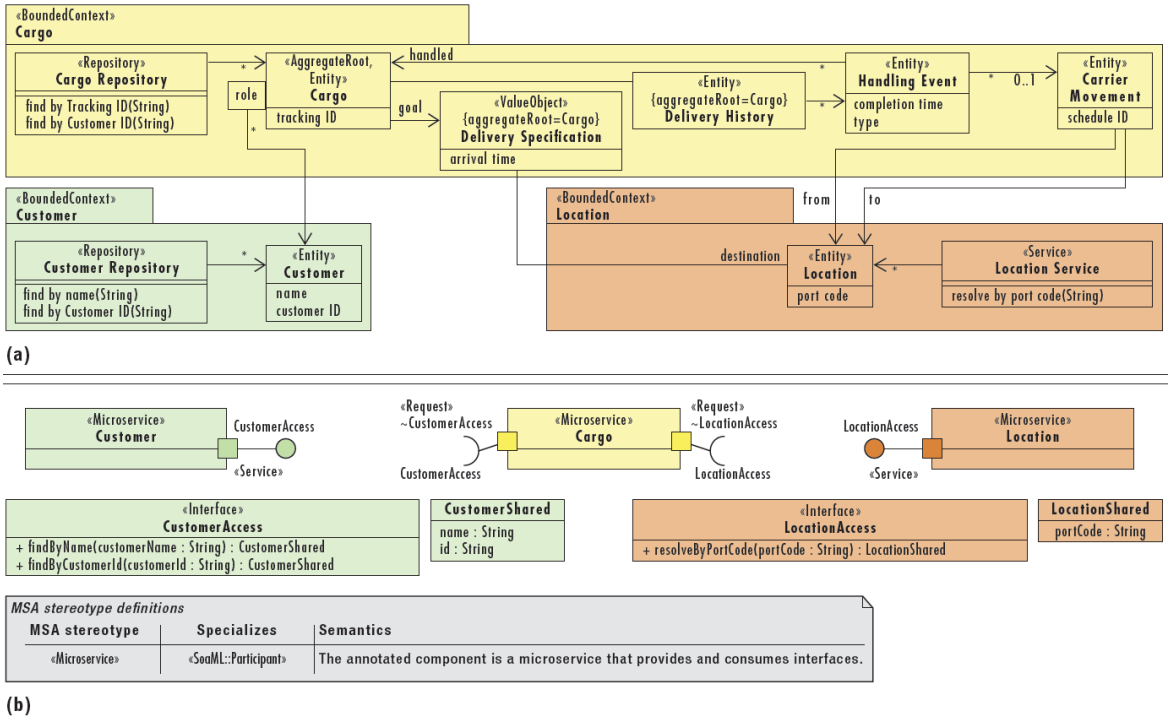


Abbildung 3.3: Auszug aus der Modellierung mit erweitertem UML-Profil und einem Übergangsmodell für die Schnittstelle aus [RS+18]

Weiterhin greifen die Autoren auch eine Einordnung in die verschiedenen Bounded Contexts auf. Die Anforderung Unterstützung beim Entwurf von Microservice-basierten Anwendungen (A2) ist daher weitestgehend erfüllt. Das vorgestellte Konzept wird von dieser Arbeit aufgegriffen und fortgeführt, indem eine Erweiterung des UML-Profils für die strategische Modellierung eingeführt wird. Der im Abschnitt 4 beschriebene Prozess der systematischen Ableitung des Domänenmodells verwendet die Formalisierung als Grundlage für die Formalisierung des Domänenmodells, insbesondere durch die Stereotypen. Die Etablierung des Domänenmodells erfolgt durch die Autoren mittels DDD und der Microservice-Architektur. Die Etablierung des Domänenwissens ist dadurch gegeben, weshalb A3 ebenfalls erfüllt ist. Ein weiterer Aspekt, der in dieser Arbeit behandelt wird, ist die Separierung von Domänenlogik und Anwendungslogik. Hierzu wird die strategische Modellierung des Domänenmodells benötigt. Dieser Aspekt wird in der Publikation nicht von den Autoren erwähnt. Es werden zwar Bounded Contexts aufgestellt, aber eine explizite strategische Modellierung (wie die Context Map) wird nicht erwähnt. Die Anforderung an die Separierung des Anwendungs- und Domänenwissens (A4) wird daher nicht oder nur sehr bedingt erfüllt. Für die Ableitung der Schnittstelle werden gemäß SoAML die Schnittstellen in einem weiteren Diagramm beschrieben. Diese Definitionen eigenen sich zur Ableitung von REST-Schnittstellen. Das Engineering vom Diagramm zu der Spezifikation der eigentlichen Schnittstelle wird jedoch nicht beschrieben. Da die Systematik zur Überführung in die Schnittstelle fehlt, aber wichtig für die Gestaltung derer ist, ist die Anforderung zur systematischen

Ableitung der Schnittstelle (A5) nur teilweise erfüllt. Das dritte Diagramm betrachtet die Verteilung der verschiedenen Microservices. Der vorige Schritt der Implementierung erfolgt aus den aufgestellten Modellen. Der Aspekt des Testens wird nicht berücksichtigt, weshalb die Anforderungen bezüglich des Einbezugs des Testens und der Ableitung von Testdaten (A6 und A7) nicht anwendbar sind.

3.2.3 Domänengetriebener Entwurf von ressourcenorientierten Microservices [Gi18]

In der Dissertation von Giessler [Gi18] wird ein systematischer Ansatz zur Ableitung von Microservices vorgestellt. Hierbei wird ein formales Domänenmodell auf der Basis von DDD-Konzepten aufgestellt. Im nächsten Schritt werden anhand verschiedener Faktoren die systematische Ableitung eines Ressourcenmodells betrachtet, welches für die Aufstellung der ressourcenorientierten Microservice-Schnittstellen genutzt wird.

Abbildung 3.4 skizziert das Vorgehen im Überblick, welches [Gi18] zur systematischen Etablierung der API verwendet. Bevor die Ableitung beginnen kann, werden die relevanten Domänenobjekte ermittelt, welche über eine ressourcenorientierte Web-API zugänglich gemacht werden sollen. Die Identifikation der Domänenobjekte werden durch die Nutzeranforderungen entsprechende Hinweise gewonnen. Ein Prototyp auf Basis dieser Nutzeranforderungen, welche Informationen oder Interaktionen benötigt werden, wird als Beispiel angeführt. Jedes Domänenobjekt ist am Ende ein Kandidat, welcher zur Ressource werden kann.

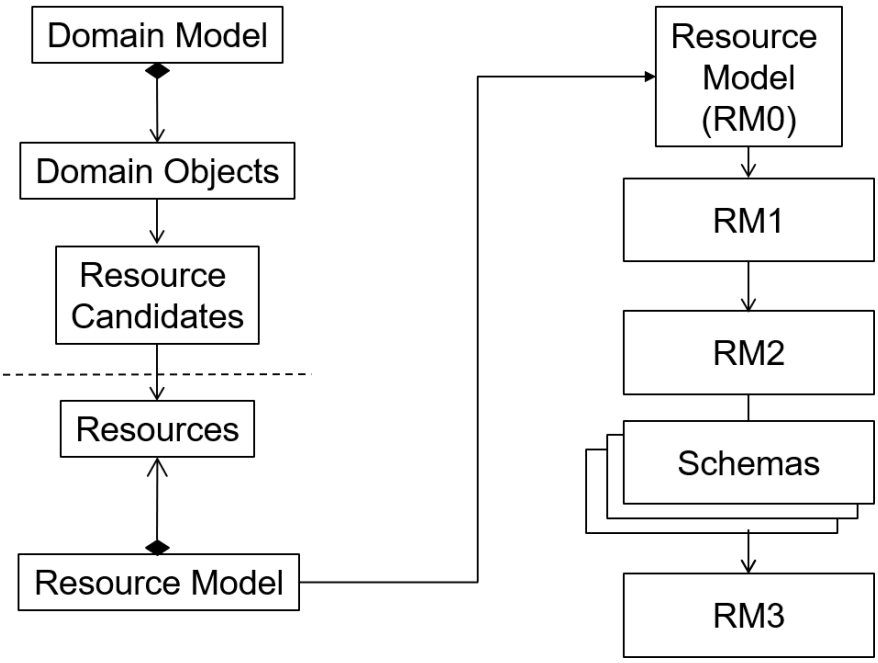


Abbildung 3.4: Ableitung des Ressourcenmodells nach [Gi18]

Giessler unterscheidet in seinem Ansatz die Ressourcen als Listenressourcen, Primärressourcen, Unterressourcen, Informationsressourcen und Aktivitätsressourcen. Aus der Semantik der einzelnen Ressourcentypen sowie den Domänenobjekten und ihren Assoziationen werden Heuristiken und Transformationsregeln angewendet, um die Domänenobjekte zu überführen. Die Heuristiken identifizieren einen Ressourcentyp auf der Grundlage des Domänenmodells. Für die Identifikation werden nur die Ressourcenkandidaten berücksichtigt. Jede Heuristik ist mit einer Transformationsregel verknüpft, die einen Ressourcenkandidaten in eine Ressource umwandelt, indem sie ihm ein Stereotyp aus einem speziellen UML-Profil zuordnet, das als Ressourcenorientierung bezeichnet wird und ihn in ein Ressourcenmodell einfügt. Das Ressourcenmodell wendet das zuvor erwähnte UML-Profil an und erweitert die UML um Konzepte des REpresentational State Transfer (REST).

Das anfängliche Ressourcenmodell wird auf der Grundlage des zuvor modellierten Domänenmodells erstellt. Anschließend werden vier Schritte durchgeführt, um das Ressourcenmodell weiter für die Ableitung einer REST API vorzubereiten. Jede Ressource wird um eine URL-Komponente (Uniform Resource Locator) ergänzt, die auf dem ihrem zugewiesenen Ressourcentyp basiert. Durch Links zwischen Ressourcen wird die vollständige URL abgeleitet. Für die Erstellung der entsprechenden Komponenten werden linguistische Muster und bewährte Verfahren verwendet, wodurch RM1 entsteht. Im nächsten Schritt wird die Interaktion mit einer Ressource vom Client aus betrachtet. Das von der Domäne angebotene Verhalten wird hierbei auf eine einheitliche Schnittstelle abgebildet. Zur Vereinfachung der Abbildung werden bestehende Muster verwendet und den einzelnen Ressourcentypen zugeordnet. Nach diesem Schritt ergibt sich RM2. Die Erstellung der Repräsentationsschemata ist der nächste Schritt im Vorgehen. Die Schemata definieren die Struktur der Anfrage und der Antwort. Neben der Auflistung der erforderlichen und optionalen Attribute werden auf deren Basis Mock-Werte generiert werden, die für Web-API-Tests geeignet sein können.

Der letzte Schritt greift die Versionierung des Ressourcenmodells und schließlich der Web-API auf. RM2 wird durch entsprechende Versionsinformationen entsprechend der semantischen Versionierung ergänzt. Die Darstellung dieser Versionsinformationen in der Web-API wird durch weitere Muster behandelt. Außerdem wird ein Vorgehen vorgeschlagen, wie die Versionsinformationen geändert werden können. RM3 ist das Endergebnis und wird für die Ableitung der REST-Schnittstelle verwendet.

Bewertung

Der Autor stellt ein systematisches Vorgehen zur Ableitung eines Ressourcenmodells aus einem zuvor aufgestellten Domänenmodell vor. Die Ableitung des Domänenmodells geschieht anhand der Anforderungen und einem Spezifikationsdokument, welches in ein Domänenmodell überführt wird. Da bei der Überführung nicht auf Systematik und Strukturerhaltung eingegangen wird, ist die Anforderung A1 nur bedingt erfüllt. Für den Entwurf wird ein domänengetriebener Entwurfsprozess eingesetzt. Dieser führt zu dem Domänenmodell und sorgt für die Etablierung des Domänenwissens, welches mittels

UML-Profil formalisiert wird. Die Anforderungen A2 und A3 zur Unterstützung des Entwicklers und der Etablierung des Domänenwissens sind daher weitestgehend erfüllt. Zudem wird beschrieben, wie das Domänenwissen statisch und dynamisch modelliert werden kann. Die strategische Aufteilung in mehrere Domänen wird vom Autor nicht behandelt. Es wird zwar festgelegt, dass die Microservices wiederverwendbar sein sollen, auf eine Differenzierung zwischen der Anwendungslogik, welche nur bedingt wiederverwendbar und die hochgradig wiederverwendbare Domänenlogik wird nicht berücksichtigt. Daher ist die Anforderung zur Separierung der Domänen- und Anwendungslogik nur bedingt erfüllt. Einer der wichtigsten Aspekte, der in der Arbeit [Gi18] behandelt wird, ist die Ableitung der Schnittstelle. Hier wird systematisch und strukturiert beschrieben, wie das Domänenmodell in eine Ressourcenmodell abgeleitet wird, welches anschließend in eine REST-Schnittstelle überführt wird. A5 ist daher vollständig erfüllt. Ein Nachteil des Ableitungsprozess der Schnittstelle ist jedoch die hohe Komplexität. Daher wird in dieser Arbeit in Abschnitt 4.3.2 eine pragmatischere Variante gewählt. Für die Implementierungen werden hinweise zur Strukturierung des Codes und die Einordnung der Domäne gegeben. Tests werden jedoch nur kurz angerissen, aber nicht weiter verfolgt. Daher ist die Anforderung A6 zur Implementierung und des Testens nur gering und die Ableitung der Tests A7 nicht erfüllt.

3.2.4 An Open IoT Framework Based on Microservices Architecture [SL+17]

In [SL+17] wird ein allgemeines Rahmenwerk für Microservice-basierte IoT-Anwendungen vorgestellt, welches sich für beliebige Domänen einsetzen lassen soll. Der Grund für das Rahmenwerk sehen die Autoren bei der steigenden Komplexität von IoT-basierten Anwendungen und der fehlenden Skalierbarkeit von monolithischen Architekturen. Die Autoren sehen bei den bereits existierenden Rahmenwerke für Microservice-basierte IoT-Anwendungen einen sehr starken Bezug zu einer spezifischen Domäne, was die Anwendbarkeit der bisherigen Rahmenwerke schmälert. Das vorgestellte Rahmenwerk greift diese Problematik auf und schlägt eine erweiterbare Architektur für Microservice-basierte IoT-Anwendungen vor. Bei der Gestaltung des Rahmenwerks berücksichtigen die Autoren die lose Kopplung und die hohe Kohäsion entsprechend der Microservice-Architektur. Für die Schnittstellen werden REST-Prinzipien angewendet. Aufgestellt wurde das Rahmenwerk durch die Analyse und Entkopplung bestehender IoT-Systeme mittels der Extraktion der Geschäftsfunktionalitäten. Wie diese Analyse jedoch konkret durchgeführt wurde, wird nicht in der Publikation von den Autoren nicht weiter vertieft. Die resultierende Microservice-basierte Systemarchitektur besteht aus neun verschiedenen Microservices, welche die während der Analyse extrahierten Geschäftsfunktionalitäten aufstellen. Abbildung 3.5 zeigt die Architektur des Rahmenwerks mittels der aufgestellten Microservices. Der Core-Service (im Zentrum der Abbildung) stellt die notwendigen Funktionalitäten für die Interaktion zwischen den Geräten und den weiteren Microservices bereit. Neben einer REST-Schnittstelle wird das Publish/Subscribe-Prinzip für die Kommunikation zwischen den Services verwendet. Die Kommunikation des Core-Service mit den weiteren Services erfolgt nach den Autoren ausschließlich

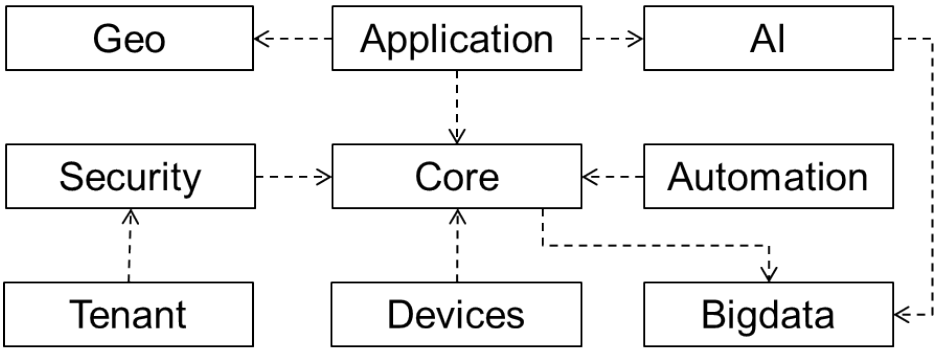


Abbildung 3.5: Architektur des Rahmenwerks aus [SL+17]

über Events, was durch die Funktionalitäten der Ereignisverwaltung, die Plugin-Verwaltung und die Ressourcenermittlung des Core-Services abgedeckt wird. Für die sicherheitstechnischen Maßnahmen wird ein Security Microservice eingesetzt, welcher die Authentifizierung, Autorisierung und das Management der Benutzer übernimmt.

Bewertung

Die Autoren stellen ein Rahmenwerk für IoT vor, welches eine Microservice-Architektur als Grundlage hat. Für die Analyse der Anforderungen stehen hierbei IoT-Anwendungen im Vordergrund. Für das angebotene Rahmenwerk wird hierbei auf bereits existierende Funktionalitäten von IoT-Plattformen gesetzt. Die angebotenen Systeme sollen im Rahmen der hohen Kohäsion und der losen Kopplung agieren. Die Anforderung der strukturierten Überführung der Anforderungen ist nicht gegeben, da nicht die Anwendungsentwicklung, sondern die technischen IoT-Gegebenheiten im Vordergrund stehen. Für die den Entwurf wird ein Rahmenwerk für die Querschnittsdomäne IoT gegeben. Die Architektur für die Geschäftslogik lässt sich jedoch nicht anhand des Rahmenwerks erschließen. Daher ist die Anforderung an die Unterstützung beim Entwurf auch nur bedingt erfüllt. Die angebotenen Microservices orientieren sich vermutlich daher auch nur sehr bedingt am Geschäft und die Beschreibungen dieser sind sehr technisch ausgelegt.

Die Etablierung des Domänenwissens wird anhand einer Analyse bestehender IoT-Plattformen durchgeführt. Was bei diesem Aspekt allerdings fehlt ist, wie systematische und nachvollziehbar bei der Analyse der Plattformen und dem Entwurf der Microservice-Architektur vorgegangen wird. In dem Rahmen der präsentierten Architektur werden anhand der Plattformen etabliert. Ein Hauptkritikpunkt an den Microservices, der vorgestellten Rahmenarchitektur, ist die Einbeziehung von anderen Bereichen (z. B. Sicherheit, Mandantenbezug), welche zu anderen (Querschnitts-) Domänen gehören. Ein Grund warum die oben genannten Probleme bei der IoT-Rahmenarchitektur entstehen liegt wahrscheinlich daran, dass bei dem Ansatz keine Domänenmodellierung herangezogen wurde. Aus diesem

Grund ist die Etablierung des Domänenwissens nur schwer nachzuvollziehen. In diesem Rahmen haben die Autoren auch nicht berücksichtigt, dass das IoT immer nur einen bestimmten Teil einer Anwendung bildet. Hierbei sind die verschiedenen Teile architekturell sauber von der Geschäftsfunktionalität der Anwendung zu separieren. Insbesondere wurden Aspekte bzw. Funktionalitäten in einen Microservice des Rahmenwerks eingearbeitet, die von der Querschnittsdomäne des Identitäts- und Zugriffsmanagement erbracht werden. Diese Lücke lässt sich durch ein systematisches Vorgehen schließen, welches neben dem Bezug der benötigten Microservices auch den Weg zu der Anwendung beschreibt. Dieses Vorgehen ist ein wichtiger Baustein für die Separierung der Querschnittsfunktionalität (in diesem Fall IoT) einer Microservice-basierten Architektur in Kapitel 6. Für eine geeignete Wiederverwendbarkeit müssen die Services anderer Bereiche in den jeweiligen Domänen angesiedelt werden. Die Anforderungen zur Etablierung des Domänenewissens und der Separierung der Domänen- und Anwendungslogik und Querschnittsfunktionalität (A3 und A4) sind demnach nur teilweise erfüllt. Die Anforderung an die systematische Ableitung der APIs (A5) ist nicht erfüllt. Es werden zwar exemplarische Implementierungen der Schnittstellen gezeigt, die Ableitung dieser wird aber nicht adressiert. Wie die verschiedenen vorgeschlagenen Microservice in die Implementierung überführt werden und wie diese getestet werden wurde von den Autoren ebenfalls nicht betrachtet, weshalb die Anforderungen des Einbezugs des Testens und die Ableitung der Testdaten (A6-A7) nicht anwendbar sind.

3.2.5 Microservice Test Process: Design and Implementation [SR+18]

Der Kernbeitrag von Savchenko et al. [SR+18] ist ein allgemeiner Testprozess, der die Microservice-Entwicklung um mehrere Testschritte, wie Komponententests, Integrationstests und kontinuierliche Systemtests erweitert. Darüber hinaus vergleicht der erste Teil der Publikation die Microservice-Architektur mit Monolithen, agentenorientierten Systemen und Akteuren.

Das Ergebnis ist eine Verlagerung der Kommunikation von der Anwendungsebene auf die Infrastrukturebene. Als Resultat wird ein Testprozess für die Microservice-Architektur unter Berücksichtigung dieser Veränderung vorgeschlagen. Abbildung 3.6 zeigt den Überblick über den vorgeschlagenen Testprozess.

Der Testprozess beginnt hierbei bei der Definition der Microservice-Schnittstellen. Diese API-Spezifikation wird als Grundlage für den Prozess verwendet, da die Logik des Microservices auf den Schnittstellen aufsetzt. Es wird zwischen internen funktionalen Tests, funktionalen Selbsttests für Container, Komponententests und Integrationstests unterschieden. Das interne funktionale Testen wird vom Entwickler durchgeführt und besteht in dieser Arbeit aus Unit- und Komponententests. Des Weiteren werden Container-Selbsttests durchgeführt, die gegen eine eigene Microservice-Schnittstelle laufen, während sie durch den Microservice selbst ausgelöst werden. Der nächste Schritt ist die

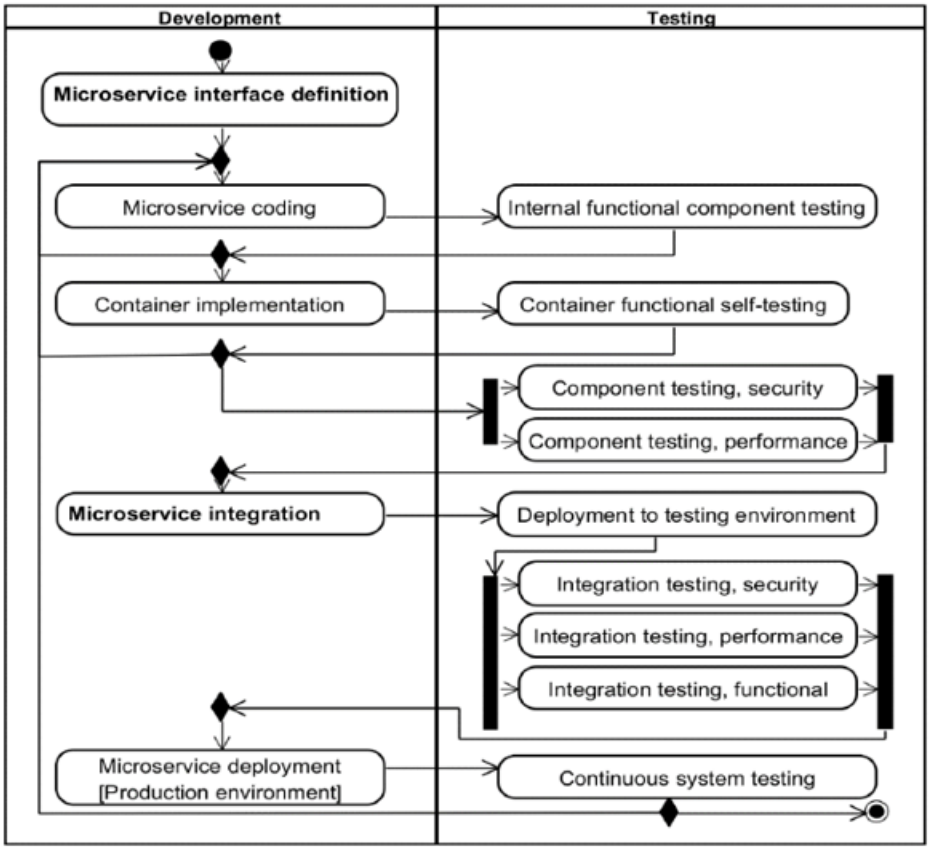


Abbildung 3.6: Testprozess für Microservices nach [SR+18]

Verteilung der Softwarekomponenten in eine Testumgebung, um funktionale und nicht-funktionale Integrationstests durchzuführen. Bei Erfolg wird der Microservice ausgeliefert.

Für die Durchführung der Tests wird eine Test-Service-Architektur eingeführt, die den Testprozess für den menschlichen Tester unterstützt. Diese Architektur verwendet verschiedene Container um das Schreiben und Ausführen der Tests zu ermöglichen. Beispielsweise gibt es eigene Container um die Authentisierung und Autorisierung des Test-Benutzers durchzuführen. Anschließend kann der Benutzer ein Testprojekt erstellen und die gewünschten Tests als Dienst bereitstellen.

Bewertung

Die Autoren stellen einen Testprozess vor, der bei der Spezifikation der API beginnt. Für die Bewertung bedeutet dies, dass die Überführung der Analyseartefakte in den Entwurf, sowie die Unterstützung beim Entwurf und die Etablierung des Domänenwissens (A1-A4) nicht bewertet werden kann. Bezüglich der Schnittstelle wird nur erwähnt, dass diese der Startpunkt des Testprozesses darstellt. Wie die API abgeleitet wird, wird nicht betrachtet. Daher ist die Anforderung der systematischen Ableitung

(A5) ebenfalls nicht erfüllt. Im Hinblick auf den Einbezug des Testens in das Engineering wird ein kompletter Testprozess offeriert. Daher ist der Einbezug des Testens in das Engineering (A6) erfüllt. Was dieser Testprozess, einschließlich der Testarchitektur, nicht berücksichtigt, sind die Anforderungen, die an die Anwendung gestellt werden. Dadurch wird nicht klar, wie die Tests systematisch von den Anforderungen abgeleitet werden können und letztendlich welche Testfälle und Testdaten einbezogen werden sollen. Daher ist die Anforderung zur systematischen und nachvollziehbaren Ableitung der Tests und Testdaten nicht erfüllt.

3.2.6 Consumer-Driven Contract Tests for Microservices: A Case Study [LM+19]

Lehvä et al. [LM+19] untersuchten in einer Fallstudie, wie eine Microservice-basierte Anwendung effektiv getestet werden kann. Sie erläutern, dass in einer Microservice-Umgebung die Abhängigkeiten von mehreren Microservices für einen Integrationstest die Komplexität erhöhen. Um die Komplexität zu reduzieren, werden Ende-zu-Ende-, Integrations- und Komponententests durch konsumentengetriebene Vertragstests (engl. Consumer-Driven Contract tests, CDC tests) unterstützt. Hierbei definiert der Vertrag, was zwischen dem Anbieter und dem Konsumenten ausgetauscht wird. Im Ansatz zur klassischen Testpyramide werden konsumentengetriebene Vertragstests zwischen den Integrations- und Komponententests eingesetzt. Die Studie führt zu dem Ergebnis, dass CDC-Tests als Ersatz von Integrationstests eingesetzt werden könnten, da die Ergebnisse der Tests vergleichbar sind. Insbesondere wird der Vorteil bei dem geringeren Entwicklungsaufwand und der Ausführungszeit gesehen.

Abbildung 3.7 zeigt die Platzierung der CDC-Tests in die Testpyramide, welche die klassische Testpyramide aus Abschnitt 2.6.1 erweitert. In der Publikation wird zusätzlich eine Unterscheidung zwischen isolierten Testen und integriertem Testen vorgenommen. Die CDC-Tests sind hierbei bei den isolierten Tests aufgeführt. Anhand mehrerer Microservices wird das isolierte Testen mit CDC-Tests demonstriert.

Die Hauptargumente, die Lehvä et al. für CDC-Tests in Microservice-Architekturen anführen ist, dass die Microservices anhand der Schnittstelle isoliert getestet werden können. Dies wird aufgrund der wohldefinierten Schnittstellen ermöglicht. Die Option Microservices isoliert zu testen sorgt dafür, dass das Testen wesentlich kostengünstiger wird, da nicht mehrere Services gleichzeitig bereitgestellt werden müssen. Abbildung 3.8 zeigt verschiedene Testoptionen im Überblick. Der grüne Kreis gibt an, welche Komponenten verfügbar sein müssen. Bei Integrations- und Ende-zu-Ende-Tests müssen sowohl der Consumer als auch der Provider gleichzeitig ausgeführt werden. CDC-Tests hingegen können sowohl für den Konsumenten als auch für den Provider isoliert ausgeführt werden. Diese Isolierung wird erreicht, indem ein Vertrag für die API-gesteuerte Kommunikation von Microservices erstellt wird.

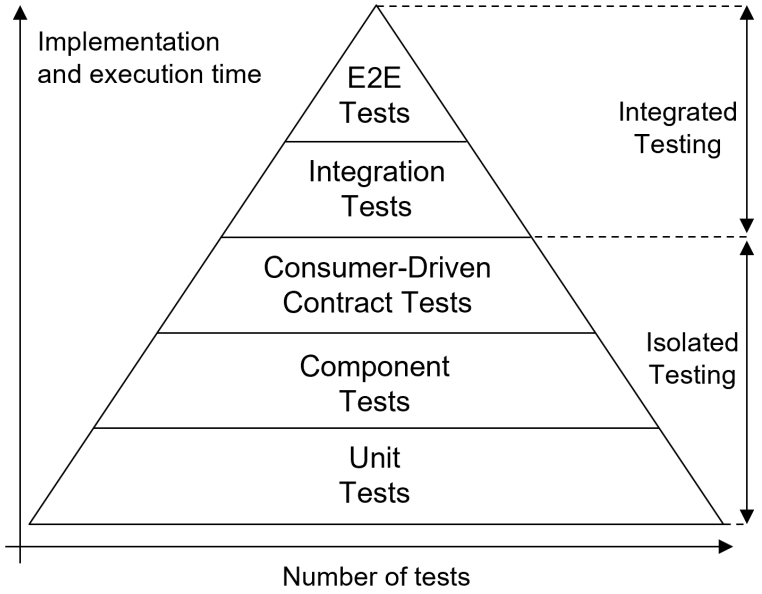


Abbildung 3.7: Testpyramide nach [LM+19]

Durch diese Verträge werden die konsumentengetriebenen Vertragstests ermöglicht. Der Vertrag legt fest, wie die Nachrichten zwischen den Services auszusehen haben. Dieses gemeinsame Verständnis wird anhand eines Vertrags [Pac-Doc] dokumentiert. Der Vertrag dient als Grundlage für die Erstellung von Mocks für den Konsumenten und den Provider, um den jeweils anderen zu simulieren, ohne dass sie eingesetzt werden müssen. Abbildung 3.8 zeigt einen Konsumenten und einen Provider, die jeweils die Inhalte des gleichen Vertrags verwenden. Dieser Vertrag ermöglicht es, den normalerweise aufwendigen Integrationstest beider Systeme in isoliert arbeitende Tests zu unterteilen. Für eine korrekte Validierung unter Verwendung dieser Tests müssen diese Verträge gepflegt und bei jeder Änderung des Konsumenten entsprechend versioniert werden. Durch eine Fallstudie über industrielle Systeme führt die Autoren zu dem Schluss, dass CDC-Tests schneller und stabil sind, dem Anbieter Informationen über die Konsumenten über die Verbraucher liefern, welche seine angebotene API benutzen. Dies ermöglicht es dem Anbieter, sich auf der Grundlage dieser Geschäftsanforderungen weiterzuentwickeln. Einen weiteren Vorteil sehen die Autoren darin, dass die verwendeten Verträge als Kommunikationsmittel zwischen den Entwicklungsteams dienen.

Bewertung

Die Autoren stellen konsumentengetriebene Vertragstests als eine leichtgewichtigere Variante für Integrationstests vor. Im Vergleich zu den Integrationstests wird immer nur der zu testende Microservice benötigt. Der Ansatz selbst steigt beim Entwurf der Schnittstelle ein. Daher sind die Anforderungen

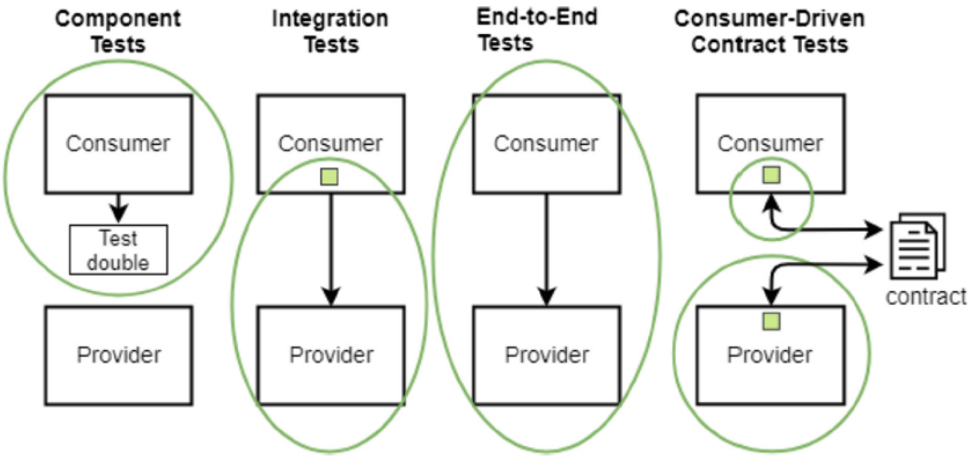


Abbildung 3.8: Umfang der verschiedenen Testmethoden aus [LM+19]

A1-A4 von der Publikation nicht anwendbar. Bezüglich der systematischen Aufstellung der Schnittstelle wird der Vertrag verwendet. Dieser enthält die beteiligten Microservices, deren Interaktionen (Anfrage und Antwort), verschiedene Regeln, sowie Metadaten wie die Version. Dadurch wird die anbietende und anfragende Schnittstelle beschrieben. Was in dem Ansatz noch fehlt ist die systematische Überführung zur API-Spezifikation (A5) selbst, weswegen diese Anforderung nur teilweise erfüllt ist. Beim Einbezug in das Engineering ist problematisch, dass die Verträge spezifisch mit dem Werkzeug "Pact" aufgebaut werden. Konkret heißt dass, dass die Implementierung zuerst erfolgt und dann die Verträge sich daraus automatisiert ableiten. Beim Engineering sollte aber immer zuerst der Vertrag als Teil des Entwurfs etabliert werden. Andernfalls besteht die Gefahr, dass keine saubere Schnittstellenspezifikation erstellt wird. Daher ist die Anforderung A6 nur teilweise erfüllt. Wie sich die Testdaten aus dem Engineering ergeben (A7), wird nicht erwähnt, weshalb diese Anforderung ebenfalls nur teilweise erfüllt ist.

3.3 Handlungsbedarf

Der Handlungsbedarf ergibt sich aus der Auswertung der bestehenden Arbeiten und den Anforderungen an die Arbeit und den Problemstellungen die gelöst werden sollen. In Tabelle 3.1 wird der Überblick dargestellt. Eine vollständig erfüllte Anforderung wird mit dem Zeichen ● angegeben. Ist eine Anforderung nur zum Teil oder bedingt erfüllt, dann wird diese mit dem Zeichen ◐ gekennzeichnet. Eine nicht erfüllte Anforderung wird mit dem Zeichen ○ angegeben. Ist eine Anforderung nicht anwendbar, wird das Zeichen / verwendet.

Wie die Übersichtstabelle zeigt, liefert keine der bestehenden Arbeiten eine vollständige Überdeckung der aufgestellten Anforderungen aus 3.1. Der Handlungsbedarf lässt sich somit aus den fehlenden

	A1 - Strukturerhaltende Überführung der Analysearteфakte in den Entwurf	A2 - Unterstützung beim Entwurf von Microservice-basierten Anwendungen	A3 - Etablierung des Domänenwissens	A4 - Separierung von Domänenlogik und Anwendungslogik	A5 - Systematische Ableitung und Strukturierung der Schnittstelle	A6 - Einbezug des Testens in das Engineering	A7 - Systematische und nachvollziehbare Ableitung von Tests und Testdaten
[BN+19] Towards a Model-Driven Architecture Process for Developing Industry 4.0 Applications	●	◐	◐	○	◐	/	/
[RS+18] Challenges of Domain-Driven Microservice Design: A Model-Driven Perspective	○	●	◐	◐	◐	/	/
[Gi18] Domänengetriebener Entwurf von ressourcenorientierten Microservices	◐	●	●	◐	●	◐	○
[SL+17] An Open IoT Framework Based on Microservices Architecture	○	◐	○	◐	◐	/	/
[SR+18] Microservice Test Process: Design and Implementation	/	/	/	/	○	●	○
[LM+19] Consumer-Driven Contract Tests for Microservices: A Case Study	/	/	/	/	◐	◐	◐

Tabelle 3.1: Bewertung der bestehenden Publikationen in Bezug auf den Anforderungskatalog

Anforderungen ableiten. Der Handlungsbedarf motiviert ebenfalls die Inhalte der Forschungsbeiträge dieser Arbeit.

Um eine fortgeschrittene Web-Anwendung zu entwickeln, ist es erforderlich, die Entwickler durch ein systematisches Engineering zu unterstützen. Hierbei ist es insbesondere hilfreich, wenn die aufgestellten Anforderungen sich systematisch auf den Entwurf abbilden lassen. Hierfür lassen sich bestimmte Aspekte aus [BN+19, Gi18] zum Engineering entnehmen. Bezüglich der Überführung in die konkrete Architektur werden Aspekte für den gesamten Engineering-Ansatz adaptiert. Eine wichtige Ergänzung und ein Handlungsbedarf hierbei ist, dass eine strukturerhaltende Überführung der Artefakte ermöglicht wird. Dadurch ist eine direkte Ableitung der Ziel-Software-Architektur möglich. Für die Unterstützung beim Entwurf und insbesondere der Formalisierung des zu erstellenden Domänenmodells bietet [RS+18] durch das erweiterten UML-Profil einen Ansatz. Die Separierung der

Anwendungslogik und der Domänenlogik, sowie die Extraktion der Querschnittsfunktionalität wird in keiner der untersuchten Arbeiten vollständig erfüllt. Dabei ist der Bedarf der Trennung relevant, insbesondere, da die Querschittsfunktionalität in vielen Anwendungen wiederverwendet werden kann.

Im Beispiel der Querschnittsdomäne IoT liefert [SL+17] eine Idee zur Strukturierung dieser. Was hierbei jedoch deutlich wird ist, dass eine gezielte Untersuchung und Strukturierung der Domäne notwendig ist. Andernfalls besteht die Gefahr, dass verschiedene Domänen vermischt werden. Für den entstehenden Handlungsbedarf ist zu klären, wie die verschiedenen Domäneninhalte extrahiert und mit dem Architekturentwurf verknüpft werden können.

Die systematische Spezifikation der Schnittstelle wird von Giessler [Gi18] sehr ausführlich behandelt. Jedoch ist die Anwendung des Vorgehens für die Entwickler sehr komplex. Hierdurch wird die Anwendbarkeit innerhalb des Entwicklungsansatzes geschmälert. Die Anforderung zur systematischen Ableitung der Schnittstelle hat hier noch den Zusatz, dass diese möglichst pragmatisch erfolgen kann. Optimalerweise lässt sich diese aus Modellen ableiten, welche direkt für das weitere Engineering genutzt werden.

Das Testen einer Microservice-basierten Anwendung wird insbesondere auf der Integrationsebene wichtig. Aus Sicht des Engineering ist das Testen nicht erst während der Implementierung zu berücksichtigen, sondern sollte bereits während der Analysephase mit aufgegriffen werden. Bei den betrachteten Publikationen [SR+18, LM+19] ist auffällig, dass diese erst zu einem späten Punkt im Engineering platziert sind. Insbesondere für die Überprüfung der Anforderung besteht hier ein Bedarf an einem durchgängigen, systematischen Testkonzept für fortgeschrittene Web-Anwendungen, welche Microservice-basierte Architekturen einsetzen. Neben der Integrationsebene betrifft dies das durchgängige Testen der gesamten Microservice-basierten Anwendung.

3.3.1 Bezug der Anforderungen zu den Kapiteln

Die verschiedenen Kapitel behandeln bestimmte Aspekte der Anforderungen, welche genauer erläutert werden. Abbildung 3.9 zeigt den Überblick über die Kapitel und die zugeordneten Anforderungen und den durchgeführten Handlungsbedarf. In Kapitel 4 wird der Gesamtansatz (A1-A7) behandelt und liefert einen Überblick über das Gesamtkonzept. In Kapitel 5 werden die Anforderungen A1, A2, A3, A5 behandelt, welche sich in Richtung des Engineering und der Systematik bewegen. In Kapitel 6 werden die Anforderungen zur Querschnittsfunktionalität behandelt. Anhand der Domäne IoT wird der Einbezug von Querschnittsdomänen diskutiert. Aber auch andere Querschnittsbereiche fließen in die Betrachtung mit ein. Kapitel 7 greift das Thema der Implementierung und des Testens auf. Hierfür werden die Anforderungen für das systematische Engineering in Bezug auf die Überführung des Entwurfs und die Integration der Tests adressiert.

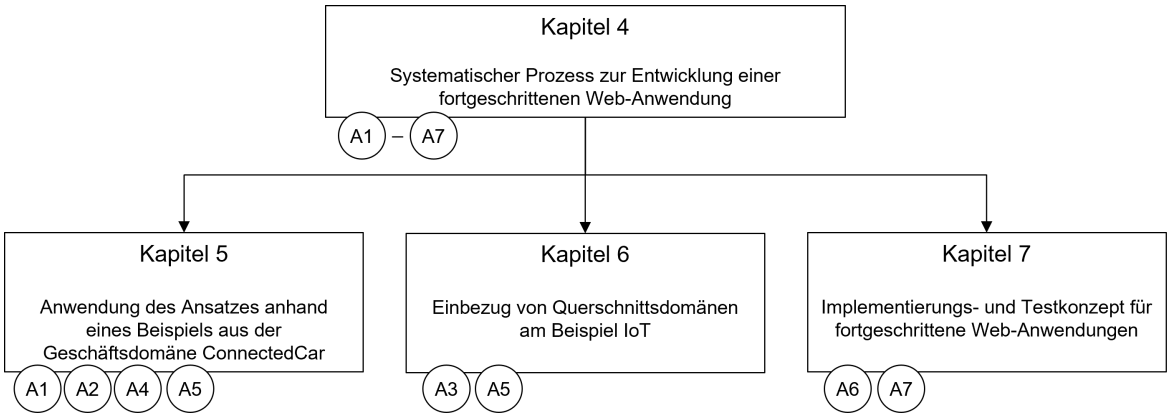


Abbildung 3.9: Behandelte Forschungsbeiträge und deren Anforderungen

4 Systematischer Prozess zur Entwicklung fortgeschrittener Web-Anwendungen

Dieses Kapitel beschreibt einen systematischen, durchgängigen Ansatz zur Entwicklung von fortgeschrittenen Web-Anwendungen, dessen Artefakte strukturerhaltend sind und dadurch gezielt die weiteren Phasen und Artefakte unterstützen. Dies fängt bei der Anforderungsanalyse an, welche vom Entwurf aufgegriffen wird. Neben der Modellierung der Anwendung findet die Modellierung von Domäneninhalten statt. Während die Domäneninhalte anwendungsagnostisch zu betrachten sind, soll die Modellierung der Anwendung weiterhin vollständig auf den Analyseartefakten aufbauen können. Zu Beginn wird in Abschnitt 4.1 die Zielarchitektur präsentiert, die vom Microservice-Engineering-Ansatz erreicht werden soll. Im folgenden Abschnitt 4.2 wird der Teilprozess zur Entwicklung der Domäneninhalte genauer erläutert und der Bezug zur Anwendungsentwicklung wird hergestellt. Der Anwendungsentwicklungsprozess wird anschließend in Abschnitt 4.3 näher ausgeführt. Anschließend wird in Abschnitt 4.4 auf die wichtigsten Implementierungsaspekte eingegangen, bevor im Kapitel 5 die Umsetzung des Microservice-Engineering-Ansatzes mit weiteren Details an einem Beispiel aufgezeigt wird.

4.1 Architektur und Ansatz zur Entwicklung einer fortgeschrittenen Web-Anwendung

Dieser Abschnitt beschreibt, wie die Entwicklung einer fortgeschrittenen Web-Anwendung systematisch erfolgen kann. Hierbei spielt die resultierende Architektur eine entscheidende Rolle für die Aufstellung des Entwicklungsprozesses für fortgeschrittene Web-Anwendungen.

Als Grundlage für die Architektur wird auf den in den Grundlagen in Abschnitt 2.4.2 vorgestellten API-led-Connectivity-Ansatz [Mul-API-led] aufgebaut. Die verschiedenen Typen von APIs (System API, Process API, Experience API) aus dem Ansatz lassen sich in eine Softwarearchitektur und die verschiedenen Ebenen einordnen. Für die Betrachtung der Architektur spielen hierbei jedoch die Softwarebausteine, im Rahmen einer Microservice-Architektur die Microservices, eine zentrale Rolle. Abbildung 4.1 zeigt, wie die verschiedenen Softwarebausteine in die Zielarchitektur einsortiert werden. Die Zielarchitektur folgt dem Muster "Layered Architecture" des domänengetriebenen

Entwurfs [Ev04], siehe Abschnitt 2.3.2. Der zugehörige Microservice zu den Process APIs wird entsprechend der Layered Architecture als Anwendungs-Microservice (engl. Application Microservice) bezeichnet.

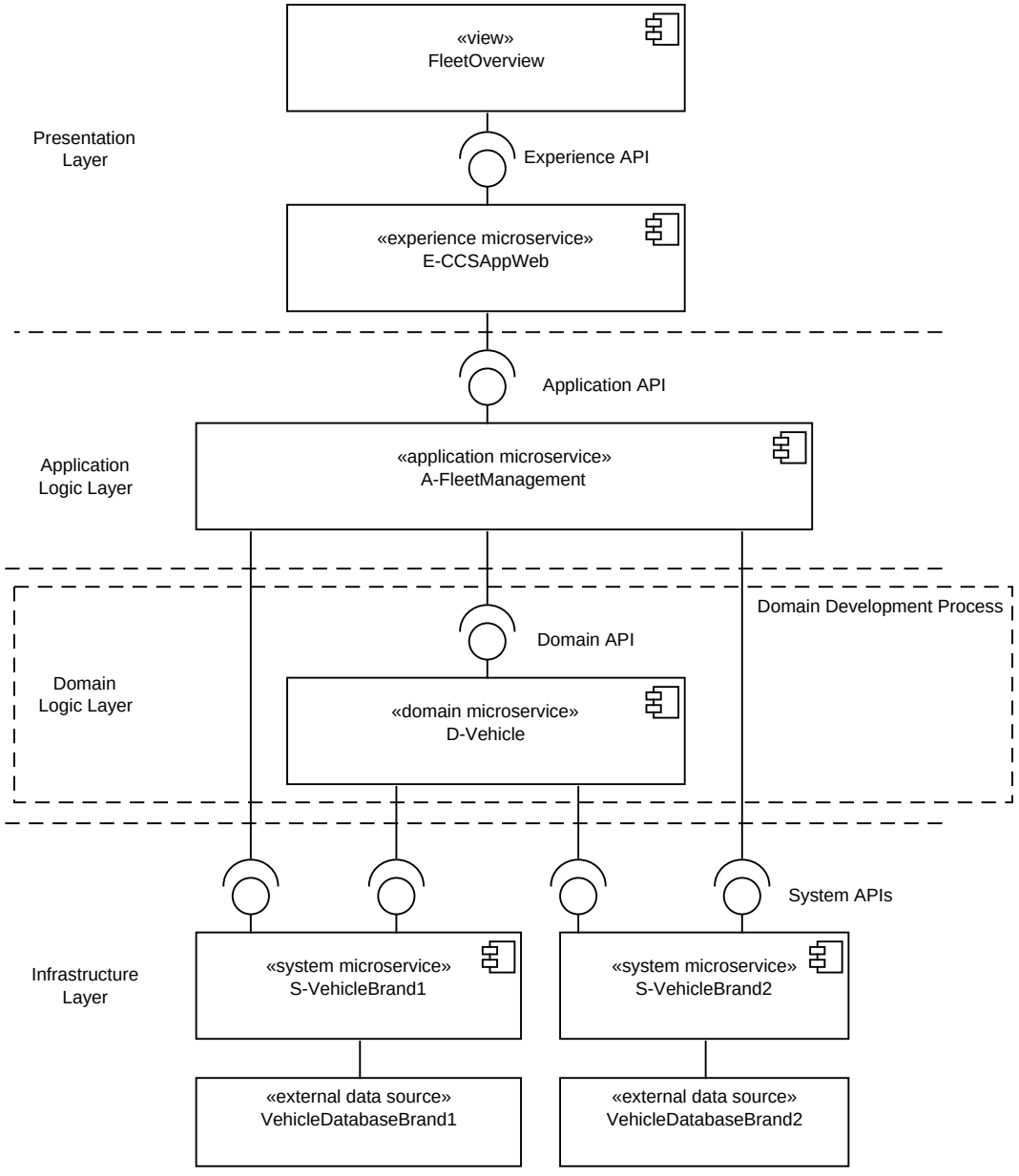


Abbildung 4.1: Architektur für fortgeschrittene Web-Anwendungen am Beispiel einer Anwendung

Bei den iPaaS-Lösungen steht die Integration von bestehenden Systemen im Vordergrund, aber nicht die Domänenschicht. So sieht beispielsweise die iPaaS-Lösung von MuleSoft keine Domänenschicht

vor. In der betrachteten Architektur spielt diese Schicht jedoch eine wichtige Rolle, da genau in dieser Schicht die wiederverwendbaren und anwendungsagnostischen Softwarebausteine platziert werden. Daher führen Schneider et al. [SA23] eine zusätzliche Domänenschicht ein um dieses Defizit zu beheben. Der eingeführte Engineering-Ansatz berücksichtigt ebenfalls dieses Detail, da dadurch sichergestellt werden kann, dass die modellierte Fachlichkeit und die dazugehörenden Softwarebausteine wiederverwendet werden können.

Der Engineering-Ansatz berücksichtigt somit neben der Anwendungsschicht die Domänenschicht. Hierbei wird zu den Inhalten der Anwendung die Fachlichkeit (die Domäneninhalte) festgehalten. Um dies zu bewerkstelligen, wird der Ansatz in zwei Prozesse aufgeteilt. Zum einen wird ein iterativer Prozess zur Entwicklung einer Anwendung eingesetzt, der die klassischen Phasen der Softwareentwicklung durchläuft. Zum anderen wird ein Prozess zur Entwicklung der Inhalte der Domänenschicht eingeführt, der die wiederverwendbaren Domäneninhalte möglichst unabhängig von der Anwendung in der Domänenschicht aufspannt. Die Domänenmodellierung sollte hierbei bei der Entwicklung weiterer Anwendung aus der Domäne unterstützen und diese vereinfachen, da die hierdurch entwickelten Domäneninhalte wiederverwendet werden können. Hierdurch ergibt sich dann insbesondere ein Mehrwert, wenn verschiedene Anwendungen in der Domäne entwickelt werden sollen. Durch das aufgestellte, anwendungsagnostische Domänenmodell kann das benötigte und bereits etablierte Domänenwissen herausgezogen und für die eigene Modellierung wiederverwendet werden. Wird das gleiche Architekturmuster für weitere Anwendungen verwendet, eignen sich die bereits implementierten Domänen-Microservices ebenfalls zur Wiederverwendung.

Ein wichtiger Teil des Entwicklungsansatzes ist die Herleitung der einzelnen Artefakte, wie beispielsweise der Architekturkomponenten. Eine strukturerhaltende Verwendung und Überführung der Artefakte in die nächsten Schritte des Prozesses bietet dafür die Grundlage. Abbildung 4.2 stellt den Ansatz im Überblick dar, der in die oben genannten zwei Prozesse aufgeteilt ist. Der auf der linken Seite dargestellte Teil des Ansatzes, der Prozess zur Entwicklung der Domäne, beginnt unabhängig von der Entwicklung der Anwendung. Hier ist das Ziel, mittels dem domänengetriebenen Entwurf (engl. Domain-Driven Design, (DDD)) [Ev04] und dem sogenannten Knowledge Crunching, die Domänenelemente und Funktionalitäten zu extrahieren. Abschnitt 4.2 beschreibt diesen Teilprozess und ermöglicht einen Einblick in die Artefakte. Als Resultat werden wiederverwendbare Domänenmodelle gewonnen. Ein in dem Kontext eingeführtes Artefakt ist die sogenannte Context Entity Relation View. Dieses Artefakt modelliert die Fachlichkeit der Domäne ähnlich zu einem Klassendiagramm.

Der Anwendungsentwicklungsprozess hingegen beschäftigt sich mir der Anforderungsanalyse, dem Entwurf und der Implementierung der gewünschten (Web-)Anwendung. Hier wird zu Beginn möglichst unabhängig von der Domänenmodellierung die Anforderungen aufgestellt, die die Anwendung erfüllen soll. Gemäß dem Anforderungskatalog aus Abschnitt 3.1 und der Anforderung A1 zur Überführung der Analyseartefakte in den Entwurf sind die Anforderungen entsprechend festzuhalten. Für

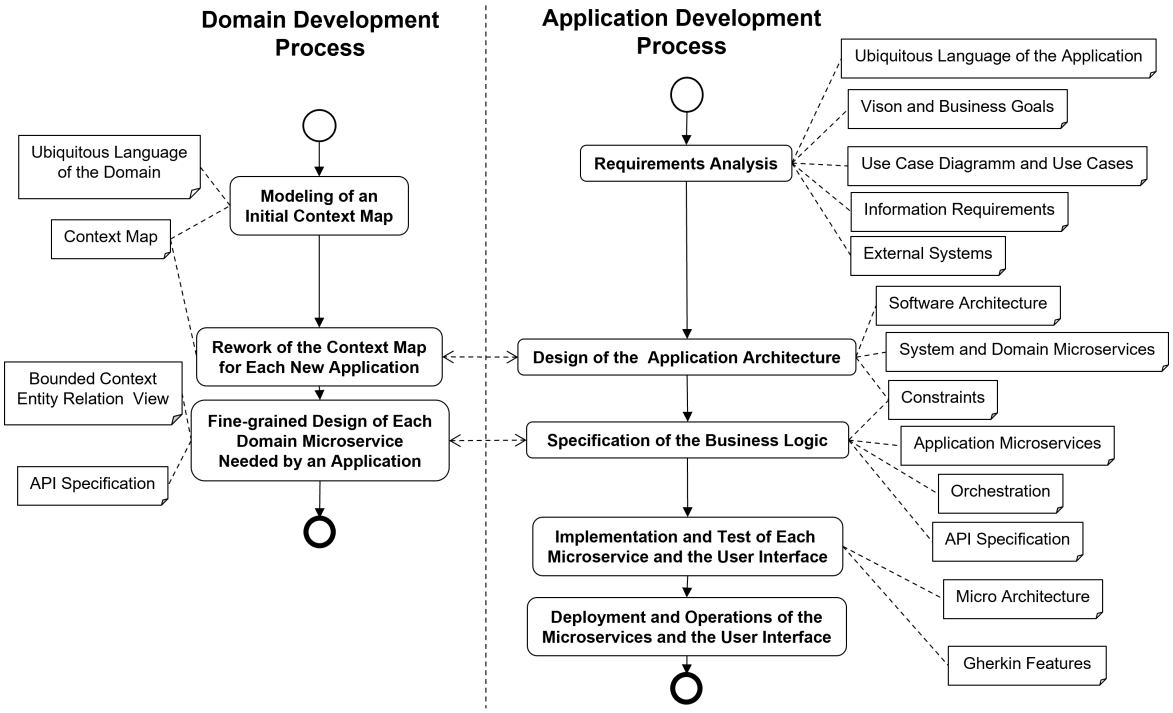


Abbildung 4.2: Übersicht des Ansatzes zur domänengetriebenen Entwicklung von Anwendungen

die Dokumentation der Anforderungen bietet die Literatur verschieden Möglichkeiten, wie beispielsweise mittels der verhaltensgetriebenen Softwareentwicklung (engl. Behavior-Driven Development, BDD) und der Beschreibung der funktionalen Anforderung durch sogenannte Gherkin Features [Sm14]. Eine weiter Möglichkeit sind Anwendungsfälle (engl. use cases), die die Anforderungen als eine Beschreibung der Interaktionen zwischen Aktoren und Systemen festhalten [JC+92, Gl13]. Während der Anforderungsspezifikation soll gemäß Evans [Ev04] stets die gleichen Begrifflichkeiten verwendet werden, weshalb anhand des Artefakts der ubiquitären Sprache diese festgehalten und definiert werden. Die Begrifflichkeiten der abzubildenden Geschäftsdomäne sind dabei führend [Gi18].

Bezüglich der Begrifflichkeiten wird sich bei der Aufstellung der Analyseartefakte daher aus er definierten ubiquitäre Sprache bedient, so dass die eingeführte Terminologie der Domäne sich auch in den Artefakten der Analyse wieder findet. Während den einzelnen Phasen werden verschiedene Artefakt erstellt, die als Eingabe für die weiteren Artefakte oder Phase dienen. Der Engineering-Ansatz ist dabei so aufgestellt, das die oben in Abbildung 4.1 dargestellte Zielarchitektur erreicht wird. Ein weiterer Aspekt ist die Auslieferung und der Betrieb der Anwendung. Dieser soll kurz erwähnt werden, wird aber nicht weiter in der Arbeit betrachtet.

Wie bereits angesprochen, sollte bevor die Entwicklung einer Anwendung erfolgt ein grundlegendes Verständnis über die Domäne etabliert werden. Dies erfolgt, indem ein eigenständiger Teilprozess zur

Entwicklung der Domäne eingeführt wird. Weiterhin bestehen Verknüpfungen zwischen den Prozessen, die im nachfolgenden weiter geklärt werden. Das Herausziehen der wiederverwendbaren Inhalten ist mit einem gewissen Aufwand verbunden und lohnt sich insbesondere dann, wenn verschiedene Anwendungen in der gleichen Domäne entwickelt werden sollen. Hierbei kann eine Erkenntnis sein, dass das Herausziehen der anwendungsagnostischen Funktionalität erst dann sinnvoll ist, wenn mehrere Anwendungen in der Domäne entwickelt wurden, bzw. entwickelt werden sollen.

4.2 Prozess zur Entwicklung der Domäne

Der Domänenentwicklungsprozess hat das Ziel, die dritte und vierte Anforderung (A3, A4) und zu erfüllen. Es wird somit das Domänenwissen erfasst und dieses geeignet von der Anwendungslogik zu separiert um wiederverwendbare Microservices auf der Domänenebene zu erhalten. Der erste Schritt zum Domänenwissen besteht darin, die zu untersuchende Domäne zu verstehen. DDD [Ev04] liefert hierfür Konzepte und Muster für die Aufstellung des Domänenmodells. Abbildung 4.3 erfasst, wie das Muster der Layered Architecture mit dem Entwicklungsprozess zusammenhängt. DDD liefert die notwendigen Muster um eine bestimmte Domäne zu erfassen, berücksichtigt aber nicht die Wiederverwendbarkeit der Domäneninhalte. In dieser Arbeit werden durch den Domänenentwicklungsprozess die Domäneninhalte erfasst und gekapselt. Das Ziel ist es diese Funktionalität so zu kapseln, so dass diese Inhalte nicht immer wieder neu entwickelt werden müssen. Die so erzielten wiederverwendbaren Services, lassen sich für eine Menge an Anwendungen aus der Domäne einsetzen.

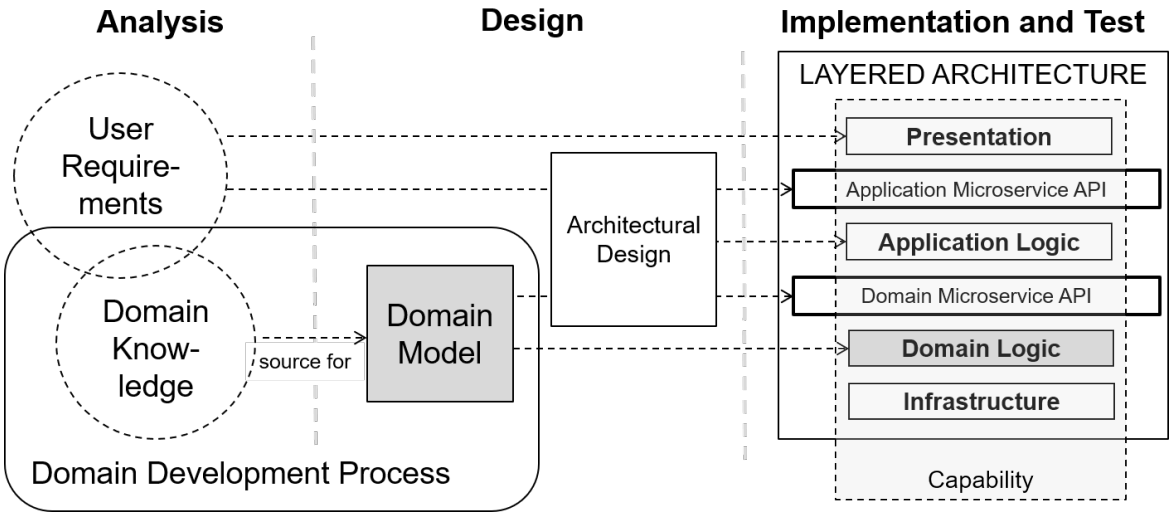


Abbildung 4.3: Zusammenhang der Entwicklungsansätze und der Einordnung in die Layered Architecture

Das relevante Domänenwissen ist hierbei in einer Vielzahl von Quellen vorhanden. Die wichtigsten

Quellen sind Fach- bzw. Domänenexperten, veröffentlichte Quellen wie Normen, aber auch Prozessbeschreibungen der Organisation und Veröffentlichungen bzgl. der Domäne (wie beispielsweise Publikationen zum Thema Connected Car).

Es ist wichtig zu beachten, dass das Domänenmodell (engl. domain model) seinen Ursprung in der Domäne und insbesondere vom Wissen innerhalb der Domäne geprägt wird. Das Domänenmodell wird nicht direkt aus den Benutzeranforderungen abgeleitet. Das Domänenmodell ist in den Benutzeranforderungen in der Weise enthalten, dass die Geschäftslogik der Anwendung neben der anwendungsspezifischen auch die anwendungsunabhängige Domänenlogik enthält. Das Domänenmodell liefert die Grundlage zu den Domänen-Microservices und deren Schnittstellen (Application Programming Interface, API), während die Anwendungs-Microservices und deren Schnittstellen aus aus der Anforderungsanalyse (insbesondere den Anwendungsfällen) abgeleitet werden.

Das Domänenmodell selbst konzentriert sich auf das Domänenwissen, das für die Bereitstellung der Funktionalität des Softwaresystems erforderlich ist. Hierbei ist jede Art der Modellierung gewünscht und eine Formalisierung wird nicht angestrebt [HG+17]. Eine präzise und eine möglichst formale Beschreibung in Form eines Domänenmodells ist die größte Herausforderung beim Entwurf des Softwaresystems.

In der Layered Architecture wird die Domänenlogikschicht durch das Domänenmodell abgedeckt. Diese Schicht stellt das Herzstück des zu entwickelnden Softwaresystems dar. Das Hauptziel des Entwicklungsprozesses für die Domäne besteht darin, das Modell zu erfassen. Ein weiterer wichtiger Punkt ist, dass die Implementierung und das entworfene Modell in der Domänenlogikschicht möglichst übereinstimmen. Wie in Abbildung 4.3 wird gemäß der geschichteten Architektur (engl. Layered Architecture) aus DDD für jede der Schichten eine eigene Microservice API angestrebt.

In Bezug auf die Anforderungen sind die folgenden Punkte zu bestimmen, um anwendungsgnostische Information aus einer Capability aufzustellen. Hierbei ist zu erfassen, welche Entitäten oder Aktivitäten relevant und in der Domäne vertreten sind. Als weiterer Punkt ist zu beantworten, welches Wissen daraus domänenspezifisch oder anwendungsspezifisch ist, da im Domänenmodell nur die Aspekte aufgenommen werden sollten, die nicht spezifisch für die betrachtete Anwendung sind. Als letzter Punkt ist zu bestimmen, welche Beziehungen zwischen den identifizierten Identitäten bestehen. Im Grundsatz lassen sich drei Arten von Beziehungen unterscheiden Hierbei handelt es sich um eine allgemeine Assoziationen, eine Enthaltenseinbeziehung (wie Komposition und Aggregation), sowie eine Vererbungsbeziehung.

Ein Domänenmodell deckt verschiedene Modellierungsaspekte ab. Die strategische Modellierung zielt auf die Strukturierung der Domäne. Das Artefakt der Context Map ermöglicht eine solche Strukturierung und kann als vorbereitender Entwurfsschritt verstanden werden, bevor mit dem Entwurf des Anwendungsentwicklungsprozesses begonnen werden kann. Durch die Context Map wird das Wissen einer Domäne so erfasst, dass die Domänenlogikschicht entsprechend der Struktur der Domäne

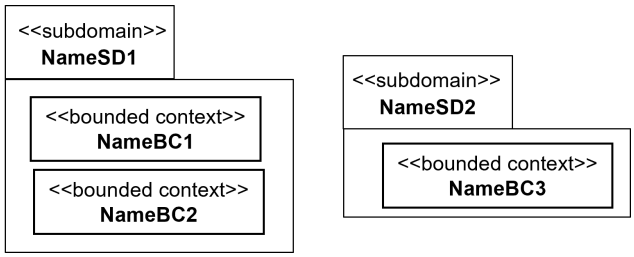


Abbildung 4.4: Context Map einer Domäne

gestaltet wird. Dadurch wird das Domänenwissen so aufbereitet, dass es in eine Microservice-Architektur überführt werden kann.

Dies geschieht, indem die Context Map die Domäne aufspannt und diese in Subdomänen gliedert. Hippchen et al. verwenden für die Modellierung der Context Map ein UML-Komponentendiagramm, welches mit einem entsprechenden UML-Profil erweitert wird [HS+19]. Abbildung 4.4 zeigt zwei Subdomänen (mit SD1 und SD2 abgekürzt) die einen oder mehrere Bounded Contexts (BCs) enthalten. Hierbei kann eine Domäne aus beliebig vielen Subdomänen, mit beliebig vielen Bounded Contexts bestehen. Die wichtigsten Quellen zur Aufstellung der Context Map sind Domänenexperten und Quellen wie Prozessbeschreibungen, Standards und Publikationen. Eine konkrete Context Map für die Domäne ConnectedCar wird von Abeck et al. [AS+19] behandelt, die basierend auf den verschiedenen Quellen erfasst wurde.

Abbildung 4.5 zeigt beispielhaft eine initialisierte Context Map aus der Domäne ConnectedCar, die in der Arbeit weiter vertieft wird. Die Domäne ist in die drei Subdomänen DrivingService, DrivingEnvironment und Driving gegliedert.

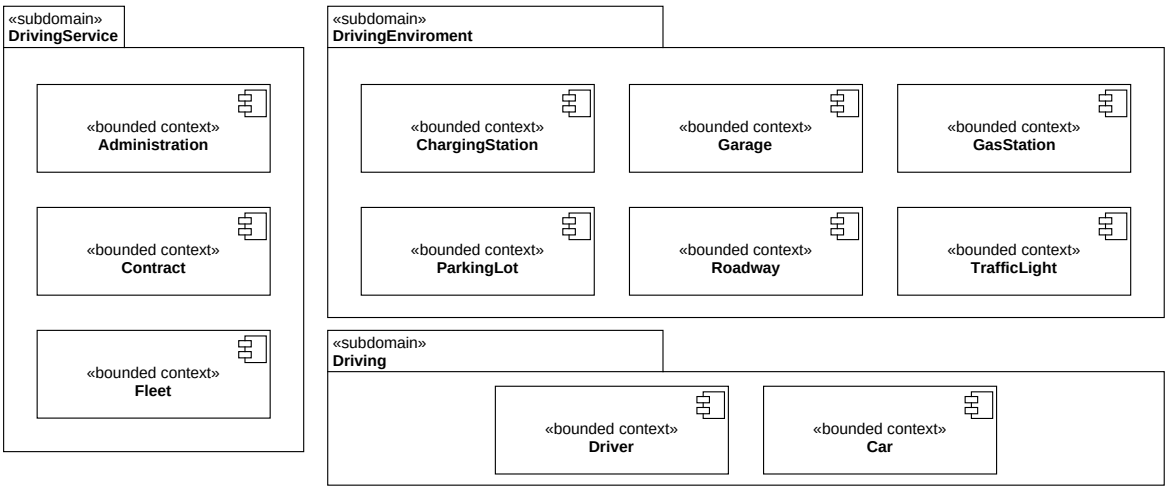


Abbildung 4.5: Context Map der Domäne ConnectedCar

Jede der Subdomänen enthält wiederum Bounded Contexts, die die Fachlichkeit kohäsiv gruppieren und jeweils ein Kandidat für einen Microservice [Ne15] darstellen. Dieser wird anhand der auftretenden Domänenobjekte und deren Beziehungen zu anderen Domänenobjekten weiter spezifiziert. Weiterhin kann das in der Domäne auftretende Wissen durch OCL-Constraints [OMG-OCL] verfeinert werden.

Bei einer Anwendungsentwicklung können verschiedene Domänen involviert sein. Grundsätzlich gilt, dass nur diejenigen Domäneninhalte (wie Context Maps und Bounded Context Entity Relation Views) entworfen werden müssen, die noch nicht modelliert sind. Die bestehenden Modelle werden bei der Anwendungsentwicklung berücksichtigt und bei Bedarf angepasst oder erweitert.

Ein Bounded Context, der Teil einer Context Map ist, wird nur dann als Domänen-Microservice entworfen und implementiert, wenn eine zu entwickelnde Anwendung diesen Domänen-Microservice benötigt. Hierbei wird beim Prozess zur Entwicklung der Domänen das Domänenwissen gebündelt, die Umsetzung des Microservices selbst geschieht bei der Anwendungsentwicklung. Ist der benötigte Domänen-Microservice bereits vorhanden, kann dieser von einer zu entwickelnden Anwendung wiederverwendet werden. Hierfür kann es notwendig sein, dass der Domänen-Microservice für die Bedürfnisse an die zu entwickelnde Anwendung erweitert wird, so dass die Wiederverwendbarkeit der Funktionalität und der Daten erhalten bleibt. Ebenfalls wichtig ist hierbei ein geeignetes Versionskonzept, da verschiedene Anwendungen unterschiedliche Versionen der Domänen-Microservices verwenden können. Um an das Domänenwissen zu gelangen, wird das sogenannte Knowledge Crunching angewendet, welches der Entwickler gemeinsam mit den Domänenexperten durchführt. Aber auch andere Quellen wie Dokumentationen und existierende Systeme können das Wissen über die Domäne schärfen. Gerade bei modernen Web-Anwendungen besteht der Bedarf, bestehende externe Systeme zu integrieren. Dies liegt daran, dass für das Geschäft bereits verschiedene Softwaresysteme von verschiedenen Hersteller im Einsatz sind, die weiter verwendet werden sollen [Da03]. Ein gutes Beispiel für die Unternehmen sind hier Systeme zur Ressourcenplanung mit ERP-Systemen (engl. Enterprise Resource Planning, ERP), die eine wichtige Grundlage für die Geschäftsprozesse bilden [CT+05]. Daher ist die Integration ein Punkt, der in Betracht gezogen werden sollte. Das daraus gewonnene Wissen wird mit den Domänenexperten diskutiert und fließt in die Gestaltung der Bounded Context Entity Relation Views mit ein.

Abbildung 4.6 zeigt den Zusammenhang des Anwendungsentwicklungsprozesses mit dem Entwicklungsprozess für die Domäne. Ein wichtiger Bezug ist die Verwendung des etablierten Wissens aus der Domäne. Die Anforderungsanalyse ermittelt, welche Informationen von der Anwendung überhaupt benötigt werden. Eine wichtige Untersuchung sind die domänenspezifischen Inhalt, welches in die Bestimmung der benötigten externen Systeme mit einfließt. Es werden noch nicht die konkreten externen Systeme untersucht, sondern das fachliche Wissen wird betrachtet. Aus fachlicher Sicht wird abstrahiert, was anwendungsagnostische und was anwendungsspezifische Inhalte sind. Daran werden die externen Systeme analysiert und in die Domäneninhalte eingeordnet. Hierunter wird verstanden,

dass die entwickelte Domäne die Grundlage für die Untersuchung für die fachliche Einordnung liefert. Das Ergebnis kann sich auf die Context Map auswirken, da durch neues gewonnenes fachliches Wissen die Domäne weiter strukturiert werden kann. Wichtig ist jedoch, was die Domäne vorgibt und dass die Entwicklung der Anwendung davon inspiriert wird.

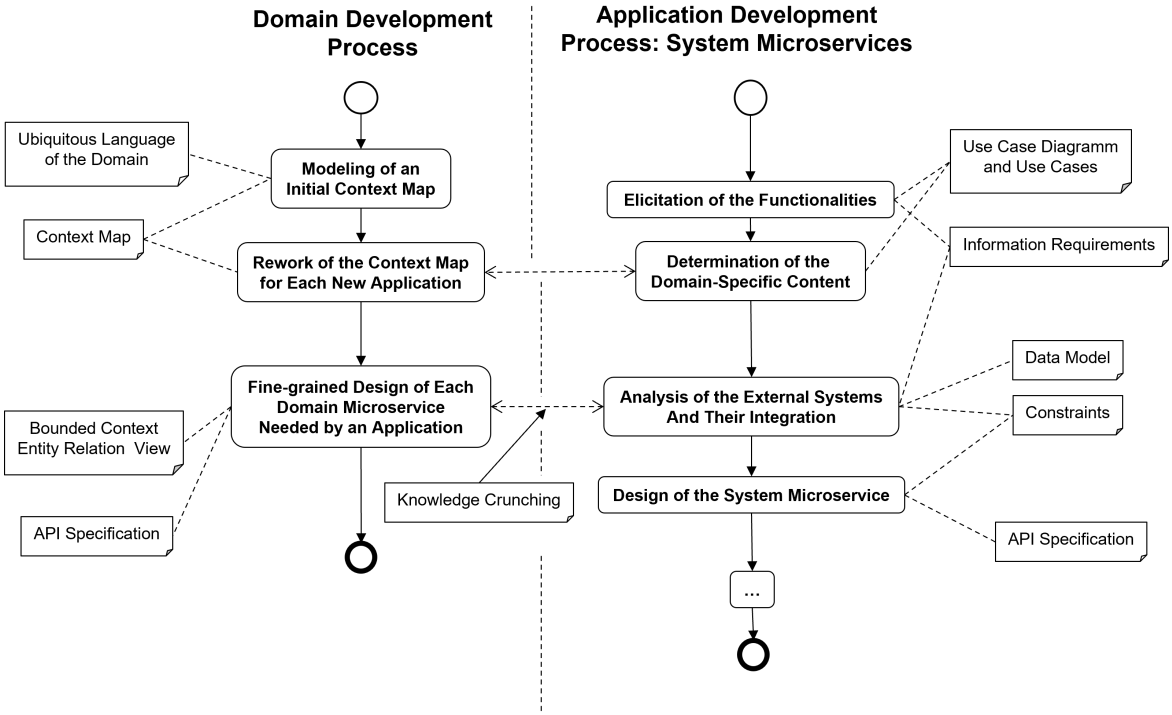


Abbildung 4.6: Zusammenspiel von Anwendungs- und Domänenentwicklungsprozess unter Berücksichtigung externer Systeme

Eine genauere Untersuchung des externen Systems, insbesondere die von dem System bereitgestellten Daten anhand eines Datenmodells, liefert die Grundlage für die System-Microservices, die im Anwendungsentwicklungsprozess weiter spezifiziert werden. Das resultierende Modell und die Schnittstelle dieses System-Microservices soll sich, wie oben beschrieben, an der Domäne orientieren und die Fachlichkeit aufgreifen.

Durch die feinere Untersuchung des externen Systems lassen sich gegebenenfalls auch neue Erkenntnisse bezüglich des in der Domäne betrachteten Domänenausschnitts gewinnen. Dies reicht von bisher nicht betrachteten Domänenobjekten bis hin zu Attributen, die zu einem Domänenobjekt gehören. Konzepte die daraus gewonnen werden, sollten im Anschluss mit einem Domänenexperten besprochen werden. Zudem muss bei diesem Vorgang darauf geachtet werden, dass nur die für die Domäne relevanten Inhalte ergänzt werden.

Eine wichtige Grundlage, die ebenfalls bei der Aufstellung der Domäne geliefert wird, ist die ubiqui-

täre Sprache. Hierbei wird die in der Domäne auftretende Terminologie der Domäne festgehalten, bei der jeder Begriff nicht nur festgehalten, sondern zusätzlich durch eine Beschreibung definiert wird. Bei der Entwicklung der Anwendung fließt die ubiquitäre Sprache aus der Domäne bei der Erstellung der Artefakte mit ein. In Abschnitt 5.1 werden beispielsweise die Begrifflichkeiten aus der Domäne ConnectedCar bei der Aufstellung der Artefakte für die Anforderungsanalyse verwendet.

4.2.1 Taktischer Entwurf der Bounded Contexts

Die Bounded Context Entity Relation View modelliert die in einem Bounded Context befindlichen Domänenobjekte und deren Beziehungen analog zu einem UML-Klassendiagramm. Hierbei wird bei der Modellierung auf das erweiterte UML-Profil [SH+19] zurückgegriffen, welches die Stereotypen der verschiedenen DDD-Muster einführt. Im Diagramm werden die Klassen durch die Stereoptypen «entity» und «value object» ergänzt. Die Modellierung selbst erfolgt ähnlich zu einem UML-Klassendiagramm. Die Entities und Value Objects werden mittels Beziehungen verbunden. Zusätzlich zu den Klassen werden ebenfalls die Attribute und Methoden modelliert. Die festgehaltenen Strukturen der Bounded Context Entity Relation View müssen am Ende möglichst unverändert im Quellcode des umgesetzten Domänen-Microservice implementiert werden können [GH+18].

Um die Inhalte der Bounded Context Entity Relation View zu gelangen, wird der betrachtete Domänenausschnitt analysiert. Hierbei werden gemäß DDD [Ev04] Domänenexperten herangezogen, aber auch Dokumente und weitere Quellen in die Betrachtung aufgenommen. Die auftauchenden Begrifflichkeiten werden in der ubiquitären Sprache der Domäne festgehalten und wie oben beschrieben analog zu einem Klassendiagramm modelliert und mit den Stereotypen versehen. Durch Diskussionen mit dem Domänenexperten wird das resultierende Modell weiter verfeinert, bis die gewünschten Domäneninhalte festgehalten sind. Richtlinien für die Bestimmung, welche Domäneobjekte zu welchem Bounded Context zugeordnet werden, lässt sich anhand der Bestimmung der Kohäsion zwischen Domänenobjekte gemäß Hippchen et al. [HS+19] und Tune [Tu19] bestimmen. Abbildung 4.7 zeigt exemplarisch die modellierte Bounded Context Entity Relation View des Bounded Contexts Vehicle. Hierbei wurde bei Modellierung sich auf ein Fahrzeug mit dem Typ "Auto" (Car) beschränkt.

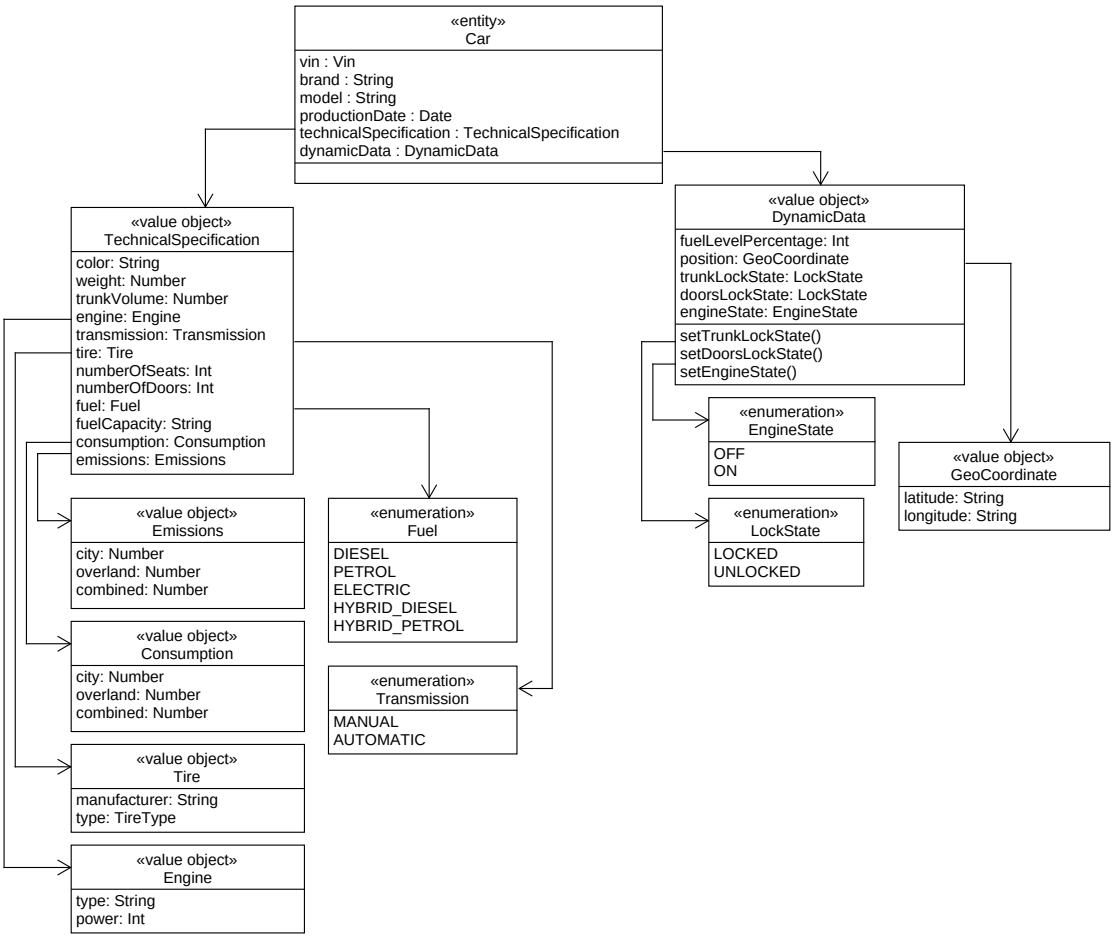


Abbildung 4.7: Auszug aus der Modellierung der Bounded Context Entity Relation View

4.2.2 Formalisierung

DDD lässt offen, wie die Modellierung einer Domäne erfolgen kann. Es werden zwar Konzepte zur Modellierung der Domäne gegeben, aber keine genaueren Aussagen, wie das Domänenmodell modelliert werden soll. Das führt dazu, dass Entwickler das Domänenmodell unterschiedlich interpretieren und modellieren. Allerdings wird betont, dass die Implementierung das erstellte Domänenmodell repräsentieren soll. Für die Entwicklerteams ist es jedoch vorteilhaft, wenn die modellierten Artefakte nicht immer unterschiedlich sind. Ohne einen systematischen Modellierungsansatz lassen sich die erstellten Modelle daher nur bedingt in den Code überführen und sind daher für die Implementierung nur bedingt geeignet. Weiterhin ist keine automatische Code-Generierung aus nicht formalisierten Modellen möglich [RS+18]. Aus diesem Grund ist eine Formalisierung der DDD-Artefakte mit beispielsweise der Unified Modeling Language (UML) sinnvoll.

Für eine Formalisierung der DDD-Konzepte beschreiben Rademacher et al. [RS+17] ein UML-Profil. Dieses UML-Profil beschränkt sich jedoch auf die Elemente der taktische Modellierung. Für die strategische Modellierung sieht dieses UML-Profil keine Elemente vor. Die beiden wichtigsten Elemente der Context Map sind Subdomänen und Bounded Contexts, sowie die Beziehungen zwischen diesen. Das eingesetzte UML-Profil für die strategische Modellierung ist in Abbildung 4.8 dargestellt. Hierbei wird das von Hippchen et al. [HS+19] verwendete UML-Komponentendiagramm und dem UML-Profil zurückgegriffen.

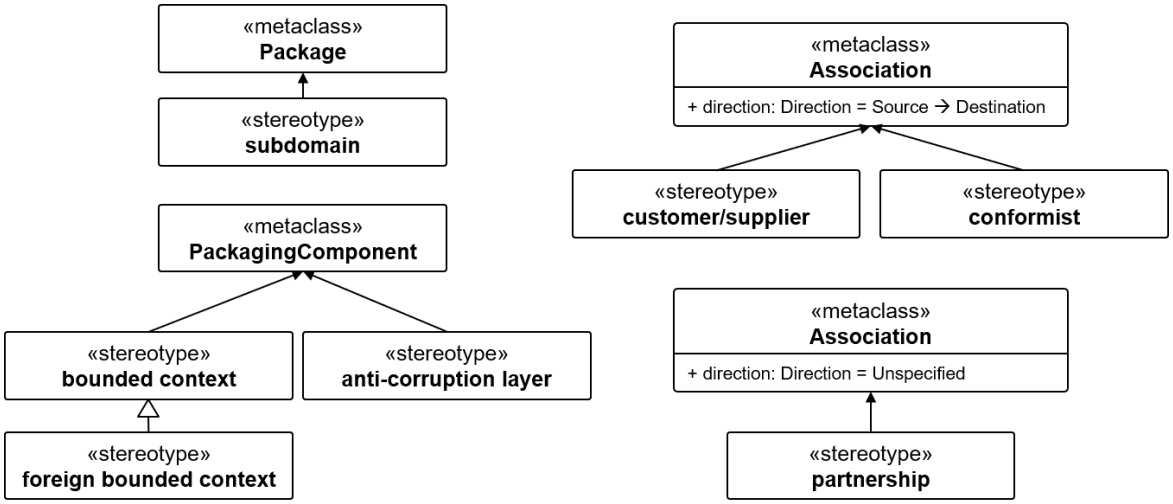


Abbildung 4.8: UML-Profil für die Context Map

Für die verschiedenen Metaklassen (Package, PackagingComponent, Association) werden durch die verschiedenen Stereotypen erweitert. Es werden die Stereotypen «subdomain, «bounded context», «anti-corruption layer», «foreign bounded context» eingeführt. Bei der Erweiterung der Metaklasse für die Assoziationen wird die Navigierbarkeit der Kante mit betrachtet. So wird die Richtung (engl. direction) des Pfeils bei einer Customer/Supplier-, sowie einer Konformist-Beziehung von der Quelle zum Ziel bestimmt. Im Falle einer Partnerschaft ist aufgrund der gleichgestellten Beziehung [Ev14] eine ungerichtete Kante vorgesehen. Gemäß Hippchen et al. [HH+19] werfen hier nicht nur die technischen Verhältnisse dargestellt, sondern vielmehr die organisatorischen Aspekte. Dies betrifft dann bei den verschiedenen Domänen-Microservices jeweils den konkreten Einsatz der Microservices bei der Anwendungsentwicklung. Da die Context Map aber in dem Kontext der Arbeit einen Überblick über die Servicelandschaft liefert, wurde bewusst auf die Relationen zwischen den Bounded Contexts (siehe Abbildung 4.5 verzichtet.

Die Metaklasse für ein Paket (Package) wird in UML für die Strukturierung von Modellelementen verwendet. Ein solches Element beinhaltet beispielsweise weitere Pakete, Klassen und weitere UML-Elemente. Zwischen diesen Paketen sind Abhängigkeits- und Enthaltenseins-Beziehungen in Form von Paketimporten und der Mitgliedschaftsbeziehung möglich. Eine PackagingComponent

ist im Gegensatz eine eigenständige Softwarekomponente. Bei dieser stehen die Modularisierung, Wiederverwendung und Ausführbarkeit im Vordergrund [BH+04].

Eine alternative zur Erweiterung des UML-Profils ist die schwergewichtigere Variante eines Metamodells. Dieses beschreibt die Struktur des betrachteten Modells und liegt eine Hierarchieebene über diesem Modell. Für die Entity Relation View müsste ein Metamodell auf der M2-Ebene erstellt werden. Es könnte auch das Metamodells des Klassendiagramm (was auf der M2-Ebene angeordnet ist) als Grundlage verwendet werden. Bei der Verwendung eines existierenden Metamodells wird jedoch ein tiefgreifendes Verständnis des bestehenden Modells und der Metamodellierung benötigt. Ohne dieses Verständnis können Fehler entstehen die zu Wechselwirkungen führen können, die bei der Erstellung nicht bemerkt werden [RQ+12]. Daher wird die leichtgewichtigere Variante des UML-Profils gewählt. Dies erlaubt den Einsatz von Werkzeugen, die die Modellierung mit einem UML-Profil unterstützen. Schneider et al. setzen die Profile gezielt für ein Modellierungswerkzeug mit dem Namen "Enterprise Architect" ein [SH+19].

Semantische Präzisierung der Bounded Context Entity Relation View

Neben den modellierten Klassen müssen auch die Randbedingungen einer Domäne festgehalten werden. Beispielsweise folgt die Fahrzeug-Identifikationsnummer (FIN) (engl. Vehicle Identification Number, VIN) eines Fahrzeugs bestimmten Vorgaben und Randbedingungen, die eingehalten werden müssen. Dieses zu ergänzende Domänenwissen soll hierbei möglichst formal zur Bounded Context Entity Relation View ergänzt werden, damit sich das Modell später möglichst einfach umsetzten und testen lässt. Eine Möglichkeit hierzu bietet die Object Constraint Language (OCL) [OMG-OCL]. Mittels OCL lässt sich das erstellte UML-Diagramm zur Bounded Context Entity Relation View weiter präzisieren. Ein Constraint ist hierbei eine Einschränkung des Zustands oder des Verhaltens einer Entität in einem UML-Diagramm. Bei den drei wichtigsten OCL-Constraints für UML handelt es sich um Invarianten, Pre- und Postconditions. Eine Invariante muss von jeder Instanz des Modells erfüllt werden. Eine Precondition muss vor der Ausführung einer Methode erfüllt sein. Die Postcondition hingegen ist ein Constraint, welches nach der Ausführung der Methode erfüllt sein muss. Diese OCL-Constraints können auch gezielt von der natürlichen Spezifikation abgeleitet werden [BB+10]. Dadurch lässt sich eine Überführung des Domänenwissens in Constraints vornehmen.

Für die systematische Aufstellung der Constraints ist es zuerst jedoch notwendig, den Sachverhalt innerhalb des modellierten Domänenauschnitts zu verstehen und festzuhalten. Andernfalls ist ein gemeinsames Verständnis nicht sichergestellt. Dies sollte wieder in enger Zusammenarbeit mit dem Domänenexperten geschehen [Ev04].

4.2.3 Entwurf der Schnittstelle

Die Funktionalität eines Domänen-Microservice wird in der Regel über eine REST API bereitgestellt. Bei der Modellierung eines Domänen-Microservice bewegt sich das Modell um eine zentrale Domänenentität. Die Domänenmodellierung wird wiederverwendet und die Umsetzung geschieht bei der Anwendung für den gewählten Architekturstil. Im Rahmen der betrachteten Anwendungsarchitektur wird ein sogenanntes API-Diagramm eingeführt, welches die Ableitung der API vereinfacht. In der Regel haben der Domänen-Microservice und seine zentrale Domänenentität den gleichen Namen. Die benötigten Microservice-Operationen sind in der Regel Create-, Read-, Update- und Delete-Operationen (CRUD) auf der Domänenentität. Zur Modellierung der API werden die REST-Prinzipien [Fi00] eingesetzt. Die REST-Prinzipien sind unabhängig von konkreten Protokollen wie HTTP. Da sich die HTTP-Operationen aber gut eignen, lässt sich beispielsweise jede Microservice-Operation als HTTP-Operation (z. B. GET, POST) spezifizieren und die Ressource wird durch einen einheitlichen Ressourcen-Zeiger (engl. Unified Resource Locator, URL) adressiert. Hierbei wird eine Entität und dessen Methoden aus dem API-Diagramm in die entsprechende Ressource überführt und die benötigten Endpunkte spezifiziert. Das genaue Vorgehen wird in Abschnitt 4.3.2 erläutert.

4.3 Anwendungsentwicklungsprozess

Der Anwendungsentwicklungsprozess durchläuft iterativ die Phasen des Entwicklungsprozesses. Angefangen bei der Analyse werden zu Beginn die Anforderungen festgehalten. Bei der Anforderungsanalyse werden zunächst die Vision und die Geschäftsziele der der Anwendung definiert, um die Erwartungen in frühen Entwicklungsphasen besser zu verstehen. Im vorgestellten Ansatz werden die Anforderungen anschließend basierend auf den Anwendungsfällen spezifiziert. Hierbei werden die erstellten Anwendungsfälle soweit modifiziert, dass sich diese für eine strukturerhaltende Überführung in die Architektur ermöglichen, wodurch die Anforderung A1 zur strukturerhaltenden Überführung der Analyseartefakte in den Entwurf adressiert wird. Bei der Aufstellung der notwendigen Modelle wird zusätzlich auf bereits etabliertes Domänenwissen zurückgegriffen, was ebenfalls bei der Modellierung berücksichtigt wird. Die Artefakte der Domäne, wie die Context Map [Ev04], werden parallel zur Anforderungsanalyse der Anwendung überarbeitet. Der Anwendungsentwicklungsprozess überführt anschließend die Analyseartefakte gezielt in den Entwurf überführt wird. Im Mittelpunkt des Entwurfs stehen die APIs, welche von den weiteren Microservices genutzt werden. Die Wiederverwendung der Domäneninhalte spielt ebenfalls eine wichtige Rolle, da hier Modellierungselemente und Microservices wiederverwendet werden. Letztendlich werden die Entwurfsartefakte in die Implementierung überführt, welche auch entsprechend der Anforderungen getestet wird.

4.3.1 Analyse

Bei der Entwicklung einer fortgeschrittenen Web-Anwendung steht zu Beginn die Analyse des zu entwickelnden Software-Systems an. Hierbei werden die Anforderungen, die die Software erfüllen muss, festgehalten, so dass die gewünschte Software entwickelt wird.

Bei der Betrachtung der Analyse und der zugehörigen Artefakte steht bei dieser Arbeit insbesondere im Fokus, dass diese sich für eine strukturierte Überführung in den Entwurf eignen. Hierzu werden existierende Ansätze aus dem aktuellen Stand der Forschung herangezogen und geeignet für den Ansatz adaptiert.

Bei der Entwicklung werden in der Regel verschiedene Begriffe verwendet, um einen Sachverhalt zu beschreiben. Schwierig wird es insbesondere dann, wenn die Begrifflichkeiten unterschiedlich verstanden werden oder Synonyme für die Begriffe eingesetzt werden. Daher wird bei der Analyse eine Terminologie, die sogenannte ubiquitäre Sprache [Ev04, Sm14] verwendet, um ein klares Verständnis der Begriffe zu schaffen. Die ubiquitäre Sprache enthält am Ende die Begriffe, die in allen Analyseartefakten (und auch in allen Artefakten der folgenden Phasen) konsistent verwendet werden.

Die Vision eines Projekts beschreibt die übergeordnete Problemstellung und Leitlinien eines Projekt. Die Vision dient zu verstehen, welches Softwaresystem und Funktionalität von welchen Benutzern benötigt wird, um das bestehende Problem zu lösen. Durch Geschäftsziele lassen sich messbare und positiven Einfluss auf das Geschäft des Kunden erfassen, für den die Software entwickelt wird. Im nächsten Schritt wird eine Anwendungsskizze erstellt. Anhand dieser Anwendungsskizze lassen sich die Beziehungen zwischen Subjekten und Objekten auf einer hohen Abstraktionsebene beschreiben. Die Beziehungen zwischen den Subjekten und Objekten stellen die Interaktion zwischen diesen dar. Durch eine Gruppierung in der Anwendungsskizze lassen sich Capabilities ableiten, die das Softwaresystem erfüllen soll. Eine Capabilitiy gibt den Benutzern oder Stakeholdern die Möglichkeit, ein bestimmtes Geschäftsziel zu erreichen oder eine bestimmte Funktion erfüllen.

Die funktionalen Anforderungen an das Softwaresystem lassen sich dann erfassen und einsortieren. Hierfür existieren verschiedene Möglichkeiten, diese zu erfassen. In der Literatur werden verschiedene Möglichkeiten genannt, um die verschiedenen Anforderungen zu spezifizieren [Eb19]. Zum einen gibt es die Möglichkeiten, diese mit der verhaltensgetriebenen Softwareentwicklung (Behavior-Driven Development, BDD) zu erfassen. Hierbei werden die Anforderungen über sogenannte Gherkin-Features erfasst. Ein Problem mit dem Ansatz ist, dass dieser sehr offen ist und es viele Möglichkeiten gibt, die Anforderungen einheitlich zu erfassen. Das liegt unter anderem daran, dass eine Systematik fehlt, wie die konkreten Daten beschrieben werden, aber auch wie mit den verschiedenen Fällen von Werten umgegangen wird. Dadurch wird ein Überblick über die Anforderungsspezifikation schwierig.

Eine weitere Möglichkeit für die Anforderungsbeschreibung stellen die sogenannten Anwendungsfälle dar [La04]. Anwendungsfälle spezifizieren die erforderliche Funktionalität der Anwendung aus der Sicht der Benutzer und zwar so, dass ein Vertrag zwischen dem Benutzer und dem System definiert wird und das System als Blackbox betrachtet wird. In dem hier verfolgten Entwicklungsansatz stehen die Anwendungsfälle im Mittelpunkt der Anforderungsanalyse und spielen eine wichtige Rolle bei der gesamten Entwicklung. Anhand der Anwendungsfälle werden somit die Anforderungen spezifiziert. Hierzu liefert die Literatur Hinweise, wie Anwendungsfälle effektiv gestaltet werden können [Co01].

Durch weitere Guidelines können die Anwendungsfälle gezielter formuliert werden. Dadurch erhält der Entwickler zusätzliche Hinweise, um die Anwendungsfälle gut zu gestalten. Hierbei sollten folgende Punkte berücksichtigt werden:

- Durchgängige und konsistente Verwendung der anwendungs- und domänenspezifischen Begriffe
- Externe Systeme werden als sekundäre Akteure modelliert
- Vor- und Nachbedingungen sollten einfache, im Präsens formulierte Behauptungen sein
- Im Fluss (engl. Flow) und in alternativen Flüssen sollten keine UI-Spezifika genannt werden
- Durch die Ergänzung von Informationsanforderungen werden die von den externen Systemen bereitgestellten Informationen festgehalten

Abbildung 4.9 zeigt den Zusammenhang der oben beschriebenen Analyseartefakte auf. Durch ein Anwendungsfalldiagramm lässt sich zudem ein Überblick über die Anwendungsfälle, die Akteure, sowie die beteiligten Systeme aufstellen. In dem Anwendungsfalldiagramm selbst lassen sich die Anwendungsfälle anhand der identifizierten Capabilities gruppieren. Diese Gruppierung bietet den Vorteil, dass bei der Übergang in den Entwurf die kohärenten Funktionalitäten bereits gebündelt sind. Jeder Anwendungsfall innerhalb des Anwendungsfalldiagramms wird dann weiter spezifiziert. Eine konkrete Beschreibung der Inhalte findet sich im Abschnitt 2.2.1.

Da gerade bei fortgeschrittenen Web-Anwendungen auch die Integration externer Systeme ansteht, ist es wichtig, dass ein Anwendungsfall die entsprechenden Informationen enthält, welche geliefert werden müssen. Hierbei kann es sich beispielsweise externe Systeme handeln.

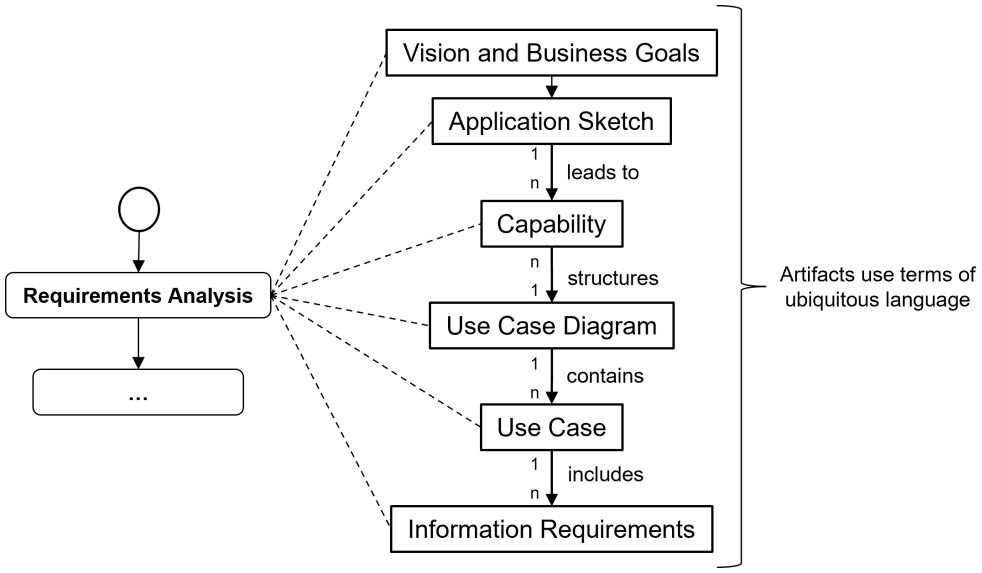


Abbildung 4.9: Zusammenhang der Analyseartefakte

Die Informationsanforderungen sind eine vorgenommene Ergänzung zu den Anwendungsfällen, die aus den Richtlinien von [Co01] abgeleitet wurden. Durch die Informationsanforderungen wird für jeden Anwendungsfall festgehalten, welche Daten bereitgestellt werden müssen. Insbesondere dann, wenn die konkreten externen Systeme und die von ihnen bereitgestellten Daten in der frühen Analysephase noch nicht bekannt sind, werden die benötigten Informationen nur abstrakt definiert. Diese abstrakte Definition der erforderlichen Informationen werden Teil der ubiquitären Sprache um eine konsistente Terminologie zu gewährleisten. Steht ein konkretes externes System, welches die Daten liefert, dann kann dieses angebunden werden. Andernfalls muss im Entwurf hier eine eigene Lösung betrachtet werden.

Im Anwendungsfalldiagramm werden hier zuerst abstrakte externe Systeme eingeführt, die als generische sekundäre Akteure für Anwendungsfälle dienen und die Systemgrenzen durch ihre Interaktionen mit der Anwendung festlegen. Abstrakte externe Systeme dienen dazu die Komplexität und die Wartbarkeit der Anwendungsfallspezifikation zu verringern.

4.3.2 Ableitung des Entwurfs aus den Analyseartefakten

Eine wichtige Anforderung an die Analyseartefakte ist die Eignung zur systematischen Überführung der Artefakte in die oben eingeführte Zielarchitektur (siehe Abschnitt 4.1). Abbildung 4.10 stellt das Zusammenspiel der Artefakte dar. Der Architekturentwurf beginnt mit der zugrunde liegenden Infrastruktur, d.h. der Integration externer Systeme, die der Anwendung die erforderlichen Informationen liefern. Während der Analyse der Anwendung werden über den Informationsanforderungen und die

daraus abgeleiteten abstrakten externen Systeme daraus abgeleitete konkrete externe Systeme identifiziert. Bei der Identifikation wird sichergestellt, dass ein externes System die für den Anwendungsfall notwendigen Informationen liefert. Zu diesem Zweck werden die externen Systeme analysiert und ein objektorientiertes Modell, das System-API-Diagramm, erstellt. Im Falle eines monolithischen externen Systems, wie z.B. Softwarelösungen zur Ressourcenplanung (ERP-Systeme), kann diese Modellierung in mehrere modulare System-APIs zerlegt werden.

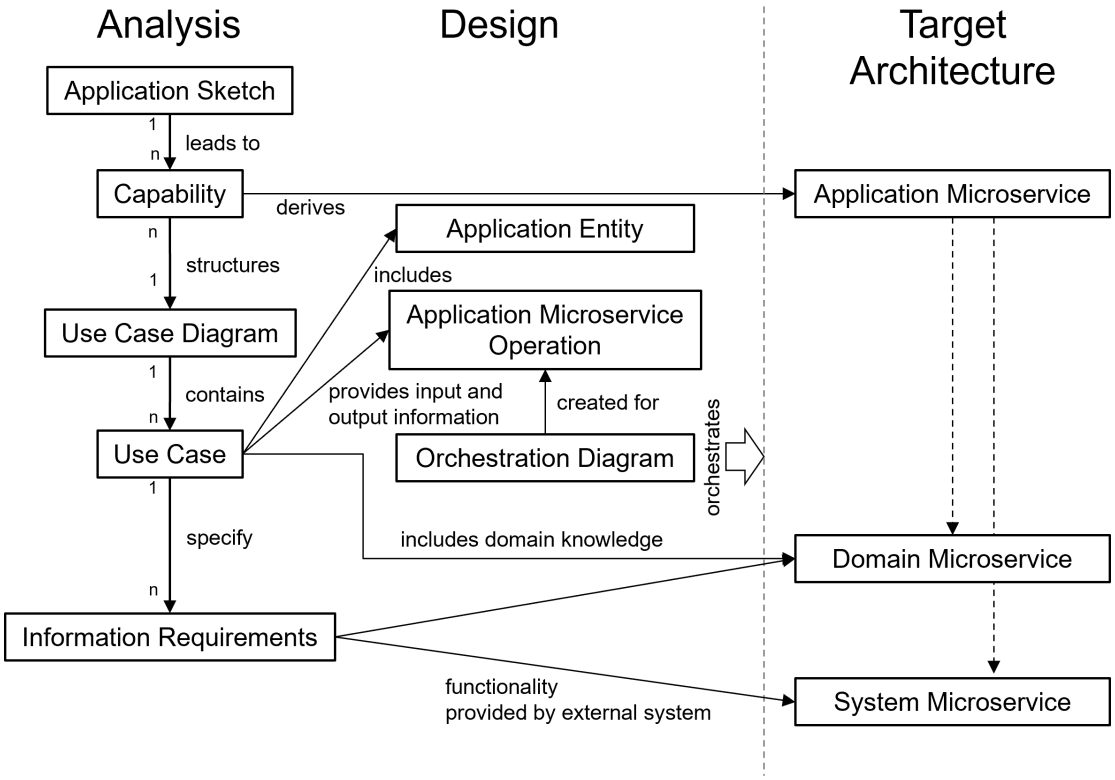


Abbildung 4.10: Ableitung der Architektur aus den Artefakten

Funktionalitäten, die von einem solchen externen System bereitgestellt werden und (noch) nicht in einem zur Anwendung gehörendem Microservice umgesetzt werden sollen, wird anhand der System-Microservices angebunden. Diese Microservices adaptieren die Schnittstelle des externen Systems und stellen eine REST-Schnittstelle bereit, welche die benötigte Funktionalität des externen Systems für den Entwickler in einer verständlichen Art und Weise anbietet und keine Einarbeitung in das externe System notwendig ist. Bei der System API steht hierbei die anzubietende API im Vordergrund, die in der Regel ressourcenorientiert ist. Das bedeutet, bei den System-Microservices werden die Systementitäten und deren Beziehungen durch das entsprechende API-Diagramm festgehalten.

Für die Domäneninhalte ist ein wichtiger Schritt die Bestimmung des anwendungsagnostischen Domänenwissens. Hierfür werden zum einen die Informationsanforderungen verwendet, da diese

abstrakt die benötigten erforderliche Informationen wiedergeben, welche von der Anwendung benötigt werden. Hierbei kommen die etablierten Domäneninhalte ins Spiel, welche die anwendungsagnostischen Inhalte bereits modellieren. Gerade im Rahmen der betrachteten Architektur wird für das in der Bounded Entity Context Relation View dargestellte Modell in ein API-Diagramm für den Ausschnitt überführt. Bei der Überführung werden die fachlichen Inhalte auf die Notwendigkeiten in der API behandelt. Anhand des sich ergebenden API-Diagramms für die Domäne wird dann analog zum Anwendungs-Microservice die benötigte API abgeleitet. Die Inhalte des API-Diagramms der System-Microservices und der Domänen-Microservices bilden die Grundlage für das API-Diagramm der Anwendungs-Microservices.

Aus der Anwendungsskizze lassen sich die Capabilities des Softwaresystems ableiten. Aus jeder Capability lässt sich strukturerhaltend ein entsprechender Kandidat für einen Anwendungs-Microservice ableiten. Zusätzlich wird das Anwendungsfalldiagramm (engl. Use Case Diagram) anhand der Capabilities strukturiert, indem die Anwendungsfälle (engl. Use Cases) gemäß ihrer Zugehörigkeit in die Capabilitys einsortiert werden. Dies vereinfacht die Ableitung der Architektur, da jeder Anwendungsfall bereits der entsprechenden Zugehörigkeit zugeordnet wurde. Bei der Überführung in den Entwurf ergibt Jeder Anwendungsfall eine Microservice-Operation. Die festgelegten Informationsanforderungen (engl. Information Requirements) lassen hierbei Rückschlüsse auf die zusätzlich benötigten Eingabe- und Ausgabeparameter zu.

Bei der in Abschnitt 4.1 vorgestellten Architektur steht am Ende hinter jedem Anwendungsfall ein entsprechender API-Aufruf, der ausgeführt wird. Bei der Anfrage an die Microservice-API und beim Ausführen der entsprechenden Operation werden Parameter benötigt, welche durch die Informationsanforderungen (engl. information requirements) festgehalten werden. Das Ergebnis der Ausführung der Geschäftslogik führt am Ende zu bestimmten Ausgaben. Ein wichtiger Teil der Anwendungsfälle ist, dass diese ebenfalls festgehalten werden. Im Rahmen der betrachteten Architektur bilden diese das Ergebnis des APIs-Aufrufs (die Antwort) der Microservice-Operation ab. Bevor die API spezifiziert werden kann, ist es notwendig, dass die auftretenden Entitäten erfasst und beschrieben werden. Die Entitäten und Beziehungen der Anwendungs-Microservices werden aus der Anforderungsbeschreibung der Anwendungsfälle abgeleitet. Damit die API systematisch spezifiziert werden kann, wird ein Diagramm benötigt, welche die Möglichkeit zur Extraktion der Endpunkte und der benötigten Ressourcen dient. Hierzu werden für die verschiedenen Microservice-Typen jeweils ein sogenanntes API-Diagramm modelliert, welches die Entitäten und deren Beziehungen modelliert. Eine wichtige Eigenschaft der Modelle ist, dass die Inhalte von den darunterliegenden Microservices von darüber liegenden Microservices einbezogen werden. So sind bei dem API-Diagramm zum Anwendungs-Microservice die unterliegenden System- und Domänen-API-Diagramme mit zu berücksichtigen, da die Anwendungs-Microservices die Inhalte der modellierten Domänen-Microservices und System-Microservices zur Realisierung der gewünschten Funktionalität wiederverwenden sollen. Konkret bedeutet dies, dass der Inhalt des Diagramms bei der Modellierung des API-Diagramms des

Anwendungs-Microservice berücksichtigt wird. Am Ende enthält das Diagramm dann alle benötigten Entitäten, deren Beziehungen und benötigten Methoden, inklusive dem Bezug zu den darunterliegenden API-Diagrammen. Konkret handelt es sich dabei um die Entitäten der Domäne sowie zusätzliche Systementitäten. Abbildung 4.11 verdeutlicht dies anhand eines generischen Anwendungs-API-Diagramms. So erbt beispielsweise eine Anwendungsentität die Inhalte einer Domänenentität. Neben der Vererbung können auch einfache Relationen verwendet werden. So zeigt der Pfeil in Richtung der System API eine einfache, gerichtete Assoziation, welche aus dem UML-Klassendiagramm verwendet werden.

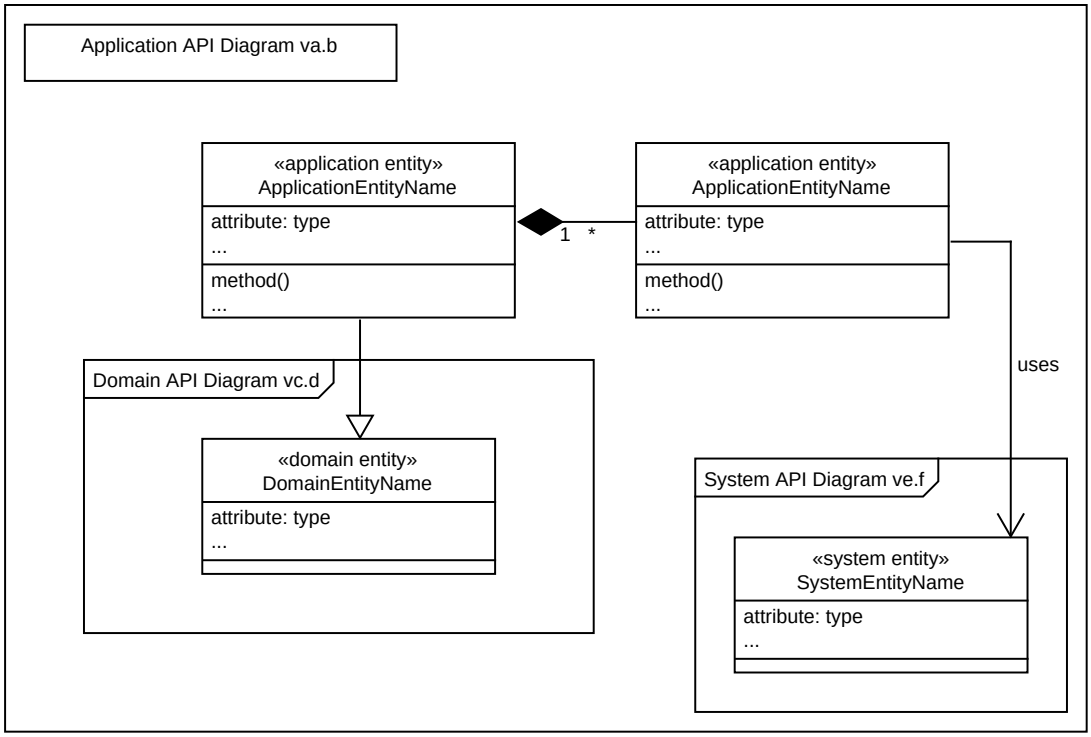


Abbildung 4.11: Aufbau eines API-Diagramms, welches weitere API-Diagramme inkludiert

Eine wichtige Eigenschaft des API-Diagramms sind die Versionsnummern. Jedem API-Diagramm wird eine Versionsnummer zugeordnet. Jedes inkludierte Diagramm wird ebenfalls mit der Versionsnummer angegeben. Die Versionsnummer bestimmt die im betrachteten Ausschnitt verwendete Version der API-Spezifikation auf der Grundlage dieses Diagramms. Das bedeutet, Version va.b des Anwendungs-API-Diagramms baut auf den API-Versionen vc.d des Domänen-API-Diagramms und auf der Version ve.f. des System-API-Diagramms auf. Die Bedeutung der Versionsnummern ist entsprechend der semantischen Versionierung festgelegt [Sem-Sem]. Bei der ersten Ziffer handelt es sich um eine große Änderung (engl. major change), welche die API so verändert das es die

konsumierende Systeme bricht. Die zweite Ziffer bezieht sich auf Änderungen oder Anpassungen die Rückwärtskompatibel (engl. minor change) sind, welche die konsumierenden Systeme nicht beeinflussen.

Das resultierende Anwendungs-API-Diagramm des Anwendungs-Microservices inkludiert somit die Entitäten aus der Anforderungsanalyse und die relevanten Ausschnitte der einbezogenen System- oder Domänen-Microservices und bildet die Grundlage für die Spezifikation des betrachteten Anwendungs-Microservices. Die Ableitung der API erfolgt auf dem API-Diagramm, indem verschiedene Schritte durchlaufen werden. Abgeleitet werden der Endpunkt, die zugehörige primäre Ressource, sekundäre Ressourcen falls vorhanden und die benötigten Parameter und Datentypen. Die detaillierte Ableitung der Schnittstelle eines Anwendungs-Microservices wird im Detail in Abschnitt 4.3.2 behandelt.

Ableitung der API-Spezifikation Durch das API-Diagramm und die systematische Ableitung der API-Spezifikation aus diesem wird die Anforderung A5 adressiert. Die benötigten Endpunkte werden aus den Anwendungsfällen und dem daraus abgeleiteten API-Diagramm aufgestellt. Abbildung 4.12 zeigt einen Überblick der Artefakte und deren Inhalte bezüglich der Ableitung der API-Spezifikation.

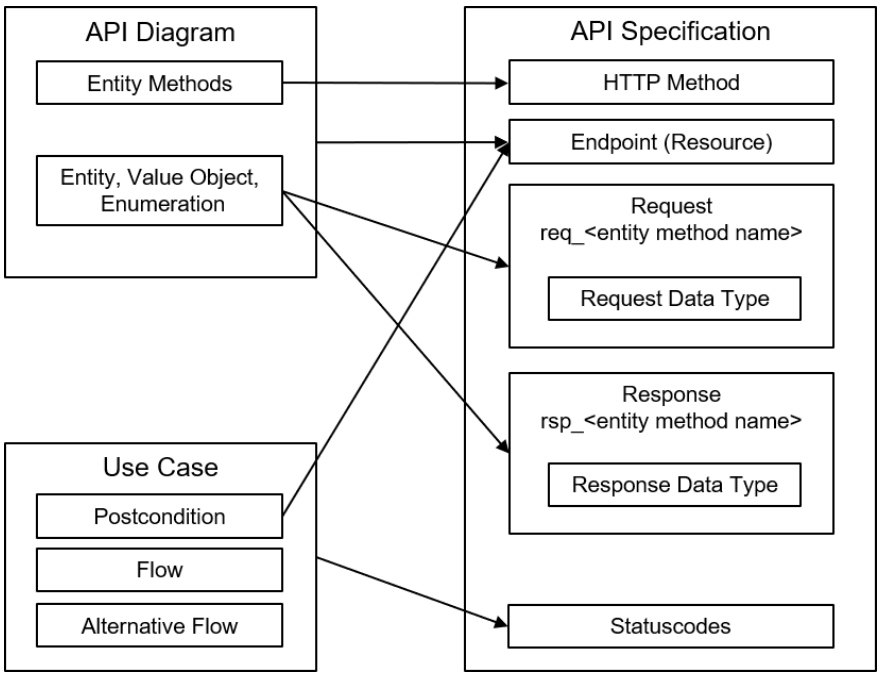


Abbildung 4.12: Von den Anwendungsfällen und API-Diagrammen zur API-Spezifikation

Der Pfad für den Endpunkt ergibt sich aus den involvierten Entitäten aus dem Anwendungs-API-Diagramm. Die primäre Ressource, welche auch für die Ausführung der Methode zuständig ist,

bestimmt den ersten Teil des Pfades. Die sekundäre Ressource ist entweder der Rückgabeparameter oder die zu verändernde Entität. An den bisherigen Pfad wird die ID der primären Ressource gesetzt, welche dann im Anschluss von der sekundären Ressource gefolgt wird. Jede Methode aus dem API-Diagramm wird zu einer ressourcenorientierte Operation überführt. Die HTTP-Methode wird entsprechend nach CRUD ausgewählt [Ya20] und anhand der Interaktion benannt. Wird im Anwendungsfall keine Nachbedingung genannt, wird die HTTP-Operation GET verwendet. Wird durch die Methode etwas erstellt oder gespeichert, wird die Operation POST verwendet. Beschreibt die Nachbedingung eine Änderung, dann wird die HTTP-Methode PUT oder PATCH verwendet. Wird etwas entfernt oder gelöscht, dann wird die HTTP-Operation DELETE verwendet. Tabelle 4.1 liefert einen Überblick über die Auswahl der Methode.

Postcondition	**HTTP-Methode**	**Mögliche Begriffe in der Postcondition**
-	GET	view, see, get
Erstellen einer Ressource	POST	add, create
Änderung einer Ressource	PUT / PATCH	update, change, edit
Löschung einer Ressource	DELETE	remove, delete

Tabelle 4.1: Ableitung der benötigten HTTP-Methode

Die Eingabeparameter werden aus dem Eingabefeld der Informationsanforderungen abgeleitet. Die Rückgabe im Erfolgsfall wird aus dem Ausgabefeld hergeleitet. Hierbei werden Datentypen aus dem API-Diagramm abgeleitet, die in API-Spezifikation definiert werden. Dies geschieht für die Entitäten selbst, sowohl als auch für die Anfragen und Antworten eines API-Aufrufs. Dadurch wird erreicht, dass die verschiedenen Datentypen sich innerhalb der API-Spezifikation wiederverwenden lassen.

Strukturierung der API-Spezifikation Die API-Spezifikation besteht aus Datentypen und API-Operationen. Die vorgeschlagene Struktur beinhaltet, dass, die beiden Aspekte Daten und Operationen voneinander getrennt platziert werden. Analog zu den Konzepten aus DDD werden Unterordner für die Entitäten und Wertobjekte eingeführt. Dies sorgt für eine bessere Strukturierung, insbesondere wenn die Anzahl der Dateien steigt. Dadurch lassen sich bereits definierte Datentypen wiederverwenden.

Orchestrierung der Funktionalität Bei der betrachteten Microservice-Architektur werden von den Anmwendungs-Microservices verschiedene Domänen- oder System-Microservices aufgerufen. Die Kette der Aufrufe wird bei dem Entwurf der Anwendung ebenfalls berücksichtigt. Für die Definition der Geschäftslogik einer Anwendung wird das sogenannte Orchestrierungs-Diagramm (engl. orchestration diagram) eingeführt. Ein Orchestrierungs-Diagramm bezieht sich hierbei auf die Operation eines Microservices und definiert die Anwendungslogik in Bezug auf Funktionalitäten, die durch einen Anwendungsfall (Use Case) vorgegeben werden. Innerhalb des Entwurfs werden für

die verschiedenen Anwendungsfälle Orchestrierungs-Diagramme erstellt. Hierbei wird immer vom zentralen Anwendungsfall ausgegangen. Anwendungsfälle, die einen Anwendungsfall inkludieren (includes) oder erweitern (extends) werden im gleichen Orchestrierungs-Diagramm behandelt. Ein Orchestrierungs-Diagramm beschreibt hierbei eine Kette von Service-Aufrufen und adressiert die dazwischen liegenden Verarbeitungsschritte. Die Grundlage der Orchestrierungs-Diagramme ist inspiriert von der Business Process Execution Language (BPEL) [Ju06] und der Service-oriented Architecture Modeling Language (SoaML) [EB+11]. Ein Orchestrierungs-Diagramm selbst verwendet das UML-Aktivitätsdiagramm als Grundlage. Die Implementierung der Anwendungslogik wird dann aus der API-Spezifikation und den Orchestrierungs-Diagrammen abgeleitet.

4.3.3 Einbezug von Querschnittsfunktionalität

Neben den eigentlichen Geschäftsfunktionalitäten, benötigt eine fortgeschrittene Anwendung in der Regel zusätzliche Funktionalität. Beispielsweise werden Sensordaten benötigt oder ein Benutzer muss authentifiziert und autorisiert werden, bevor er bestimmte Funktionalitäten einer Anwendung verwenden darf. Diese Querschnittsfunktionalität wird in der Regel von weiteren Domänen bereitgestellt. Damit diese Funktionalität nicht für jede Anwendung neu entwickelt werden muss, wird diese Funktionalität ebenfalls aus der Anwendung herausgezogen. Die Funktionalität selbst wird jedoch nicht in der Geschäftsdomäne modelliert, sondern in eine eigene Domäne einer sogenannten Querschnittsdomäne festgehalten. Hierbei wird die Querschnittsdomäne analog zur Geschäftsdomäne aufgestellt, sodass die wesentlichen Aufgaben, die zu erbringen sind, erfüllt werden können. Auch hier ist wichtig, dass die Domäne gemäß dem Entwicklungsprozess für die Domäne aufgestellt wird. Zusätzlich ist es wichtig, dass durch die Beschreibung der Architektur, insbesondere der Softwarearchitektur die Verknüpfung der Geschäftsdomäne und der beteiligten Querschnittsdomänen erfolgt. In Kapitel 6 wird anhand der Domäne IoT erläutert, wie eine Querschnittsdomäne anhand des vorgestellten Prozesses zur Domänenmodellierung erfasst und modelliert werden kann. Eine weitere wichtige Querschnittsdomäne ist das Identitäts- und Zugriffsmanagement, welche von den meisten fortgeschrittenen Web-Anwendungen ebenfalls benötigt wird. Diese Domäne wird in der Arbeit nicht weiter behandelt.

4.4 Implementierung und Testen der Anwendung

Nachdem die Modellierung der Anwendung und deren Komponenten abgeschlossen ist, steht die Implementierung der Artefakte im Fokus. Die strukturerhaltende Überführung der Entwurfsartefakte in die Implementierung sowie das systematische Testen der Anforderungen stehen hierbei im Fokus. Für eine gezielte Überführung der Inhalte ist es sinnvoll, eine geeignete Mikroarchitektur einzuführen.

In der Literatur werden für Microservices die hexagonale Architektur [Co05] und nach Vernon äquivalent die Onion Architecture [Ve13] genannt.

Eine Abwandlung hiervon ist die "Clean Architecture" (übersetzbar mit sauberer Architektur [Ma12]. Abbildung 4.13 zeigt die verschiedenen Ebenen, die als Ringe dargestellt werden. Eines der wichtigsten Konzepte dieser Architektur ist die Festlegung der Abhängigkeiten, welche besagt, dass eine Abhängigkeit nur zwischen einem äußeren Ring seinem nächsten inneren Ring bestehen sollen. Beispielsweise kann der Ring, der für die Operationen zuständig ist, auf die Entitäten aus dem Modell zugreifen. Die Operationen selbst können aber nicht direkt auf die Datenbank zugreifen. In einigen Fällen muss ein innerer Ring auf einen äußeren Ring zugreifen. Zum Beispiel sollen die Ergebnisse der Operationen (bspw. ein Ergebnis nach der Ausführung und somit eine angepasste Entität) in einer Datenbank gespeichert werden. Für diese Abhängigkeiten wird das Prinzip der Dependency Inversion aus der "Clean Architecture" verwendet.

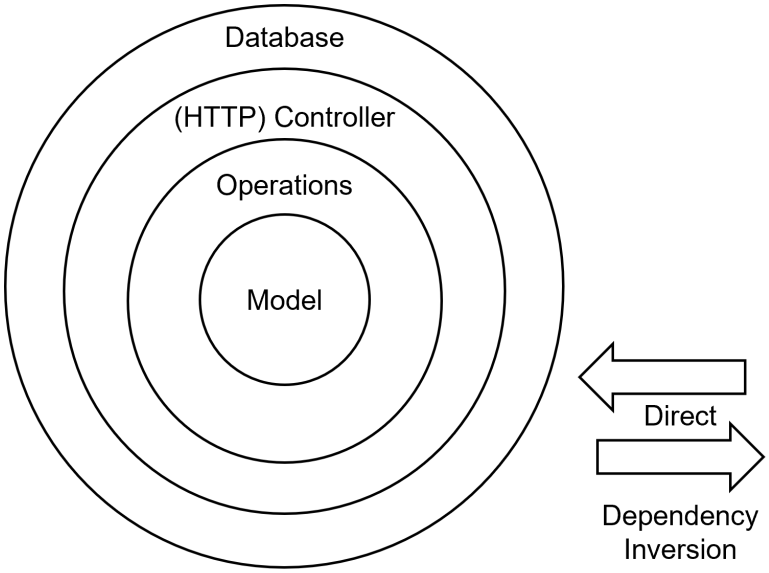


Abbildung 4.13: Abhängigkeiten in der Clean Architecture

Die Microservices umfassen in der Literatur die verschiedenen Schichten wie die Infrastruktur, Domäne, Anwendung und Präsentation. Das Problem besteht darin, dass Domänenlogik und Anwendungslogik nicht getrennt behandelt werden. Innerhalb des Microservices ist die (Domänen-)Logik gekapselt, aber nach außen zur Schnittstelle mit der Anwendungslogik vermischt. Im Rahmen dieser Arbeit werden die Domäneninhalte in eigene Microservices ausgelagert. Dadurch wird keine Schichtung der Anwendung und Domäneninhalte innerhalb des Microservices vorgenommen, sondern die Domänenschicht wir heraus extrahiert. Trotzdem bieten die Microservices nach außen eine Schnittstelle an, die prinzipiell auch direkt von einem Endbenutzer verwendet werden kann. Insbesondere für die Domänen-Microservices passt das Paradigma von REST optimal, da die Entitäten passend als

Ressourcen abgebildet werden können. Als Resultat wird eine hohe Wiederverwendbarkeit erreicht, da die Domänen-Microservices für verschiedene Anwendungen aus der Domäne wiederverwendet werden können. Würde die Domänenlogik nicht heraus extrahiert werden, wäre der Grad der Wiederverwendung geringer.

Als Grundlage dieser Mikroarchitekturen und der Anforderungen an die eigenen Bedürfnisse lässt sich daraus die folgende in Abbildung 4.14 dargestellte Mikroarchitektur ableiten, die für die Überführung der Artefakte in die Implementierung und zugehörigen Tests ermöglicht. Jeder Teil der Mikroarchitektur befindet sich in einem eigenen Verzeichnis.

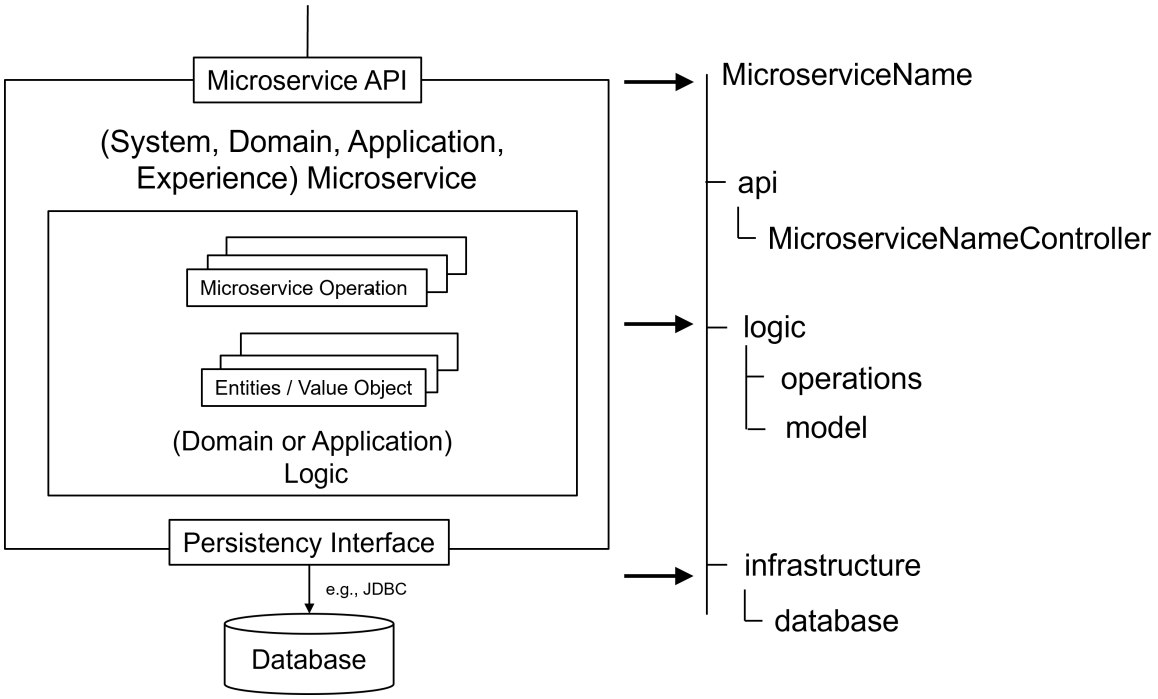


Abbildung 4.14: Eingesetzte Mikroarchitektur

Die interne Struktur der Microservices besteht aus den drei Teilen api, logic und infrastructure.

- Der API-Teil der Mikroarchitektur ist für die Offenlegung der Schnittstelle des Microservices verantwortlich. Die Hauptkomponente des API-Teils ist der Controller, der eingehende Anfragen annimmt, sie an die Logik weiterleitet und die Antwort des Microservices nach außen liefert.
- Der Logikteil enthält das implementierte Modell einschließlich der Entitäten und Wertobjekte sowie die zugehörigen Microservice-Operationen.
- Der Infrastrukturteil stellt dem Microservice eine Persistenzschnittstelle zur Verfügung, über die er auf Datenbanken oder andere Datenspeicher zugreifen kann. Die Persistenzschnittstelle wird in Form eines Repositories implementiert, das grundlegende Funktionen für den Zugriff auf

und die Speicherung von Daten bereitstellt. Diese werden von den Microservice-Operationen angesprochen.

Daraus ergibt sich die klare Positionierung der Funktionen- und Methoden der umzusetzenden Microservices. Für die Entwickler ist einer der Vorteile, dass eine gezielte Platzierung der Implementierungsinhalte möglich ist und die gezielte Überführung der Analyse- und Entwurfsartefakte in die Implementierung unterstützt. Die Einarbeitung in einen bereits bestehenden Microservice wird ebenfalls erleichtert, da die Strukturen bekannt sind. Weiterhin wird die Wartbarkeit ebenfalls dadurch verbessert. Durch die Nutzung von Vorlagen kann hierbei eine zusätzliche Vereinfachung der Implementierung, auch in Richtung von DevOps, erfolgen. Die Mikroarchitektur erlaubt ein strukturiertes Implementierungs- und Testkonzept, welches in Kapitel 7 eingeführt wird. Hierbei werden die Analyse und Entwurfsartefakte systematisch in die Implementierung und Tests überführt, indem die verschiedenen Inhalte an der entsprechenden Stelle umgesetzt werden.

Beispielsweise werden bei den Domänen-Microservices die Bounded Context Entity Relation View im Ordner "model" umgesetzt, und die zugehörigen Operationen und Constraints im Ordner "operations". Analog wird die spezifizierte API und der benötigte API-Controller im Ordner "api" umgesetzt. Was die Tests angeht, wird eine systematische Aufstellung dieser verfolgt, welche ebenfalls in einer gespiegelten Mikroarchitektur umgesetzt werden. Das bedeutet, für jeden umgesetzten Ordner in der Mikroarchitektur wird ebenfalls ein gleichnamiger Ordner für die Tests vorgesehen, welcher Tests für die entsprechende Funktionalität enthält. Die Ableitung der Tests erfolgt anhand der Artefakte, wie den OCL Constraints.

Die Implementierung und das Testen der einzelnen Microservices sollte gemäß Hippchen [Hi21] und der Gründe von Hasselbring und Steinacker [HS17] durch Service-Teams umgesetzt werden. gemäß Bass et al. [BW+15] besteht ein Service-Team aus verschiedenen Rollen, die unterschiedliche Kenntnisse in die Entwicklung des Microservices einbringen. Wichtige Rollen sind hierbei die Programmierer, Tester, aber auch DevOps-Ingenieure. Die Anzahl der Teammitglieder sollte jedoch möglichst gering gehalten werden, um eine gute Effizienz innerhalb des Teams zu erreichen [Ne15, BW+15].

4.5 Verteilung der Domänen-Microservices für Microservice-Anwendungen

Die Domänen-Microservices lassen sich für verschiedene Anwendungen wiederverwenden. In welcher Art die Wiederverwendung erfolgt, hängt von den organisatorischen Bedürfnissen ab. Benötigen verschiedene Anwendungen die gleiche Datengrundlage, dann kann der Microservice für die betroffenen Anwendungen bereitgestellt und eine gemeinsame Datenbasis genutzt. Wird hingegen nur die Logik des Microservices benötigt, aber der Datensatz ist unterschiedlich, dann wird der Microservice mit anderen Parametern (insbesondere einer anderen Datenbank) bereitgestellt. Dies ermöglicht zudem,

dass für eine bestimmte Anwendung auch Modifikationen am Domänen-Microservice vorgenommen werden können, um bspw. weitere Funktionalitäten bereitzustellen. Auf der organisatorischen Ebene ist es somit wichtig, solche Entscheidungen beim Einsatz der Domänen-Microservices zu erfassen. Der Vorteil des Einsatzes liegt in einer einheitlichen Taxonomie, die ein Domänen-Microservice beinhaltet.

4.6 Zusammenfassung

Die Entwicklung einer fortgeschrittenen Web-Anwendung erfolgt über verschiedene Artefakte. Der systematische Prozess, der für die Entwicklung einer fortgeschrittenen Web-Anwendung erforderlich ist, wurde in diesem Kapitel festgelegt und stellt den ersten Forschungsbeitrag (B1) dar. Hierzu wurde zu Beginn der Rahmen anhand der Layered Architecture aus DDD festgehalten. Abbildung 4.15 zeigt die resultierenden Softwarebausteine und die Überführung der Artefakte in die verschiedenen Microservice-Typen.

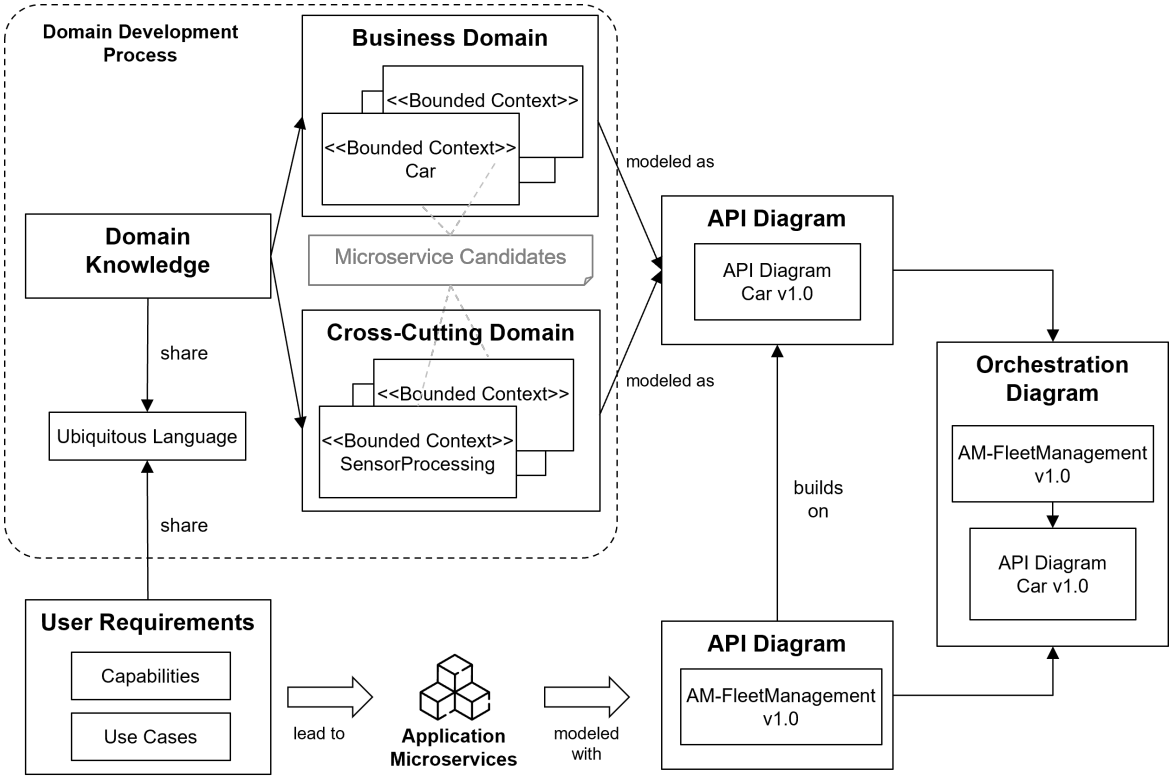


Abbildung 4.15: Überblick über den Gesamtansatz

Die Erfassung der Inhalte für die Domänen- und Anwendungs-Microservices wurde in zwei separate Prozesse aufgetrennt. Hierbei wurde mit dem Prozess zur Entwicklung der Domäne begonnen und

die anwendungsagnostischen Inhalte herausgezogen, die (1) strategisch durch die Context Map und (2) taktisch durch die "Bounded Context Entity Relation View" mittels DDD erfasst wurden. Durch eine einfache Form der Formalisierung eines UML-Profils wurde dafür gesorgt, dass die durch das Domänenmodell festgehaltenen Domäneninhalte in die Implementierung überführt werden können. Dadurch wird insbesondere die Wiederverwendbarkeit von Softwarebausteinen ermöglicht, da diese Bausteine anwendungsagnostisch sind.

Die Entwicklung der Anwendung erfolgte gemäß dem Anwendungsentwicklungsprozess, der sich in die Analyse, den Entwurf und die Implementierung und Test aufteilt. Hierbei war eine strukturerhaltende Überführung der Artefakte von zentraler Bedeutung. Der Ansatz inkludiert die Überführung der Analyseartefakte in den Entwurf. Anhand der benötigten Anforderungsfällen wird ein API-Diagramm erstellt, welches im nächsten Schritt für die Ableitung der benötigten Schnittstelle herangezogen wurde. Hierbei bedient sich ein API-Diagramm eines Anwendungs-Microservices bei den davor modellierten API-Diagrammen der Domäne, so dass die dort definierten Entitäten wiederverwendet werden. Dadurch werden die Softwarebausteine (Microservices) bei der Anwendungsentwicklung einbezogen.

Die Orchestrierung der Anwendung erfolgte durch das eingeführte, Orchestrierungs-Diagramm. Dieses Diagramm verknüpft die einzelnen Microservices der Domäne, der Anwendung, aber auch der benötigten Querschnittsfunktionalität. Damit lassen sich die Abläufe und Aufrufe zwischen den beteiligten Microservices in der Gesamtarchitektur der Anwendung dynamisch abbilden. Zuletzt werden im Prozess die aufgestellten Analyse- und Entwurfsartefakte in die Implementierung überführt. Als Grundlage für jeden Microservice diente die entworfene Mikroarchitektur, die eine Annäherung an die Clean Architecture darstellt. Während in diesem Kapitel nur grundlegend die Implementierungsstruktur geklärt wurde, wird in Kapitel 7 genauer auf die Implementierung und das Testen dieser eingegangen.

5 Anwendung des Ansatzes anhand eines Beispiels aus der Geschäftsdomäne ConnectedCar

Dieses Kapitel greift den in Kapitel 4 eingeführten Microservice-basierten Entwicklungsansatz für fortgeschrittene Web-Anwendungen auf und demonstriert anhand der Geschäftsdomäne ConnectedCar und der Anwendung ConnectedCarServicesApplication (CCSApp) die Entwicklung erfolgt. Zuerst werden in Abschnitt 5.1 die Anforderungen ermittelt. Hierbei wird nicht nur auf die benötigte Funktionalität eingegangen, sondern die externen Systeme sowie die anwendungsagnostischen Inhalte werden ebenfalls beleuchtet. Anschließend folgt in Abschnitt 5.2.1 die Überführung in die Software-Architektur. Als Resultat lässt sich die Software-Architektur aufstellen, die ebenfalls in diesem Abschnitt weiter vertieft wird. Diese besteht aus System-Microservices, welche die externen Systeme integrieren, optionalen Domänen-Microservices, die die domänenspezifische Logik bereitstellen, Anwendungs-Microservices, die aus den Anwendungsfällen abgeleitet werden und Experience-Microservices, welche die Anforderungen der verschiedenen Benutzeroberflächen unterstützen. Der Entwurf inkludiert auch die Ableitung des wichtigen API-Artefakts der einzelnen Microservice-Typen. Der Einbezug der Domäneninhalte und der externen Systeme wird bei der Orchestrierung in Abschnitt 5.2.4 anhand der Orchestrierungs-Diagramme erläutert.

5.1 Anforderungsanalyse

Zu Beginn der Analyse werden die Vision und Geschäftsziele festgelegt. Anhand einer Anwendungsskizze können die auftretenden Begriffe in Zusammenhang gesetzt werden. Weiterhin können Capabilities gebildet werden, die die Geschäftsziele weiter untergliedern. Die Anforderungen werden durch Anwendungsfälle mit einer bestimmten Struktur ausgedrückt. Ein weiterer Gesichtspunkt ist die Berücksichtigung und Integration von externen Systeme (insbesondere bereits bestehende Anwendungen eines Unternehmen) in die zu entwickelnde Microservice-Anwendung. Hierzu ist die Analyse dieser Systeme notwendig. Anschließend kann anhand der Anforderungen, der Analyse der externen Systeme eine optionale domänenspezifische Analyse gemäß dem domänengetriebenen Entwurf (Domain-Driven Design, DDD) durchgeführt werden. Mit der Spezifikation der eigentlichen Anwendungsfälle werden auch die Begriffe der ubiquitären Sprache aus der Domäne verwendet. Nachdem die verschiedenen Artefakte der Anforderungsanalyse aufgestellt sind, werden diese in den Entwurf überführt.

5.1.1 Anwendungsfalldiagramm

Anwendungsfälle spezifizieren die funktionalen Anforderungen des Systems [La04]. Diese werden so spezifiziert, dass ein Vertrag zwischen dem Benutzer und dem System definiert wird. Das System wird hierbei als Blackbox betrachtet [Co01]. Zunächst wird ein UML-Anwendungsfalldiagramm erstellt, welches einen Überblick über die Anwendungsfälle, Akteure und ihre Beziehungen enthält. Dieses Diagramm bietet noch keine tiefere Spezifikation der Anwendungsfälle, sondern dient als Übersicht des Systems und seiner Umgebung.

Das Anwendungsfalldiagramm ist ein wichtiges Artefakt für die Ableitung der verschiedenen benötigten Microservices. Damit eine Ableitung ermöglicht wird, sind die in Abschnitt 4.3.2 beschriebenen Voraussetzungen bei der Modellierung des Anwendungsfalldiagramms einzuhalten. Dadurch lassen sich die verschiedenen Microservice-Typen und später die benötigten Microservice-Operationen ableiten. Die Ableitung der Softwarearchitektur aus dem Anwendungsfalldiagramm wird in Abschnitt 5.2.1 exemplarisch an der CCSApp beschrieben.

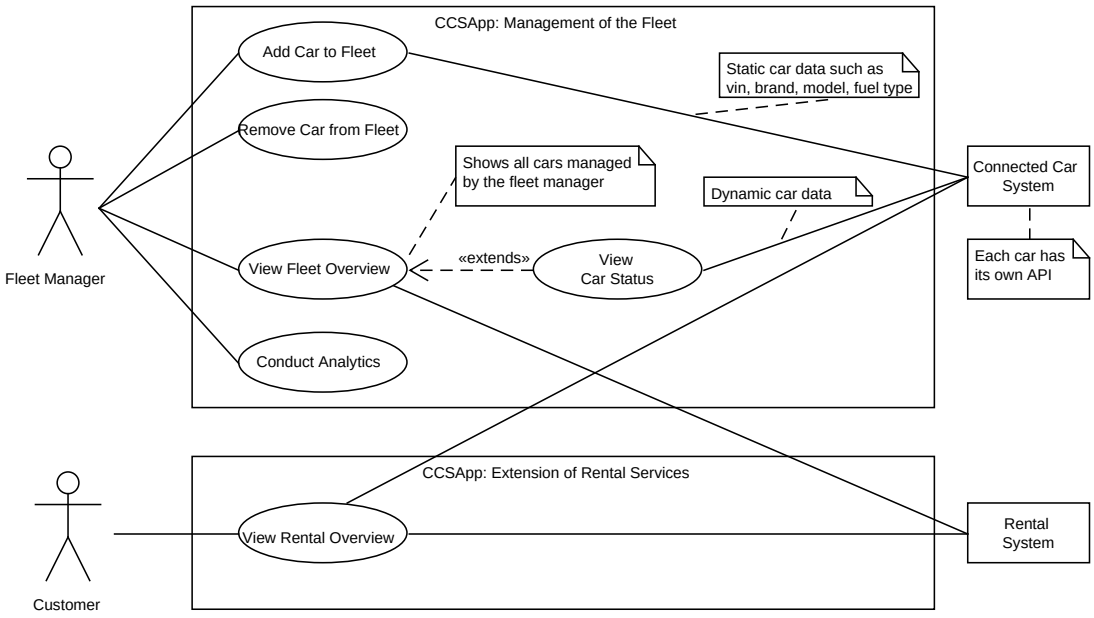


Abbildung 5.1: Aus der Anforderungsanalyse resultierendes Anwendungsfalldiagramm der CCSApp

Die Anwendungsfälle aus Abbildung 5.1 sind in die jeweilige zugehörige Capability gruppiert. Die Capability "Management of Fleet" umfasst die benötigt Funktionaltität eines Flottenmanagers, um die Flotte zu verwalten (d.h. Fahrzeuge hinzuzufügen und zu entfernen) und zu überwachen (d.h. die

relevanten Informationen zu erhalten). Die Fähigkeit "Management of Rental Services" bietet die Möglichkeit Kunden zusätzliche Dienstleistungen für die gemieteten Fahrzeuge bereitzustellen.

5.1.2 Spezifikation der Anwendungsfälle

Die genannten Anwendungsfälle werden weiter spezifiziert, um die Interaktion mit dem Benutzer und dem System zu spezifizieren. Die Beschreibung der Inhalte orientiert sich hierbei an bewährte Verfahren zur Spezifikation [Co01]. Listing 5.1 zeigt den Anwendungsfall für die Funktionalität "View Fleet Overview". Es enthält, für welchen Akteur (in diesem Beispiel der Fleet Manager) der Anwendungsfall gültig ist. Jeder Akteur (primär oder sekundär) wird in der ubiquitären Sprache definiert. Anhand der Vor- und Nachbedingungen werden die Systemzustände vor und nach der Ausführung des Anwendungfalls festgehalten. Neben der Interaktion mit dem System enthält ein Anwendungsfall auch alternative Abläufe und die Informationsanforderungen, welche für die Erfüllung der Funktionalität der Anwendung erforderlich ist.

```
1  Title: Add Car to Fleet
2
3  Primary Actors: Fleet Manager
4  Secondary Actors: Connected Car System
5
6  Preconditions:
7      - System has a fleet registered for the actor
8  Postconditions:
9      - The car with the given VIN is added to the fleet
10     - Static car data on the car with the given VIN is saved
11
12 Flow:
13 1. Actor adds a new car to the fleet by providing the VIN
14 2. System validates that the given VIN is valid and a car with the
       given VIN is not present in the fleet
15 3. System gathers static car data on the car with the given VIN from
       the connected car system
16 4. System adds the car to the fleet with its VIN, brand, model and
       production date
17
18 Alternative flows:
19 2a. Given VIN is invalid
20     2a1. System prompts actor to re-enter the VIN
21 2b. A car with the given VIN is already present in the fleet
22     2b1. System informs the actor that the car is already in the fleet
           and stops the use case
23 3a. System detects failure to communicate with the connected car system
24     3a1. System signals error to the actor and stops the use case
25 3b. Connected car system does not have a car with the given VIN
```

```
26      3b1. System informs the actor that a car with the given VIN does
            not exist and stops the use case
27
28
29  Information Requirements:
30      Connected Car System:
31      -  Static car data: VIN, brand, model, production date
```

Listing 5.1: Use Case View Vehicle State

Durch den Fluss (engl. flow) wird die Geschäftlogik spezifiziert, die erfüllt werden muss. Die Geschäftslogik kann ggf. über mehrere verschiedene alternative Flüsse ebenfalls erfüllt werden. Zusätzlich ist es möglich, dass verschiedene Bedingungen zu einem Fehler im Fluss führen, was zu einem Abbruch und in der Regel die Zurücksetzung des Zustands auf den Ausgangszustand zur Folge hat.

Bei der Erstellung des Artefakts wird auf eine durchgängige und konsistente Verwendung der Anwendungs- und domänenspezifischen Begriffe geachtet. Alle externe Systeme werden als sekundäre Akteure repräsentiert. Die Vor- und Nachbedingungen werden im Präsens formuliert und sollten einfach zu verstehen sein. Weiterhin sollte bei der Erstellung der Anwendungsfälle darauf geachtet werden, dass bei dem Hauptfluss und in den alternativen Flüssen keine Entscheidungen in Richtung der Benutzeroberfläche formuliert wird. Zu den Besonderheiten der Benutzeroberfläche gehören beispielsweise Navigationsschritte, wie das Drücken oder Befüllen bestimmter Oberflächenelemente.

Die Spezifikation der funktionalen Anforderungen mithilfe von Anwendungsfällen ist eine Möglichkeit, die Anfoderungen spezifizieren. Eine weitere Möglichkeit ist die Nutzung der verhaltensorientierte Entwicklung (Behavior-Driven Development, BDD), welche als agile Softwareentwicklungstechnik betrachtet werden [Sm14]. Da das spezifizierte Verhalten die Grundlage von BDD ist, wird es auch als spezifikationsgetriebene Entwicklung bezeichnet. Die verhaltensgetriebene Entwicklung ist hierbei eine Weiterentwicklung von der Testgetriebenen Entwicklung (Test-Driven Development, TDD), welche das gleiche Ziel eines automatisierten und konsistenten Testens der Software verfolgt [No06]. Die Anforderungen selbst werden bei BDD in sogenannten Gherkin-Features festgehalten. Die Sprache Gherkin besteht aus spezifischen Schlüsselwörtern, die zur Definition der Anforderungen verwendet werden. Der Vorteil der Spezifikation liegt darin, dass die Anforderungen direkt in die Tests (über sogenannte Schritt-Definitionen) überführt werden kann. Das bedeutet, die Spezifikation kann leicht in die Testfälle überführt werden. Im Gegensatz zu den Anwendungsfällen bringt das einen gewissen Vorteil. Schwieriger wird es bei der Beschreibung der Anforderungen, da durch das vorgegebene Muster und den großen Freiheitsgrad die Anforderungen von jedem Entwickler anders hingeschrieben werden.

5.1.3 Analyse der involvierten externen Systeme

Bei der Analyse jeder Anwendung werden über die festgehaltenen Informationsanforderungen der Anwendungsfälle und die daraus abgeleiteten abstrakten externen Systeme konkrete externe Systeme identifiziert. Bei der Identifikation wird sichergestellt, dass ein externes System in der Lage ist, die für den Anwendungsfall notwendigen Informationen zu liefern. Zu diesem Zweck werden die externen Systeme analysiert und die gelieferten Daten festgehalten. Damit die Abbildung des externen Systems erfolgen kann, muss dieses zuerst analysiert werden und ein Datenmodell erstellt werden.

Hierbei ist ein wichtiger Gesichtspunkt, wie die externen Systeme in die Anforderungsbeschreibung der Anwendung einzubeziehen sind. Aus fachlicher Sicht ist hier zu klären, welche Rolle das System einnimmt. Bei der Analyse ist somit zu bestimmen, in welchen Szenarios das externe System verwendet wird. Dadurch lässt sich erfassen, welche Credentials, Verträge oder sonstige Eigenschaften (bzw. Funktionalität) von einem solchen System genutzt werden sollen. Im Beispiel der CCSApp werden Daten von zwei verschiedenen Autoherstellern benötigt, damit die gewünschten Fahrzeuginformationen bereitgestellt werden können. Im Falle der CCSApp bieten die Hersteller jeweils ein (unterschiedliches) System an, welches den Zugriff auf die Fahrzeuge und bestimmte Informationen gewährt. Hierbei werden jedoch nicht alle Fahrzeuge inkludiert, sondern es ist nur der Zugriff auf Fahrzeuge erlaubt, zu denen der Flottenmanager auch der Eigentümer ist. In der Sprache der Anwendung bedeutet dies, dass nur Zugriff auf die Fahrzeuge mit den VINs möglich ist, welche im Besitz des Flotten-Managers sind.

Ein weiterer wichtiger Schritt ist die Analyse des externen Systems, auf dem die System-API aufbaut. Abbildung 5.4 beinhaltet neben der Ergänzung zum System-API-Diagramm der Ausschnitt des analysierten externen Systems. Die Abbildung zeigt auch, welche Komplexität das externes System VehicleBrand1Car aufweist. Dies verdeutlicht die Wichtigkeit, dass ein externes System gründlich analysiert werden muss. Die Entität VehicleBrand1Car ist eine spezifische Repräsentation eines Autos des externen Systems und hat grundlegende Attribute wie die vehicleId und die Farbe. Weiterhin verfügt ein Auto über verschiedene Befehle (engl. commands), die auf der Entität ausgeführt werden kann. Jeder Befehl gehört zu genau einem VehicleBrand1Car, da die Entität Command nur dazu verwendet wird, einen Befehl auf einem bestimmten VehicleBrand1Car. Ein Befehl hat eine Bezeichnung und ist in der Aufzählung CommandName festgehalten. Durch einen API-Endpunkt "GET /vehicle/commands" werden die verschiedenen Befehle abgerufen. Ein entsprechender Befehl wird mit dem API-Endpunkt "POST /vehicle/commands/{commandname}" ausgeführt. Außerdem hat eine Ressource einen Wert vom Typ "ResourceValue". Ein "ResourceValue" wiederum besteht aus dem eigentlichen Wert und einem Zeitstempel. Diese Analyse dient anschließend als wichtige Grundlage zur Erstellung des System-API-Diagramms.

5.1.4 Analyse der Domäne

Einer der ersten Schritte der Analyse ist die Erfassung der ubiquitären Sprache. Dadurch werden die benötigten Begriffe der Fachlichkeit festgehalten, die auch bei der Entwicklung der weiteren Anwendungsinhalten verwendet werden sollte. Damit ist gemeint, dass die Begrifflichkeiten auf der Anwendungsebene ebenfalls bei der Analyse mit berücksichtigt werden. Durch die zuvor erstellte Context Map lässt sich der Teil der Domänenmodellierung auf der strategischen Ebene einordnen. Ein Domänen-Microservice ist optional und baut auf der Fachlichkeit der Domäne auf. Wie in Abschnitt 4.2 beschrieben, wird hier anhand von Knowledge Crunching im Rahmen des Domänen-Entwicklungsprozesses die Inhalte für den Domänen-Microservice erarbeitet. Basierend auf den modellierten Inhalten wird dann anschließend ein API-Diagramm hergeleitet, welches die Anforderungen an die Architektur erfüllen. Hierdurch lassen sich im betrachteten Kontext auch mehrere heterogene Datenquellen zu einer einheitlichen Schnittstelle zusammenführen. Dies ist insbesondere dann interessant, wenn verschiedene Hersteller semantisch sehr ähnliche Daten anbieten, diese aber in einem unterschiedlichen Format ausdrücken. Im Kontext der CCSApp werden Daten von zwei Automobilherstellern erhalten. Jeder Automobilhersteller bietet zur Bereitstellung der Daten eine eigene API an. Das Datenformat der Fahrzeuge unterscheidet sich jedoch. Durch die Ergänzung eines Domänen-Microservices können die Daten semantisch anhand der anwendungsagnostischen Inhalte vereinheitlicht werden. Fahrzeughersteller, welche noch keine API anbieten, können von der bereits modellierten Domäne profitieren und das Modell als Grundlage für die eigene Modellierung nehmen. Dadurch wird die Heterogenität ebenfalls reduziert.

Bei der Erstellung des Domänenmodells zu dem Ausschnitt des Autos steht die Fachlichkeit des Fahrzeuges im Vordergrund. Das bedeutet, das Modell sollte die Entität in der Domäne repräsentieren und das Verhalten dieser beschreiben, wobei das Modell nur eine Abstraktion der realen Entität ist [St73]. Dadurch wird sichergestellt, dass auch weitere Anbieter ein Modell zu Grunde haben, dass sich für die Beschreibung der eigenen Autos (oder Fahrzeuge) eignet.

5.2 Entwurf

Die einzelnen Bausteine der Architektur werden in den folgenden Abschnitten anhand der zuvor aufgestellten Analyseartefakte, sowie den Erfassten Anforderungen zu den externen Systemen und der Domäne abgeleitet. Basierend auf der strukturierten Anforderungsanalyse lässt sich bereits die Software-Architektur aufspannen. Anschließend wird auf die Inhalte der einzelnen Teile der Software-Architektur eingegangen.

5.2.1 Software-Architektur

Dieser Abschnitt erläutert, wie die Software-Architektur anhand der Analyseartefakte aufgestellt wird. Das Anwendungsfalldiagramm (engl. use case diagram) und die zugehörigen Anwendungsfälle (engl. use cases) werden nun zur Ableitung der benötigten Informationen genutzt. Abbildung 5.2 zeigt das aufgestellte Anwendungsfalldiagramm der CCSApp-Anwendung und die Ableitung der benötigten Information aus diesem Diagramm. Die verschiedenen Anwendungsfälle werden bei der Erstellung des Anwendungsfalldiagramms in ihre Capabilities einsortiert. Dadurch wird die zusammengehörende Funktionalität bereits in der Analysephase gruppiert.

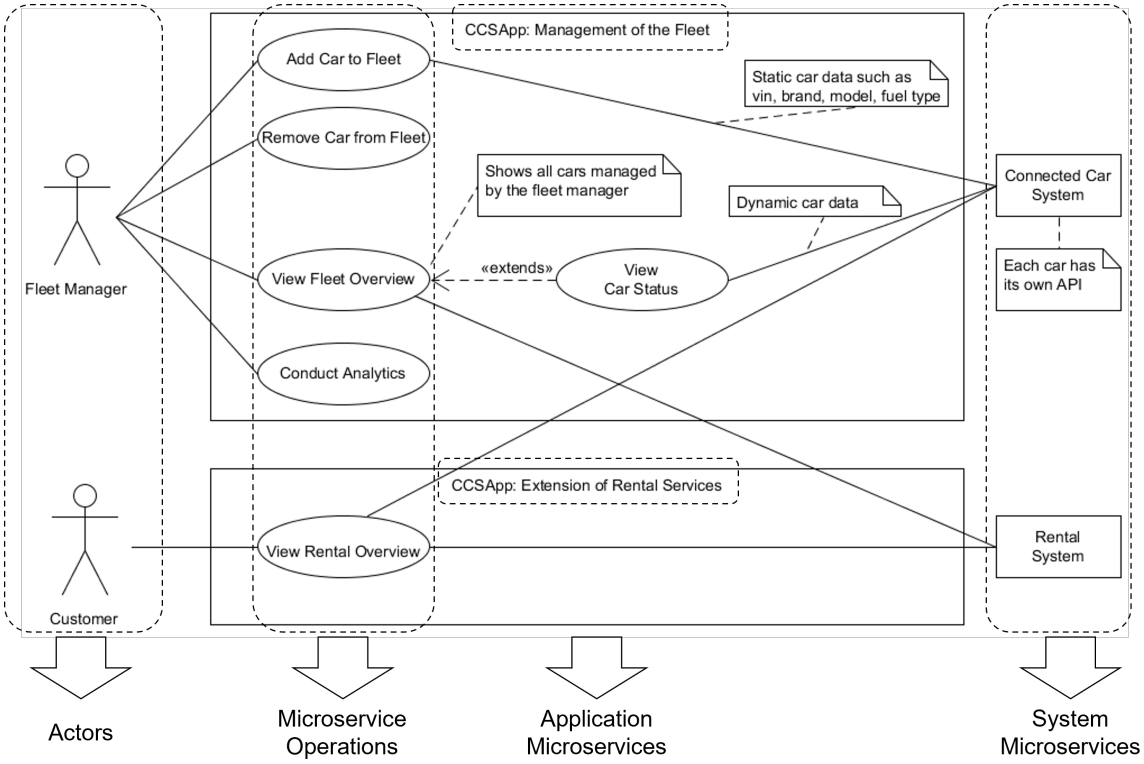


Abbildung 5.2: Anwendungsfalldiagramm und Ableitung der benötigten Informationen

Anhand dieser Gruppierung lassen sich im ersten Schritt die verschiedenen Anwendungs-Microservices aus den Anwendungsfällen ableiten. Im Beispiel der Betrachtung der Anwendungsfälle zur CCSApp ergeben sich insgesamt zwei Anwendungs-Microservices A-FleetManagement und A-RentalServiceExtensions. Aus der Analyse der externen Systeme, sowie der Analyse der anwendungsagnostischen Domäneninhalte lassen sich die involvierten externen Systeme, sowie die Domänen-Microservices ableiten. Anhand der spezifizierten Geschäftsabläufe wird deutlich, welche Informationsanforderungen benötigt werden. Dadurch lässt sich die Gesamtarchitektur der benötigten Softwarerachitektur aufstellen.

Abbildung 5.3 stellt die aus der Analyse abgeleitete resultierende Software-Architektur dar. Neben den oben beschriebenen Inhalten wurde noch eine Benutzerschnittstelle vorgesehen, die die Interaktion mit dem System ermöglicht. Für jede Benutzerschnittstelle wird in der Architekturbetrachtung genau ein Experience-Microservice vorgesehen, welcher die Schnittstellen der Anwendungs-Microservices für die entsprechende Benutzerschnittstelle aufbereitet. In der Literatur ist das vergleichbar mit dem sogenannten Backend-For-Frontend (BFF) [Ne15].

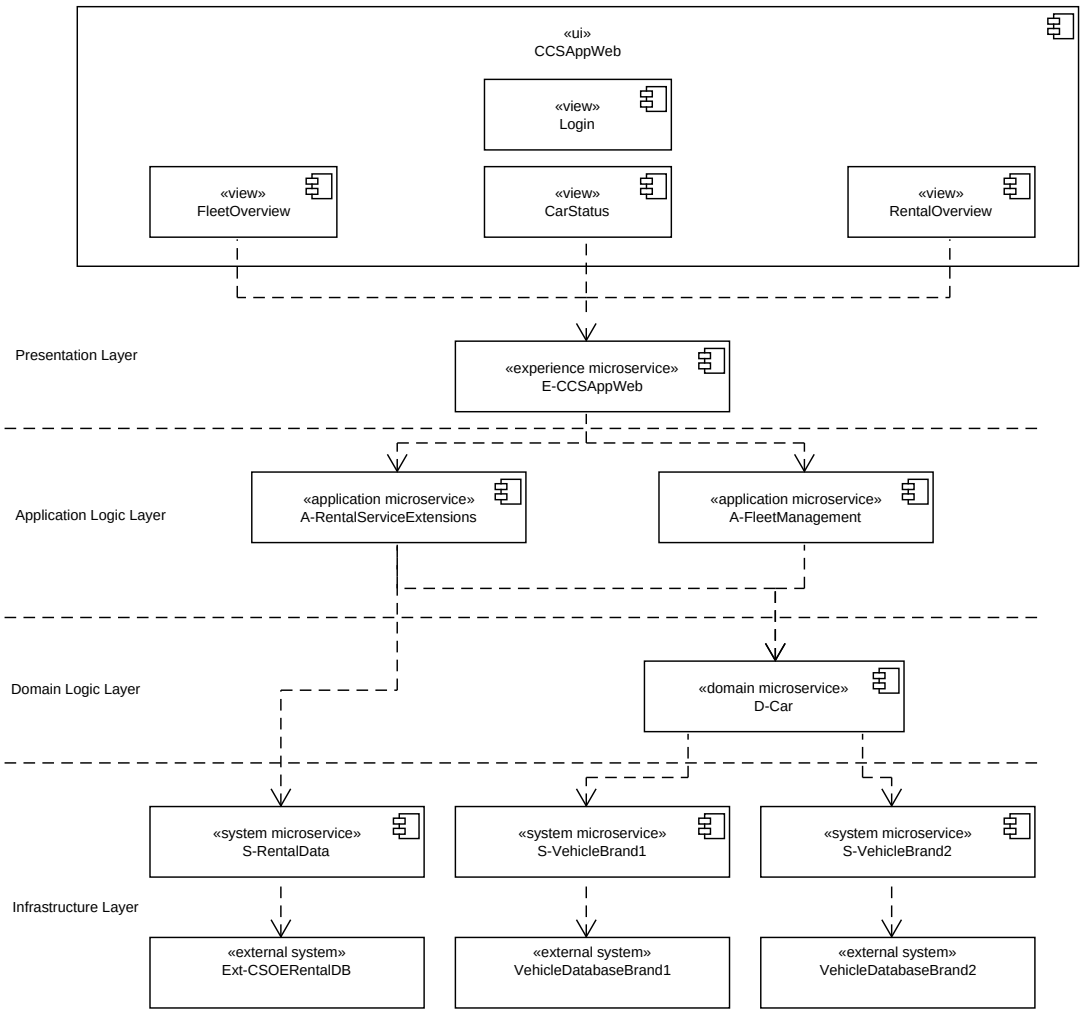


Abbildung 5.3: Software-Architektur der Anwendung CCSApp

5.2.2 System-Microservices und Domänen-Microservices

Das initiale Anwendungsfalldiagramm beinhaltet zunächst nur die abstrahierten externen Systeme. Diese werden bei der feineren Untersuchung durch die konkreten externen Systeme ersetzt. Für

jedes konkrete externe System lässt sich dann anschließend eine System Microservices ermitteln. Damit ein System Microservice erstellt werden kann, muss eine Aufbereitung des bzw. der externen Systeme (welche Daten liefert) erfolgen. Hierzu kann die beispielsweise anhand der Aufbereitung eines Datenmodells, der Schnittstelle, der Datenstruktur oder der Tabellenstruktur erfolgen. Aus der Analyse des System-Microservice wird ein objektorientiertes Modell, aus den bereitgestellten Schnittstellen entworfen. Dieser Reverse-Engineering-Aufwand ist erforderlich, um das externe System zu einer stabilen und RESTful API, d.h. eines System-Microservices, weiterzuentwickeln. Diese Abstraktion erlaubt es auch, mit inhärent komplexen und monolithischen externen Systemen besser umzugehen, da ein System-Microservice die Komplexität des externen Systems vor den Anwendungsentwicklern abstrahiert. Im Falle eines monolithischen externen Systems, wie z. B. ERP-Systeme, kann diese Modellierung zur Zerlegung in ein modulare Microservices unterstützen. Zwischen den konkreten externen Systemen der Automobilhersteller werden die System-Microservices erstellt, welche die Daten der externen Systeme als REST API bereitstellen.

Für die einfache Anbindung an ein externes System bieten können bereits etablierte Konnektoren verwendet werden. Hierbei liefern beispielsweise Integrationsplattformen, die als SaaS-Lösung angeboten verschiedene Konnektoren für die einfache Integration an. So lässt sich beispielsweise die Integrationsplattform MuleSoft einsetzen, um externe Systeme anzuschließen [SA+23].

Um den anfänglichen Aufwand für die Implementierung eines System-Microservice zu verringern, werden die erforderlichen Teile des System-API-Diagramms anhand der Informationsanforderungen extrahiert und in eine API-Spezifikation übersetzt. Diese Extraktion soll die Struktur des Modells möglichst nicht verändern, sondern lediglich die Endpunkte fokussieren, die von der Anwendung benötigt werden. Um die Abwärtskompatibilität der API zu gewährleisten, sollte die Extraktion nicht bis auf die Ebene der einzelnen Attribute der über die Endpunkte bedienten Ressourcen gehen. Wenn beispielsweise beschlossen wurde, einen Endpunkt einzubeziehen, sollte die Antwort alle Attribute der bedienten Ressource enthalten und nicht eine gefilterte Liste von Attributen auf der Grundlage der Informationsanforderungen.

Die System-Microservices werden iterativ für die Anwendungen entwickelt. Die benötigten und neue Endpunkte werden bei Bedarf auf der Grundlage der relevanten Teile des System-API-Diagramms hinzugefügt. Abbildung 5.4 zeigt das System-API-Diagramm, welches als Grundlage die Modellierung des externen Systems beinhaltet. In dem gegebenen Beispiel wird eine neue System-Entität erstellt, welche die darunterliegende Modellierung nutzt und Methoden bereitstellt, welche sich pragmatisch in API-Operationen überführen lassen. Dadurch wird die Ableitung der API-Spezifikation vereinfacht. Zusätzlich lässt sich das in Abschnitt 4.3.2 eingeführte Vorgehen zur Ableitung der Schnittstelle aus dem API-Diagramm in gleicher Weise für die Spezifikation der REST API wieder anwenden.

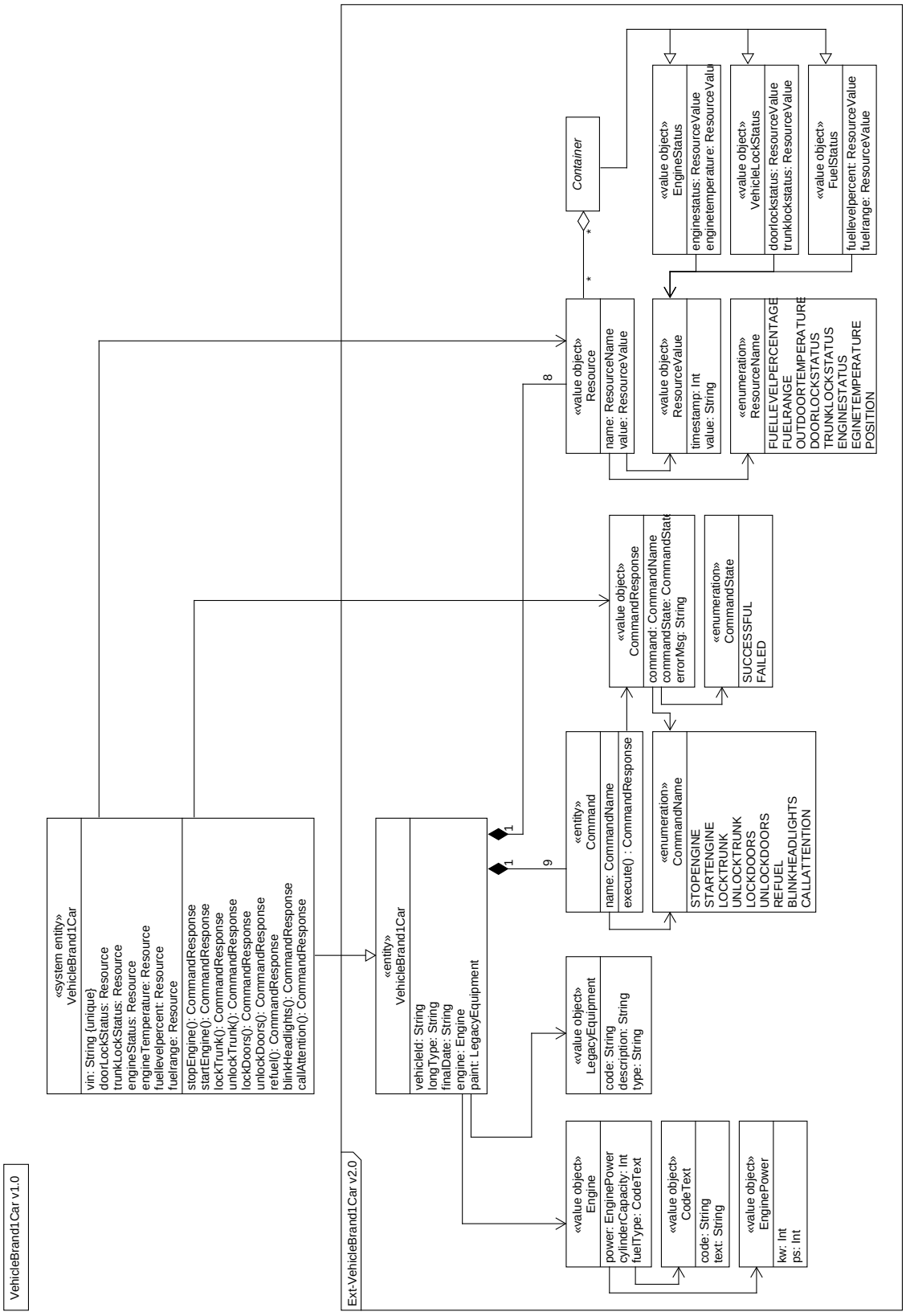


Abbildung 5.4: Abgeleitetes System-API-Diagramm zum externen System

Ein Domänen-Microservice bildet die anwendungsagnostische Funktionalität ab. Ein solcher Microservice kann ebenfalls als eine einheitliche Schnittstelle zu mehreren heterogenen Datenquellen angesehen werden. In der Regel bieten die externen Systeme zusätzliche Daten an oder Daten werden nicht angeboten, als die die in der Domäne modelliert werden. Bei solchen Fällen können die Anwendungs-Microservices weiterhin direkt die System-Microservices nutzen. Dennoch sollte ein übergeordneter Anwendungs-Microservice die System-Microservices nur dann verwenden, wenn die benötigten Daten nicht über die Domänen-Microservices aufgrund der oben genannten Problematik bereit gestellt werden können. Andernfalls könnten die Anwendungs-Microservices die Daten auf ähnliche, aber nicht gleichwertige Weise wie die Domänen-Microservices transformieren, was wiederum zu einer Heterogenität der Daten führt. Außerdem isoliert sich ein Anwendungs-Microservice durch die Verwendung der Domänen-Microservices vollständig von dem externen System. System-Microservices nutzen das Modell des externen Systems als Grundlage. Wenn sich das Modell des externen Systems ändert, kann sich gegebenenfalls auch die Schnittstelle der System-Microservices ändern. In diesem Fall müssen die Anwendungs-Microservices, welche die System-Microservices verwenden, ebenfalls angepasst werden. Die Anwendungs-Microservices, welche die Domänen-Microservices verwenden, müssen nicht angepasst werden. Stattdessen muss nur die Übersetzungslogik innerhalb des Domänen-Microservices angepasst werden.

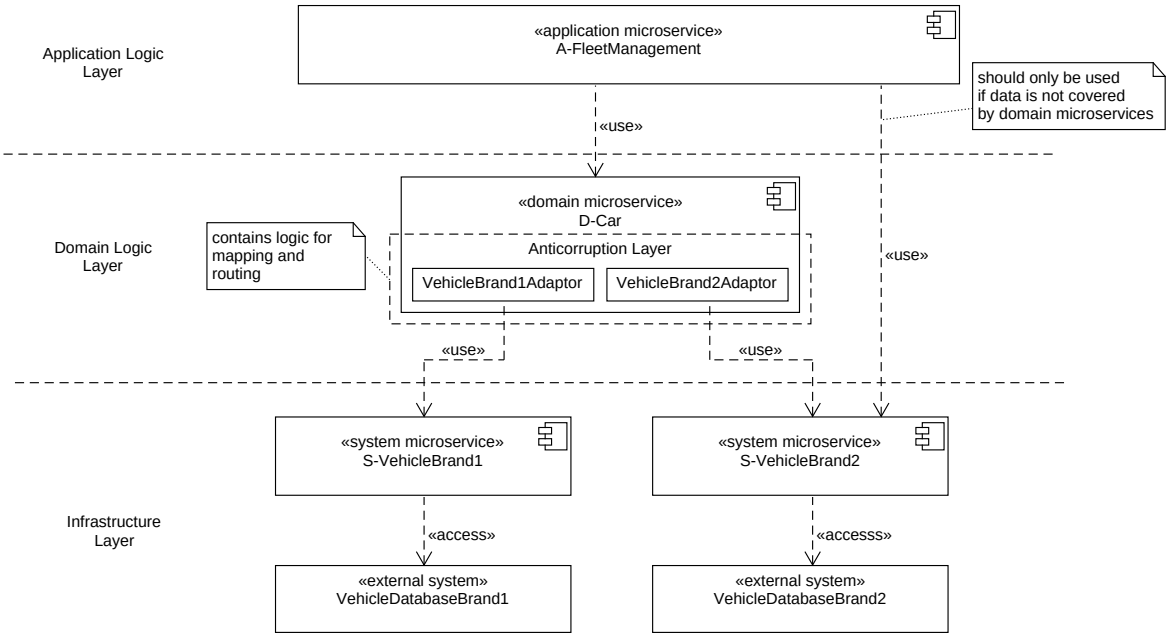


Abbildung 5.5: Integration der Domänen-Microservices

Ein wesentlicher Bestandteil der Übersetzung ist die Abbildung der Attribute, die in der Bounded Context Entity Relationship View (siehe Abschnitt 4.2.1) modellierten Attribute auf die in der Modellierung des externen Systems (ein sogenannter Foreign Boundend Context) herausgearbeiteten

Attribute. Die Abbildung der Daten kann anhand von Tabellen festgehalten werden, welche die konkrete Abbildung beschreiben. Eine solche Abbildungstabelle wird während der Analyse des externen Systems und dessen Modellierung aufgestellt. Da die Abbildungstabelle die Semantik der Attribute ausdrückt, kann sie verwendet werden, um die Attribute der Foreign Bounded Contexts zu identifizieren, die auf die Attribute des Bounded Contexts abgebildet werden können.

Wie in der in Abbildung 5.5 dargestellten Ausschnitt der Software-Architektur angedeutet, muss ein Domänen-Microservice eine Logik für Übersetzung und Weiterleitung enthalten. Zur Gestaltung der Übersetzung zwischen dem Domänenmodell, auf dem der entsprechende Domänen-Microservice zugrunde liegt, wird das DDD-Muster der Antikorruptionsschicht [Ev14] verwendet. Dadurch wird sichergestellt, dass bei Änderungen am externen Systems nur die Antikorruptionsschicht angepasst werden muss, nicht aber die Implementierung selbst.

5.2.3 Modellierung der Geschäftslogik anhand der Anwendungs-Microservices

Nachdem die System- und Domänen-Microservices modelliert sind, werden die Anwendungs-Microservices modelliert, indem die benötigten API-Diagramme erstellt werden. Dazu wird, analog zum Ansatz aus Abschnitt 4.3.2 die zugehörigen Microservice-Operationen abgeleitet. Für jeden Anwendungsfall aus dem Anwendungsfalldiagramm werden Endpunkte vorgesehen, die von der entsprechenden Microservice-Operation umgesetzt wird. Die Operationen werden entsprechend der Gruppierung im Anwendungsfalldiagramm zu dem entsprechenden Microservice vorgenommen. Das Anwendungs-API-Diagramm modelliert die auftretenden Entitäten analog zu dem in Abschnitt 4.2.2 vorgestellten Ansatz eines erweiterten UML-Klassendiagramms.

Im Folgenden wird beschrieben, wie eine systematische Ableitung eines Anwendungs-API-Diagramms unter der Berücksichtigung der Domänen- und System-Microservices stattfinden kann. Basierend auf der Software-Architektur werden die zu berücksichtigenden Domänen- und System-Modelle identifiziert. Die benötigten Inhalte ergeben sich aus dem spezifizierten Anwendungsfalldiagramm und werden in das Anwendungs-API-Diagramm übernommen. Noch nicht vorhandene, aber benötigte Entitäten (welche sich aus dem Anwendungsfall ergeben) werden als Anwendungsentitäten eingeführt. Anschließend werden die Relationen zwischen den Anwendungsentitäten zu den System- und Domänenentitäten ermittelt. Hierbei erbt in der Regel eine Anwendungsentität die Inhalte einer Domänenentität, währenddessen eine Systementität meistens durch eine Benutzen-Relation ausgedrückt wird. Sobald alle Entitäten und deren Beziehungen bestimmt sind, werden die benötigten Attribute und Methoden anhand der Anwendungsfalldiagramme ermittelt. In den Domänen- und Systementitäten werden nur die benötigten Attribute und Methoden aus den bestehenden Diagrammen übernommen. In den Prozessentitäten werden die anwendungsspezifischen Inhalte aufgenommen.

Jeder Anwendungsfall wird in eine Methode überführt. Beispielsweise wird aus dem Anwendungsfall "Add Car to Fleet" die Methode addCar() abgeleitet. Die Eingabeparameter der Methode werden von den Informationen abgeleitet, die der Hauptakteur an das System übergibt. Analog werden die Rückgabeparameter von den Informationen abgeleitet, die die Antwort des Systems an die primären Akteure darstellt. Abbildung 5.6 zeigt das resultierende API-Diagramm für den Anwendungs-Microservice A-FleetManagement.

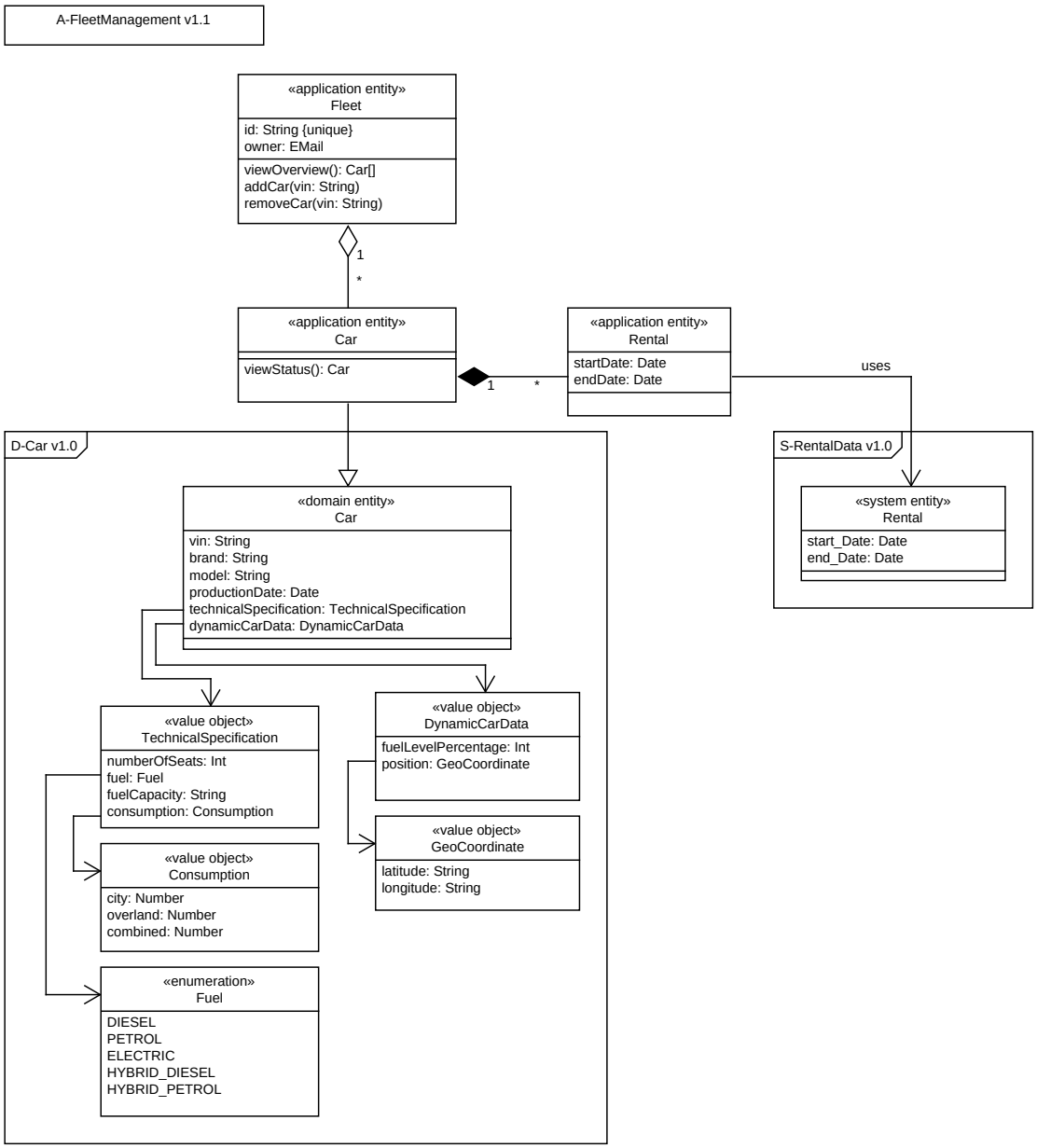


Abbildung 5.6: Anwendungs-API-Diagramm zum Anwendungs-Microservice A-FleetManagement

Da die Modelle eine wichtige Grundlage für die Ableitung der API darstellen, ist ein weiterer wichtiger Punkt die Versionierung der Modelle. Die Versionsnummer des Modells sollte hierbei immer der Versionierung der zugehörigen API entsprechen. Entsprechend einer API-Versionierung [De17] wird das Modell mit einer Versionsnummer versehen. Zusätzlich werden die Versionsnummern der eingebundenen Domänen- und System-API-Diagramme angegeben. Dadurch wird nicht nur die Version des aktuellen Modells geklärt, sondern auch die Version der Abhängigkeiten werden festgehalten. Wird eine kleine Änderung (engl. minor change) vorgenommen, dann wird die hintere Ziffer der Version erhöht. Geben sich strukturelle Änderungen, dann führt dies zu einer Änderung der vorderen Ziffer (engl. major change). Entsprechend der API-Versionierung wird dadurch erkenntlich gemacht, wenn das Modell gravierende Änderungen (engl. breaking changes) erfahren hat.

Gemäß dem in Abschnitt 4.3.2 beschriebenen Vorgehens wird nun die Schnittstelle des Anwendungs-Microservices abgeleitet. Listing 5.2 zeigt die resultierende API-Spezifikation, welche sich anhand des API-Diagramms (Abbildung 5.6) und des Vorgehens resultiert. Zeile 7 und 17 erhalten hier beispielsweise die spezifizierten Datentypen, die bei einer Anfrage und einer Antwort enthalten sind.

```
1  paths:
2   /fleets/{id}/cars/{vin}
3      uriParameters:
4          id:
5            description: Fleet ID
6            body:
7              type: AddCarToFleetRequest
8          vin:
9            description: Vehicle Identifier
10           type: string
11       post:
12         description: Adds the car with the given VIN to the fleet
13         responses:
14           200:
15             description: Car with the given VIN has been successfully
                   added to the fleet with Fleet ID
16             body:
17               type: AddCarToFleetResponse
18           400: ...
```

Listing 5.2: Abgleiteter API-Endpunkt zum Anwendungsfalldiagramm

5.2.4 Orchestrierung der Geschäftslogik

Anhand der Orchestrierungs-Diagramme wird die Dynamik der Geschäftslogik erfasst und die umzusetzende Geschäftslogik spezifiziert. Die Orchestrierungs-Diagramme werden anhand der spezifizierten API-Diagramme zu den Anwendungs-Microservices und der von den Anwendungsfällen geforderten Geschäftslogik abgeleitet. Für jede Microservice-Operation eines Anwendungs-Microservices wird ein Orchestrierungs-Diagramm modelliert. Der Startpunkt des Diagramms besteht aus der Operation, welche von einem Client angestoßen wird. Die Operation entspricht hierbei einer Aktion des Benutzers, welche auch die Eingabe und die Ausgabe der Operation definiert.

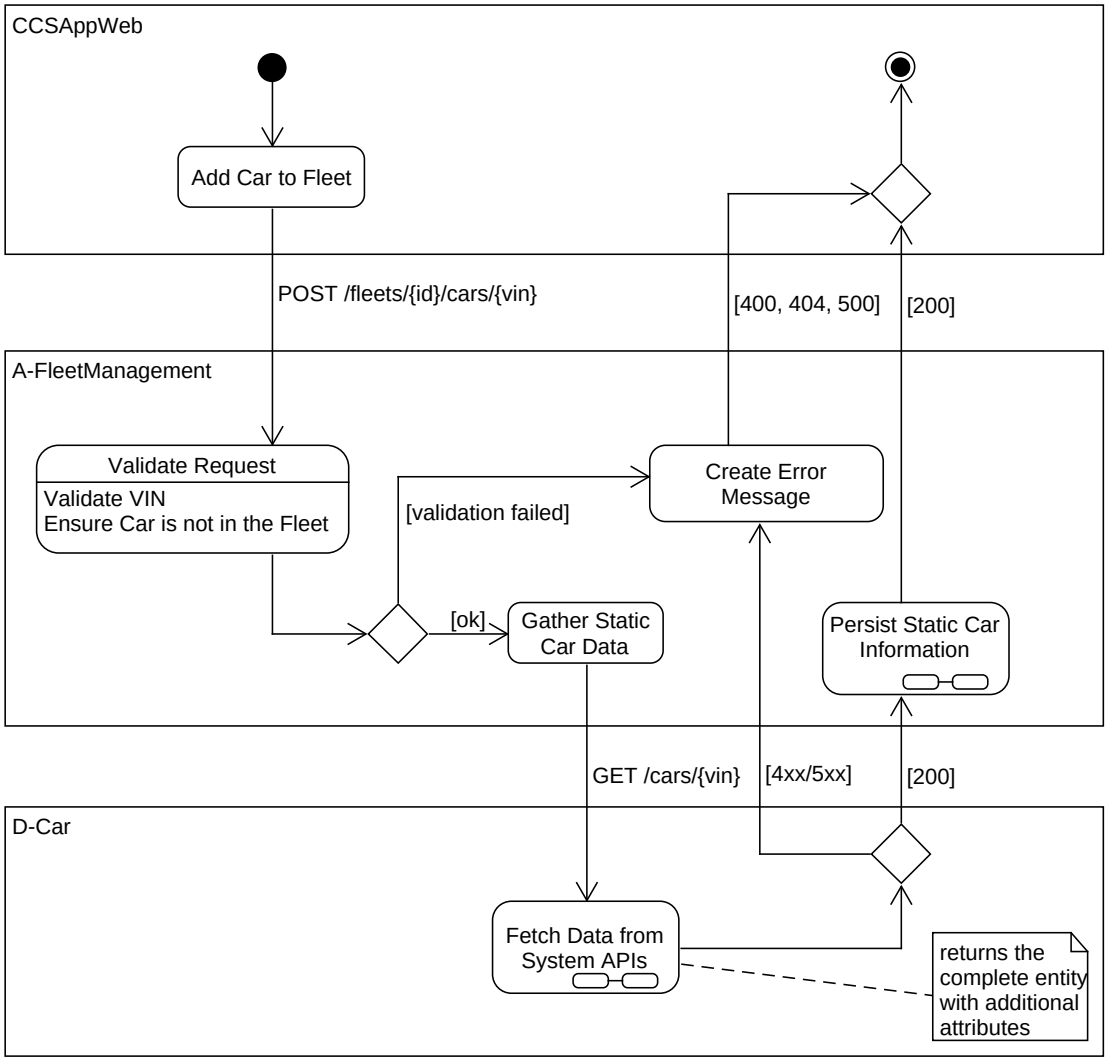


Abbildung 5.7: Orchestrierung der Anwendung mittels des Orchestrierungs-Diagramms der Microservice-Operation "Add Car to Fleet"

Im mittleren Teil des betrachteten Anwendungs-Microservices wird die umzusetzende Geschäftslogik und die notwendige Orchestrierung der darunterliegenden Microservices spezifiziert. Die Informationsanforderungen liefern hierbei die Information der darunterliegenden, zu orchestrierenden Systeme. Bei den Aufrufen der benötigten Funktionalität wird zudem der entsprechende Endpunkt angegeben. Die Ableitung der Inhalte erfolgt hierbei vom Anwendungsfalldiagramm. Auf der unteren Ebene werden die orchestrierten Microservices dargestellt. Bei jedem Orchestrierungs-Diagramm werden zusätzlich die Antworten angegeben. Das inkludiert die zurückgegebenen Statuscodes und Antworten.

In Abbildung 5.7 ist für den Microservice A-FleetManagement und die Microservice-Operation addCarToFleet das Orchestrierungs-Diagramm angegeben. Dieses modelliert hier den spezifizierten Anwendungsfall "Add Car to Fleet", welcher vom Benutzer angestoßen wird. Bei der betrachteten Orchestrierung wird der Benutzer nach einer Eingabe, der VIN des Fahrzeugs, gefragt. Anhand der verschiedenen Aktivitäten innerhalb des Orchestrierungs-Diagrammes werden die Eingaben validiert und weiter verarbeitet. Beispielsweise wird bei der ersten Aktivität des Orchestrierungs-Diagramms die VIN überprüft, welche der benötigte Eingabeparameter der Microservice-Operation addCarToFleet darstellt.

Neben der Validierung der Eingaben wird die notwendige Orchestrierung zu weiteren Microservices festgelegt. Im betrachteten Beispiel wird der Domänen-Microservices D-Car angesprochen. Die nicht weiter aufgeführte Orchestrierung (erkennbar an dem Verkettungs-Stereotyp) des Microservices D-Car fordert die notwendigen Daten von den darunterliegenden System-APIs der Fahrzeughersteller an. Wenn der Domänen-Microservice in der Lage ist, die Daten von den entsprechenden System-Microservices anzufordern, gibt dieser die für das Fahrzeug verfügbaren Daten in einem herstellerunabhängigen Datenformat zurück. Falls nicht, wird ein Fehler zurückgegeben, der anhand der Schnittstelle mit einem entsprechenden Statuscode zurückgegeben wird.

5.2.5 Benutzerschnittstelle und Experience-Microservices

Die Benutzerschnittstelle sendet die Anfragen über die Experience-Microservices. Für die Experience-Microservices wird der Ansatz des klassischen Backend-for-Frontends (BFF) [Ne15] verfolgt. Pro Benutzerschnittstelle (Webanwendung, mobile Anwendung) wird genau ein Experience Mircroservice vorgesehen. Ein Experience-Microservices ist für die Aufbereitung der von den Anwendungs-Microservices empfangenen Daten in einem für die Benutzerschnittstelle geeigneten Format zuständig. Die Syntax und Semantik des Formats hängt hierbei von der Benutzerschnittstelle selbst ab. Ein Beispiel hierfür ist die Paginierung im Bereich einer mobilen Benutzerschnittstelle, um eine Limitierung der übermittelten Daten zu erreichen.

Im Sinne der Literatur und technischen Möglichkeit bieten sich hier weitere Optionen. Beispielsweise kann die Benutzerschnittstelle auch mit GraphQL angebunden werden [HP18].

5.3 Überführung in die Implementierung

Nachdem die benötigten Artefakte modelliert sind, findet die Überführung in die Implementierung statt. Hierbei werden die einzelnen Aspekte gemäß der in Kapitel 4 vorgestellten Mikroarchitektur überführt.

5.3.1 Mikroarchitektur

Jeder Microservice lässt sich mit einer unterschiedlichen Technologie umsetzten. Dadurch kann eine heterogene Vielfalt an Technologien eingesetzt werden [Ne15, KJ+15]. So kann die Logik des Microservices A-FleetManagement beispielsweise mit der Programmiersprache Go [Ja17] implementiert werden.

```
1
2  v src/main
3    v api
4      openapi.yaml   #API specification
5      > controller
6        FleetService
7        CarService
8      > mappers
9        APItoLogicMapper
10       LogicToAPIMapper
11   v logic
12      > model
13        FleetRepositoryInterface.go
14        Fleet.go
15        Rental.go
16        ...
17      > operations
18        FleetOperationsInterface.go
19        ViewOverviewOperation.go
20        ViewStatusOperation.go
21        ...
22    v infrastructure
23      > database
24        > mappers
25        > persistententities
26      > external
27        > DCar
```

Listing 5.3: Mikroarchitektur des AM-Microservices A-FleetManagement

Die in Abschnitt 4.4 eingeführte Mikroarchitektur lässt sich für diese Programmiersprache nutzen. Listing 5.3 zeigt die übergeordnete Struktur der Go-Implementierung und der Unterteilung in die drei wichtigen Bestandteile API, Logik und Infrastruktur.

Die Umsetzung eines Domänen-Microservices erfolgt analog. In der Regel sollte ein Domänen-Microservice aber keine zusätzlichen Anbindungen an andere Domänen-Microservices implementieren.

5.3.2 Überführung des Entwurfs in die Implementierung

Die Empfehlung bei der Implementierung ist, dass zu Beginn die Entitäten in die entsprechenden Konstrukte der verwendeten Programmiersprache umgesetzt werden. Im Beispiel von Java sind dies Klassen, bei der Programmiersprache Go werden hingegen sogenannte "structs" verwendet. Konkret wird für jede Entität und jedes Wertobjekt eine entsprechende Klasse im Logik-Ordner angelegt (wie bspw. beim Anwendungs-Microservice A-FleetManagement Fleet und Rental als Go structs). Die Ableitung der konkreten Objekte für das Modell erfolgt aus dem API-Diagramm. Durch ein zusätzliches Repository-Interface (FleetRepositoryInterface) kann eine Kapslung der Implementierung der Persistenz der Entitäten erfolgen. Hierbei sind insbesondere Methoden zur Persistenz der Entitäten (save(), find(), findByVin()) vertreten. Dies ist immer dann notwendig, wenn die entsprechende Entität persistiert werden soll. Anschließend werden im Order "operations" die im API-Diagramm modellierten Methoden innerhalb einer "Interface-Klasse" beschrieben. Wenn die Operation auf Persistenzmechanismen zugreifen müssen, erfolgt dies über die in dem Modell beschriebene Persistenzschnittstelle. Für die konkrete Implementierung der Logik ist neben den Anwendungsfällen noch das Orchestrierungsdiagramm hilfreich, da dieses die dynamischen Aspekte der Anwendungsfalldiagramme festhält.

Nach der Implementierung der Logik erfolgt die Implementierung der API. Hierbei wird der benötigte API-Controller (wie in Listing 5.3 exemplarisch gezeigt) im entsprechenden API-Ordner implementiert. Für die Implementierung des Controllers wird die zuvor erstellte API-Spezifikation (siehe Abschnitt 4.3.2) verwendet, indem für jeden Endpunkt in der OpenAPI-Spezifikation eine entsprechende Methode erstellt wird. Hierzu können auch Werkzeuge (wie oapi-codegen [OAP-Ope] für Go) verwendet werden, welche die API-Spezifikation gemäß der OpenAPI-Spezifikation [OAI-Wha] in Implementierungsschnittstellen überführt, die noch im Controller implementiert werden müssen.

Im Bereich der Infrastruktur wird die Datenbankanbindung implementiert. Hierfür werden die sogenannten Persistenz-Entitäten (engl. persistency entities) benötigt, da dass Modell lose von der Anbindung der Datenbank gekoppelt sein soll. Dies ist insbesondere notwendig, da die Datenbank-Technologie ausgetauscht werden kann. Daher werden noch zusätzlich Mapper benötigt, welche die

Überführung der Persistenz-Entitäten in die Modell-Entitäten (sowie in die Rückrichtung) vornehmen.

Neben der Datenbank sind insbesondere auch die Anbindung der benötigten (Domänen-)Microservices für die Orchestrierung relevant. Die notwendigen Aufrufe an diese werden ebenfalls im Infrastruktur-Teil implementiert. Hierbei wird im Ordner "infrastructure/external" für jeden benötigten Microservice die entsprechenden Anfragen umgesetzt.

5.4 Zusammenfassung

Das Ziel dieses Kapitels war es, den Entwurf des systematischen Ansatzes anhand des Beispiels der ConnectedCarServicesApplication (CCSApp) weiter zu schärfen. Hierbei wurde mit der Anforderungsanalyse begonnen und für die verschiedenen Ebenen der Zielarchitektur (vgl. Abschnitt 4.1) die Erhebung der Anforderungen präzise geklärt.

Nach der Spezifikation der Anforderungen wurde die systematische, strukturerhaltende Ableitung der Software-Architektur basierend auf den bisher erstellten Artefakten aufgezeigt. Hierzu wurden die Analyseartefakte in die Anwendungsarchitektur überführt. Anhand der beiden Entwicklungsprozesse wurden die Domänen- und Anwendungs-Microservices unabhängig voneinander aufgestellt. Für den Domänen-Microservice Car wurde zu Beginn die Fachlichkeit von konkreten System-Microservices eines Fahrzeugherstellers modelliert. Basierend auf der Fachlichkeit der Domäne ConnectedCar wurde das Domänenwissen extrahiert und Domänen-Microservices eingeführt. Die eingeführten Domänen-Microservices kapseln hierbei die heterogenen Daten der verschiedenen Fahrzeughersteller und liefern für die darüber liegenden Anwendungs-Microservices eine einheitliche Schnittstelle.

Die Anwendungs-Microservices greifen bei der Modellierung die Modelle der Domänen-Microservices auf und verwenden diese als Grundlage. Dadurch ergibt sich bei der Aufstellung des API-Diagramms der Bezug zu den Orchestrierungs-Diagrammen und den benötigten Microservice-Operationen. Zum Abschluss des Kapitels wurde gezeigt, wie die einzelnen Artefakte des Anwendungs-Microservices (und hierzu analog des Domänen-Microservices) in die Mikroarchitektur überführt werden. Dabei wurde die Ableitung entsprechend der Vorgehensweise im Abschnitt 4.4 durchgeführt und weiter diskutiert.

6 Einbezug von Querschnittsdomänen am Beispiel IoT

Das Domänenmodell für das Internet der Dinge wird systematisch gemäß dem in Abschnitt 4.2 eingeführten Domänenentwicklungsprozess erarbeitet. Hierbei steht zu Beginn in Abschnitt 6.1 die Erfassung des Domänenwissens im Vordergrund. Durch die Analyse verschiedener IoT-Quellen (Abschnitt 6.1.1 wird in Abschnitt 6.1.2 ein konzeptionelles Modell aufgestellt, welches zur Überführung der IoT-Domäneninhalte in eine Context Map dient. Diese Überführung wird in Abschnitt 6.1.3 beschrieben. In Abschnitt 6.1.4 werden im nächsten Schritt die Inhalte der Domäne taktisch modelliert und durch die genauere Spezifikation der Bounded Contexts erläutert. Das weitere Domänenwissen wird anhand von Constraints erfasst, wodurch dieses weiter geschärft und mittels der Object Constraint Language (OCL) formalisiert wird. Durch die Constraints wird auch die Einhaltung der Fachlichkeit sichergestellt, was auch die Ableitung von Tests ermöglicht. Anschließend erfolgt in Abschnitt 6.2 die Verknüpfung der Domäne zur Anwendungsentwicklung. Durch die Separierung der Querschnittsdomäne wird der zweite Teil der Anforderung A4 zur Separierung der Domänenlogik von der Anwendungslogik aufgegriffen.

6.1 Domänenmodell für das Internet der Dinge

Um ein Domänenmodell für das Internet der Dinge (Internet of Things, IoT) aufzustellen, ist es essentiell, das Domänenwissen der IoT-Domäne zu verstehen und zu erfassen. Auf der einen Seite benötigen Anwendungen verschiedene IoT-Funktionalitäten, damit diese ihre Aufgaben erfüllen können und auf der anderen Seite existieren bereits Standards und Rahmenwerke, welche die IoT-spezifischen Gegebenheiten vereinheitlichen.

Die IoT-basierten Anwendungen sind ebenfalls von einer zentralen Bedeutung. Es reicht nicht aus, nur die IoT-Quellen zu berücksichtigen. Nur wenn bekannt ist, welche IoT-Anforderungen eine Anwendung hat, kann das entsprechende Domänenwissen passend formuliert werden. Daher müssen auch die Anforderungen, die eine IoT-Anwendung mit sich bringt, berücksichtigt werden. Die redundanten Bausteine (welche verschiedene Anwendungen benötigen) sind hierbei herauszuziehen und die IoT-Domäne zu integrieren. Daher muss die Microservice-Architektur der Domäne IoT auch auf die Anforderungen der Anwendung eingehen.

Daher ist es für die Entwicklung von Anwendungen zu Beginn wichtig, dass in der Analysephase die IoT-Anforderungen einer Anwendung festgehalten werden. Diese Anforderungen sind eine wichtige

Grundlage für die Aufstellung des Domänenwissens, da die Anwendungen Anforderungen an die IoT-Domäne stellen. Im Kontext des Internets der Dinge spielen hier Sensoren und Sensordaten eine elementare Rolle. Benötigt eine Anwendung Sensordaten oder sollen Aktuatoren angesteuert werden, dann ist IoT ein wichtiger Bestandteil dieser Anwendung.

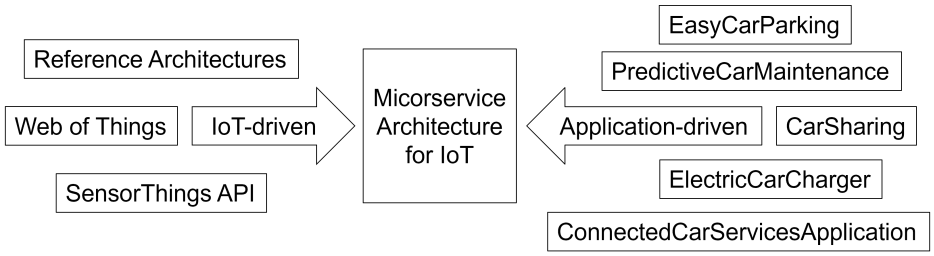


Abbildung 6.1: Weg zur Etablierung der Microservice-Architektur für IoT

Abbildung 6.1 verdeutlicht die verschiedenen Gesichtspunkte, die zur Etablierung herangezogen wurden. Auf der einen Seiten liefern die verschiedenen IoT-Quellen einen Beitrag. Des Weiteren werden von der anderen Seite auch die Anforderungen der Anwendungen betrachtet, um das Domänenmodell für IoT zu festigen. Im ersten Schritt wird die Context Map für das Internet der Dinge aufgestellt. Hierbei werden initial die bestehenden IoT-Quellen analysiert, die Konzepte des IoT abgeleitet und die benötigten Bounded Contexts etabliert. Für die Aufstellung der Bounded Contexts werden auch die Anforderungen von IoT-basierten Anwendungen berücksichtigt, da die Querschnittsdomäne diese berücksichtigen sollte.

6.1.1 Analyse der bestehenden Quellen

Als Quellen bilden die Standards und Rahmenwerke eine gute Grundlage für die IoT Context Map. Für die Domäne IoT liefert das IoT-Referenzmodell der Standardisierungsorganisation ITU [ITU-T12] nützliche Hinweise für die Extraktion der Subdomänen und der Bounded Contexts. Eine weitere geeignete Quelle ist [Fr15], ein White Paper von WSO2 (ein Anbieter von Open-Source-Technologie), in dem eine Referenzarchitektur für das Internet der Dinge vorgestellt und diskutiert wird, welche sich neben der Aufbereitung und Speicherung von IoT-Daten mit zusätzlichen IoT-Aufgaben befasst. Ein prominenter Standard, der sich mit dem Thema Sensing beschäftigt, ist die Sensor-Things API (STA). Diese liefert ein Datenmodell und eine Schnittstelle, um Dinge und Sensoren über das das Web bereitzustellen [OGC-STA-Sen]. Anhand der verschiedenen Quellen lässt sich ein konzeptionelles IoT-Modell aufstellen.

6.1.2 Aufstellung eines konzeptionellen IoT-Modells

Beim IoT stehen bei den verschiedenen Quellen die Dinge im Vordergrund, die verschiedene Fähigkeiten mit sich bringen. Ein wichtiger Bestandteil dinf hierbei physische Geräte. Ein solches Gerät hat mehrere Fähigkeiten (engl. abilities). Darunter fällt die Kommunikation, sowie die sensorischen (engl. sensoring) und die und ausführenden (eng. tasking) Fähigkeiten. Auch bei dem Standard der SensorThings API wird zwischen Sensing [OGC-STA-Sen] und Tasking [OGC-STA-Tas] unterschieden. Zudem benötigt jedes IoT-Gerät die Fähigkeit, kommunizieren zu können [ITU-T12].

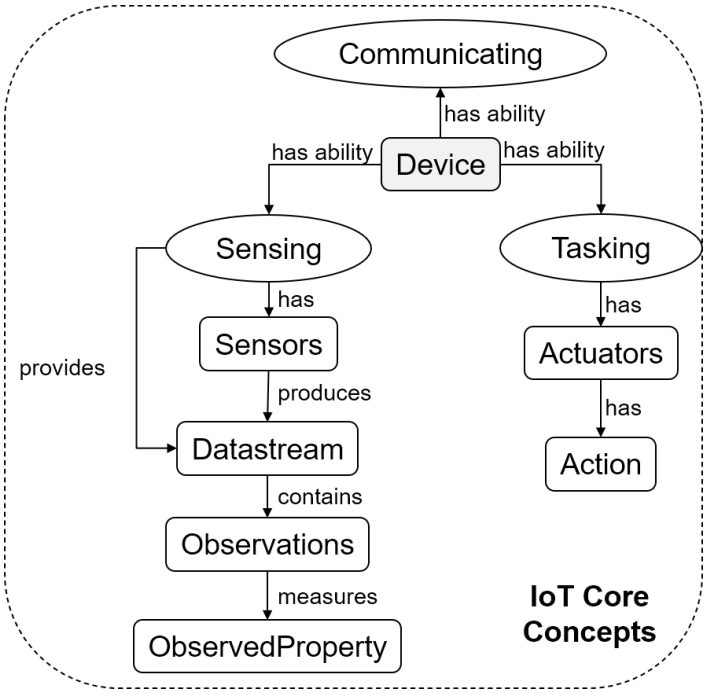


Abbildung 6.2: Konzeptionelles IoT-Modell für die Aufstellung der Domäne IoT

Abbildung 6.2 zeigt den Zusammenhang der Fähigkeiten eines solchen Gerätes und die zugehörigen Konzepte. Die Fähigkeiten Sensing und Tasking stehen bei IoT-Anwendungen im Vordergrund. Sensing wird benötigt, um die Informationen bereitzustellen, die eine Anwendung benötigt (z. B. die Position eines Gerätes). Damit ein Gerät die verschiedenen Fähigkeiten erfüllt, benötigt dieses die entsprechende Ausstattung. Dazu werden Sensoren benötigt, die die Beobachtungen in Form von Datenströmen erzeugen. Sensoren produzieren in der Regel durchgängig Beobachtungen. Eine Beobachtung ist einer bestimmten Eigenschaft zugeordnet, die gemessen wird. Die Beobachtung besteht aus einen Typ (das, was gemessen wird) und einem Ergebnis (den Wert, der gemessen wird) und werden in einem Datenstrom (engl. datastream) gruppiert. Ein Datenstrom enthält somit alle Beobachtungen zu einem bestimmten Sensor und einer beobachteten Eigenschaft (engl. ObservedProperty). Ein Gerät ist hierbei nicht nur auf einen Sensor beschränkt, sondern kann auch mehrere

Sensoren enthalten, die unterschiedliche Eigenschaften beobachten [OGC-STA-Sen].

Die Tasking-Fähigkeit betrifft die Aktoren. Ein Tasking-Gerät hat also mindestens einen Aktor, der dafür zuständig ist, die Umgebung in irgendeiner Weise zu manipulieren. Die Fähigkeit Tasking ist auch für IoT-basierte Anwendungen relevant, da die Anwendungen Aktoren steuern müssen (z. B. das Ein- und Ausschalten eines Lichtes). Aktoren sind die Hauptkomponenten der Tasking-Fähigkeit, wie auch im Standard der SensorThings API zum Tasking-Teil [OGC-STA-Tas]. Sie sind für die Manipulation der Umgebung durch Aktionen zuständig. Weiterhin definiert der Standard Web of Things (WoT) [W3C-WoT] beispielsweise Schnittstellen, die die Steuerung der Aktionen ermöglichen.

Zur Kommunikation gehört, wie die IoT-Netzwerke mit unterschiedlichen Topologien verbunden sind. Außerdem werden unterschiedliche Protokolle und Standards für die Kommunikation verwendet. Für die Netzwerke selbst ist das wichtig, aber im Rahmen der betrachteten IoT-basierten Anwendungen, welche die IoT-Daten nutzen oder Aktoren manipulieren, spielt dieser Punkt nur eine untergeordnete Rolle und wird daher nicht weiter ausgeführt.

6.1.3 Ableitung der Inhalte für die Context Map

Anhand des aufgestellten konzeptionellen Modells (siehe Abbildung 6.2) lassen sich nun die IoT-Funktionalitäten ableiten und eine Gruppierung der Inhalte durchführen. Hierbei lässt sich eine Gruppierung in zwei Subdomänen vornehmen. Die erste Subdomäne behandelt die verschiedenen Fähigkeiten der IoT-Geräte, während die zweite Subgruppe sich um verarbeitende Angelegenheiten kümmert.

Die erste Subdomäne IoTDevice beinhaltet die Sensing- und Tasking-Fähigkeiten der Geräte. Die Trennung von Tasking und Sensing erfolgte aufgrund der unterschiedlichen Aufgaben, die die Bounded Contexts zu erfüllen haben. Für Sensing ist vor allem die Bereitstellung der gemessenen Beobachtungen wichtig, welche durch den Bounded Context SensingDevice bereitgestellt werden sollen. Daher bietet der Bounded Context SensingDevice die Funktionalität zum Speichern, Abrufen und Verwalten von Beobachtungen von Sensoren.

Für die Tasking-Fähigkeiten adressiert der entsprechende Bounded Context TaskingDevice die Aktoren und stellt hierfür die entsprechenden Schnittstellen bereit. Ein wichtiger Bestandteil der Bounded Contexts ist somit die Bereitstellung von REST-APIs, die zur Erfüllung der gewünschten IoT-Funktionalität notwendig sind. Zum Beispiel müssen Beobachtungen von Geräten mit Sensing-Fähigkeit über eine GET-Operation abgerufen sowie über eine POST-Operation gespeichert werden. Außerdem könnte es auch notwendig sein, diese APIs für das Streaming der Sensordaten zu erweitern. Weiterhin wäre denkbar, einen Bounded Context für die Communicating-Fähigkeit zu etablieren. Dieser wurde jedoch

nicht betrachtet, da aufgrund der betrachteten Anwendungen kein Bedarf eines solchen Bounded Context besteht.

Unterstützende IoT-Funktionalitäten, welche die IoT-Daten verarbeiteten, wird von der zweiten Subdomäne IoTProcessing bereitgestellt. Der Bounded Context Analytics ermöglicht die Kombination von IoT-Daten. Beispielsweise können unstrukturierte Daten in Zeitreihen umgewandelt werden, die für weiterführende Analysen geeignet oder notwendig sind. Die Umwandlung von unterschiedlichen Eingabeformaten in gängige Standards wird durch den Bounded Context Format ermöglicht. Der Bounded Context Report behandelt die Erstellung von Berichten, welche die IoT-Funktionen betreffen. Dies könnte beispielsweise ein Bericht über aufgetretene Fehler von den IoT-Geräten sein. Funktionalitäten zur Protokollierung wichtiger Ereignisse wird vom Bounded Context Log geboten.

Formalisierung der Context Map mit UML

Die abgeleiteten Subdomänen und Bounded Contexts werden im nächsten Schritt mittels dem erweiterten UML-Profils formalisiert und in eine Context Map überführt. Abbildung 6.3 zeigt die resultierende Context Map mit den zugehörigen Subdomänen und Bounded Contexts. Im unteren Bereich ist die Subdomäne IoTDevice angeordnet, welche die Bounded Contexts SensingDevice und TaskingDevice beinhaltet. Im nächsten Schritt werden die Inhalte der Bounded Contexts modelliert. Hierbei wird auch nach Bedarf vorgegangen. Das bedeutet, sobald eine Anwendung den Inhalt eines Bounded Contexts benötigt, wird dieser modelliert.

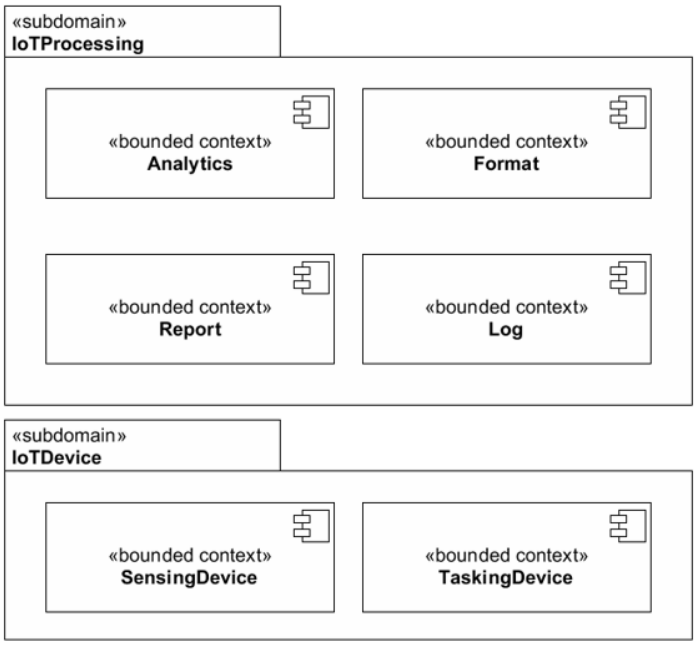


Abbildung 6.3: IoT Context Map

6.1.4 Modellierung der Bounded Contexts der IoT-Domäne

Nachdem die initiale Context Map aufgestellt ist, steht die Modellierung der konkreten Domäneninhalte an. Hierzu werden die Bounded Context Entity Relation Views der einzelnen Bounded Contexts aufgestellt. Am Beispiel des Bounded Context SensingDevice wird das Vorgehen erläutert, da dieser als zentraler Einstiegspunkt für IoT-basierte Anwendungen gesehen werden kann. Grundlegend ist die aufgestellte Entity Relation View an der SensorThings API (STA) [OGC-STA-Sen] orientiert und liefert ein vereinfachtes Modell. Der Name SensingDevice wurde für den Bounded Context gewählt, da die Sensing-Fähigkeit des Gerätes die notwendige Funktionalität bereitstellt. Ein wichtiger Punkt des Modells ist, dass dieses von vielen verschiedenen Anwendungen wieder verwendet werden kann. Für das Modell ist wichtig, dass die verschiedenen Domänenobjekte erfasst und in Relation gesetzt werden. Die auftretenden Begrifflichkeiten werden oder sind Teil der ubiquitären Sprache.

Abbildung 6.4 zeigt die resultierende Entity Relation View des Bounded Context SensingDevice. Dieser Bounded Context liefert für IoT-basierte Anwendungen die benötigten IoT-Daten und ermöglicht es, Sensordaten zu speichern und zu verwalten. Die wichtigsten Entitäten sind die Observation und der Datastream. Beide Entitäten sind Shared Entities, was bedeutet, dass diese Entitäten nach außen zur Verfügung gestellt werden.

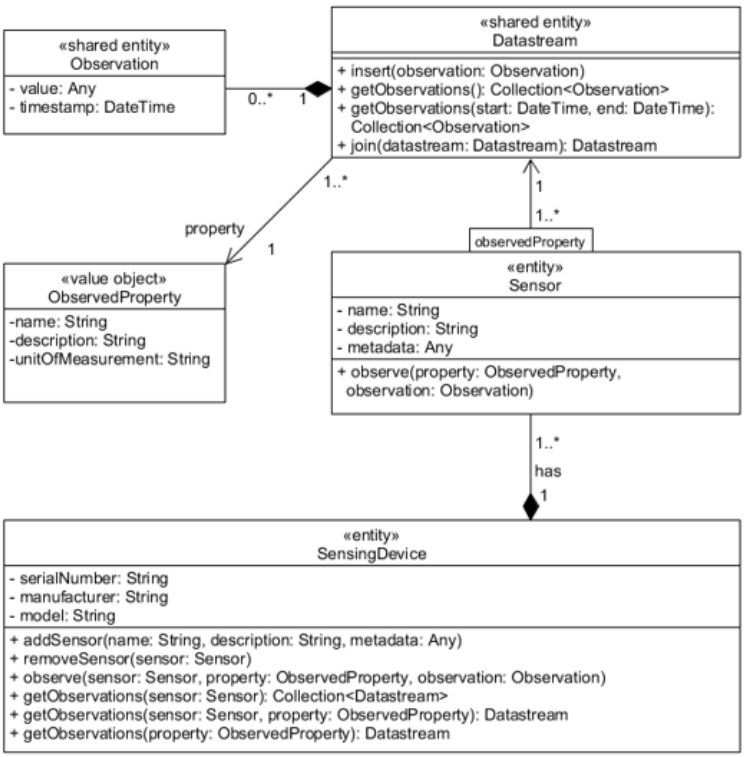


Abbildung 6.4: Entity Relation View des Bounded Context SensingDevice

Die Entität SensingDevice kann als Aggregat für verschiedene Sensoren gesehen werden und ermöglicht es, weitere Sensoren hinzuzufügen oder vorhandene Sensoren mittels der Methoden addSensor(...) und removeSensor() zu entfernen. Die Entität Sensor repräsentiert physikalische Sensoren, die an einem SensingDevice angebracht sind und können mittels der Methode observe(...) Beobachtungen durchzuführen. Zu einem Sensor selbst gehört ein Datenstrom, welcher im Modell durch eine qualifizierte Assoziation ausgedrückt wird. Diese Assoziation bedeutet, dass jeder Sensor pro ObservedProperty genau einen Datastream zugeordnet wird. Die Qualifizierung der Assoziation ermöglicht es auch, dass ein Sensor mehrere Datastreams zu unterschiedlichen ObservedProperties besitzen kann. Die Multiplizität von Datastream zu Sensor gibt an, dass verschiedene Datastreams zu einer ObservedProperty (mittles join(...)) zusammengeführt werden können.

Die Entität Datastream kann als das zentrales Objekt gesehen werden, welches die einzelnen Beobachtungen eines Sensors anhand der beobachtete Eigenschaft gruppiert. Dieses Datastream-Konzept ist von der STA abgeleitet. Die Beobachtung eines Datenstroms enthält den Wert der Messung. Ein Sensor ist für den beobachteten Wert verantwortlich, der mittels der Methode insert() hinzugefügt werden kann. Die beobachtete Eigenschaft gibt an, was beobachtet wurde. Da diese Eigenschaft für einen Datenstrom immer gleich ist, ist die Eigenschaft direkt mit dem Datenstrom verbunden. Weiterhin ist es möglich, dass verschiedene Datastreams zu einem Datastream kombiniert werden, insofern die beobachtete Eigenschaft übereinstimmt.

Die API des Bounded Context SensingDevice ermöglicht den Zugriff auf die Beobachtungen, sowie die Datenströme und verschiedene Metainformationen. Dies ermöglicht einer IoT-Anwendung den Zugriff auf die Daten. Außerdem kann eine Middleware die API nutzen, um neue Beobachtungen von Sensoren zu speichern. Die Middleware wird benötigt, da Sensoren unterschiedliche Datenformate liefern, die auf das gewünschte Format abgebildet werden müssen. Im Rahmen einer Microservice-Architektur eignen sich hier Ressourcen-basierte Schnittstellen, welche dem REST-Paradigma folgen.

Präzisierung der Domäneninhalte durch Constraints

Die Constraints erweitern die modellierten Bounded Context Entity Relation View der Domäne IoT (siehe Abschnitt 4.2.2 und ergänzen somit die Fachlichkeit der Domäne IoT um zusätzliche Restriktionen. Dadurch wird sichergestellt, dass die Domäne möglichst so abgebildet wird, wie diese in der Realität auch funktioniert. Die Constraints betreffen in der Regel die Domänenobjekte, welche durch die Attribute und die Methoden der Entitäten in der Entity Relation View repräsentiert werden. Für den Bounded Context SensingDevice sind mehrere Constraints von verschiedenen Entitäten (bspw. Observation, Datastream) notwendig. Um die Constraints zu formalisieren, werden OCL-Ausdrücke verwendet, welche die Domänenlogik des entsprechenden Bounded Contexts weiter spezifizieren. Listing 6.1 zeigt exemplarisch einen Auszug der OCL-Constraints mit Vor- und Nachbedingungen zur Methode insert() der Entität Datastream.

```
1  context Datastream::insert(observation: Observation)
2  pre: forAll(o:Observation) | observation.timestamp.before(o.timestamp)
3  post: (self.getObservaions() -> exists (o:Observation | o.equals(
       observaion))
4  and self.getObservations()@pre .forAll(o:Observation | self.
       getObservations().includes(o))
5  and self.getObservations().size() = self. getObservations()@pre.size()
        + 1
```

Listing 6.1: Erweiterung der Entity Relation View durch OCL-Ausdrücke

Soll eine neue Observation hinzugefügt werden, so lautet die Vorbedingung, dass die neue Observation vom Zeitstempel her ein aktuelleres Datum aufweisen als alle bereits eingetragenen Zeitstempel. Weiterhin muss die neueste Beobachtung vor den alten Zeitstempeln eingefügt werden, um eine geordnete Liste der Observations zu gewährleisten. In Zeile 3 des OCL-Ausdrucks wird überprüft, ob die neueste Beobachtung eingefügt wurde. Zusätzlich zu dem neuen Eintrag müssen auch noch alle alten Einträge vorhanden sein, was in Zeile 4 sichergestellt wird. Schließlich darf sich, wie in Zeile 5 spezifiziert, die Anzahl der Observations nur um eins erhöhen. Dieser Teil der Einschränkung ist notwendig, um sicherzustellen, dass nicht mehr als ein Eintrag hinzugefügt wurde.

6.2 Einbezug der Domäne im Kontext der Anwendungsentwicklung

Vor der Entwicklung von IoT-Anwendungen wird die IoT-Domäne gemäß dem Microservice-Engineering-Prozess modelliert. Anschließend wird die fortgeschrittene Web-Anwendungen gemäß dem Microservice-Entwicklungsprozess entwickelt, wobei verschiedene IoT-Belange wie Sensoren und IoT-Daten hinzugefügt werden. Abbildung 6.5 zeigt den erweiterten Prozess.

Infolgedessen müssen die bestehenden Analyse-Artefakte des Entwicklungsprozesses um weitere IoT-bezogene Artefakte erweitert werden. Die Anforderungsanalyse muss die Ableitung dieser Daten und der erforderlichen Sensoren, die für die Erfassung der erforderlichen Daten verantwortlich sind, einschließen. Wenn nicht bekannt ist, welcher Sensor die Daten liefert, kann die Anwendung ihre Anforderungen nicht erfüllen. Infolgedessen müssen die fehlenden Sensoren von der IoT-Domäne installiert und verwaltet werden.

Die Verwaltung der Sensoren und der IoT-Daten unterliegt der Domäne IoT ist, da diese Daten auch von anderen Anwendungen wiederverwendet werden könnten. Daher wird die Verwaltung und Handhabung von IoT-Daten in der Entwurfsphase von den geschäftlichen Belangen getrennt. Die etablierte Context Map hilft hier auf der strategischen Ebene bei der Einordnung der IoT-Belange beim

Entwurf von fortgeschrittenen Web-Anwendungen mit IoT-Bezug. Während des Entwurfs werden daher die IoT-Belange von der Geschäftsdomäne getrennt.

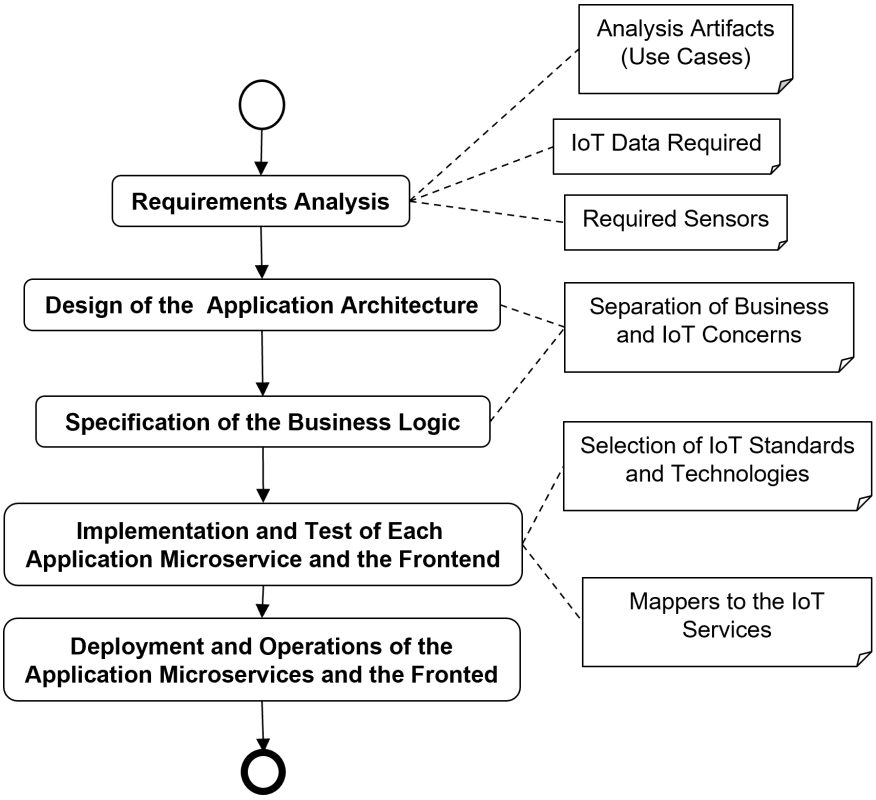


Abbildung 6.5: Erweiterung des Entwicklungsansatzes um IoT

Bei der Anwendungsentwicklung können ebenfalls verschiedene IoT-Standards herangezogen werden. So ist beispielsweise die SensorThings API ein etablierter Standard zur Verwaltung und Bereitstellung von IoT-Daten für Anwendungen. Vorhandene Implementierungen erleichtern die Implementierung. So bietet beispielsweise der FROST-Server [Fra-Fro] eine Implementierung des SensorThings-Standards an.

Oftmals haben IoT-Daten, die von der IoT-Domäne verarbeitet werden sollen, nicht direkt ein einheitliches Format. Die Daten müssen durch eine Transformation (z. B. durch eine Middleware) in das entsprechende Format gebracht werden. Anschließen können die IoT-Daten auf das Objekt der Geschäftsdomäne abgebildet werden, damit sie den Anforderungen der Geschäftsdomäne entsprechen.

Im Kontext der eingeführten CCSApp (siehe Kapitel 5) wird nun die Verknüpfung mit der Domäne IoT erläutert. Am Beispiel des Anwendungsfalls "View Car Status" wird der Einbezug der Domäne genauer erläutert.

Die dynamischen Daten, welche sich ändern und durch eine Sensorik eines Autos erbracht werden, sind im Anwendungsfall bei der Auflistung der Informationsanforderungen in Listing 6.2 festgehalten. Für eine saubere Trennung ist es erforderlich, dass diese Daten entsprechend der vorgestellten Domänenmodellierung der IoT-Domäne von der Anwendung separiert werden.

```
1  Title: View Car Status
2
3  Primary Actors: Fleet Manager
4
5  Preconditions:
6      - The car with the given VIN is part of the fleet
7  Postconditions:
8      - The Fleet Manager obtained Information about the current status
           of the car
9
10 Flow:
11 1. Actor selects the car with the given VIN from the fleet
12 2. System validates that the given VIN is valid and a car with the
       given VIN is present in the fleet
13 3. System loads the current status of the car from the Connected Car
       System
14 4. System displays the current status of the car to the Fleet Manager.
       This includes VIN, brand, model, production date, number of seats,
       fuel, fuel capacity, fuel level, consumption, position, door lock
       state and trunk lock state
15
16 Alternative flows:
17 ...
18
19 Information Requirements:
20     Connected Car System:
21     -   Static car data: VIN, brand, model, production date, number of
           seats, fuel, fuel capacity, consumption
22     -   Dynamic car data: fuel level, position, door lock state, trunk
           lock state
```

Listing 6.2: Use Case View Car Status

Dadurch, dass die IoT-Daten separiert werden, wird eine alternative Softwarearchitektur der CCSApp benötigt. Hierzu wird bei der Softwarearchitektur der Anwendung der Domänen-Microservice D-SensingDevice ergänzt. Dieser Microservice übernimmt die Domänenlogik für die IoT-Daten. Die IoT-Daten werden vom externen ConnectedCar System im D-SensingDevice bereitgestellt. Bei der Extraktion der dynamischen Daten ist zu beachten, dass die Datenhoheit der IoT-Daten bei dem Microservice SensingDevice liegt. Hierzu ist eine Transformation der Daten in das entsprechende

Format notwendig. Anschließend kann Die Anwendung CCSApp (und auch andere Anwendung) die Daten im homogenisierten Format verwenden. Der Bezug des Autos wird über die VIN hergestellt, da diese eindeutig für jedes Auto ist. Für die Entität SensingDevice lässt sich das Auto auch in mehrere Komponenten aufteilen, die verschiedene Sensoren anbieten. Wichtig ist, dass jede Komponente den Bezug zum Auto über die VIN herstellt. Abbildung 6.6 zeigt einen Ausschnitt der Softwarearchitektur der CCSApp mit der ergänzten Anbindung an die IoT-Domäne.

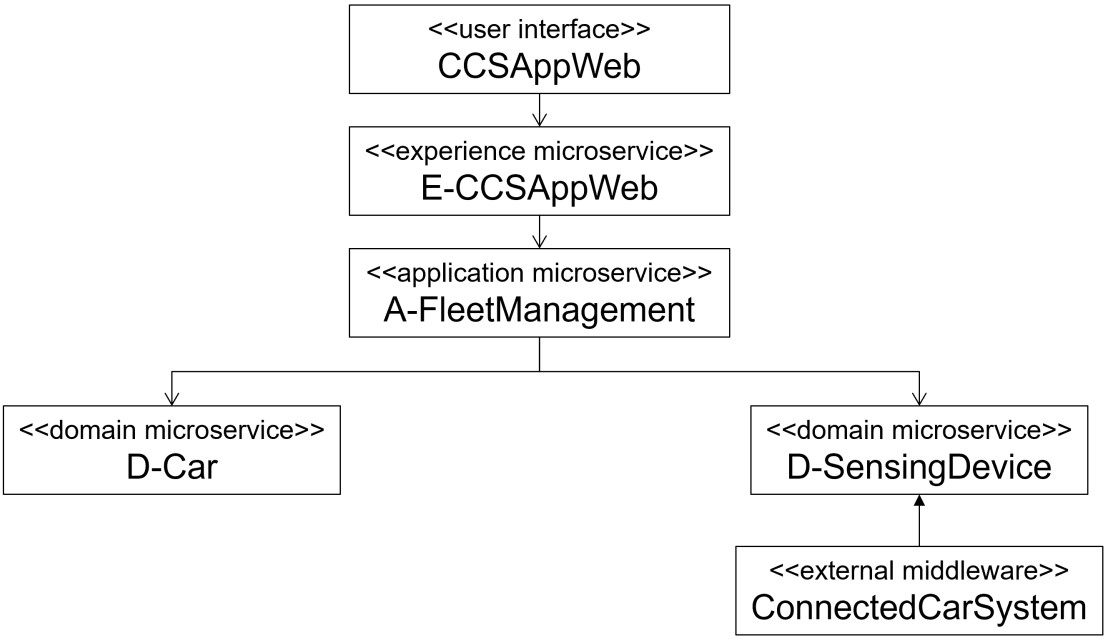


Abbildung 6.6: Architektur der CCSApp mit Bezug zur IoT-Domäne

Eine wichtige Aufgabe des Prozess-Microservices A-FleetManagement ist nun die Orchestrierung der erhaltenen IoT-Daten mit den Fahrzeugdaten aus dem Domänen-Microservice D-Car. Die dargestellte Middleware ist dafür zuständig, die Sensordaten nach dem gewünschten Format in den Domänen-Microservice einzupflegen. Dies ist notwendig, da aufgrund der Fragmentierung der IoT-Daten ein einheitliches Format erreicht werden muss [SH+18]. Diese Aufgabe übernimmt die Middleware. Gemäß der benötigten Anwendungslogik wird im nächsten Schritt die Verknüpfung der Geschäftslogik mit der Domäne hergestellt. Hierbei ist beim Entwurf zu beachten, dass die Belange der Geschäfts- und Querschnittsdomäne richtig modelliert werden. Der Bezug der Inhalte wird in der Regel durch einen Bezeichner (engl. IDentifier, ID) hergestellt.

Abbildung 6.7 zeigt das Orchestrierungs-Diagramm für die Microservice-Operation "View Car Status". Die Operation ermöglicht es dem Benutzer bspw. die aktuelle Position des Autos anzeigen. Die CCSApp-Benutzerschnittstelle wird zur Initiierung des Anwendungsfalls verwendet. Nach dem Empfang der Anfrage vom Frontend validiert A-FleetManagement die Parameter der Anfrage. An-

schließend wird der Domänen-Microservice D-Car nach den statischen Daten des Autos abgefragt. Wenn das Auto nicht existiert, dann wird eine entsprechende Fehlermeldung an die Benutzerschnittstelle zurückgegeben. Andernfalls werden die aktuellen Positionsdaten (und weitere dynamische Daten) des Autos aus dem Domänen-Microservice D-SensingDevice angefragt.

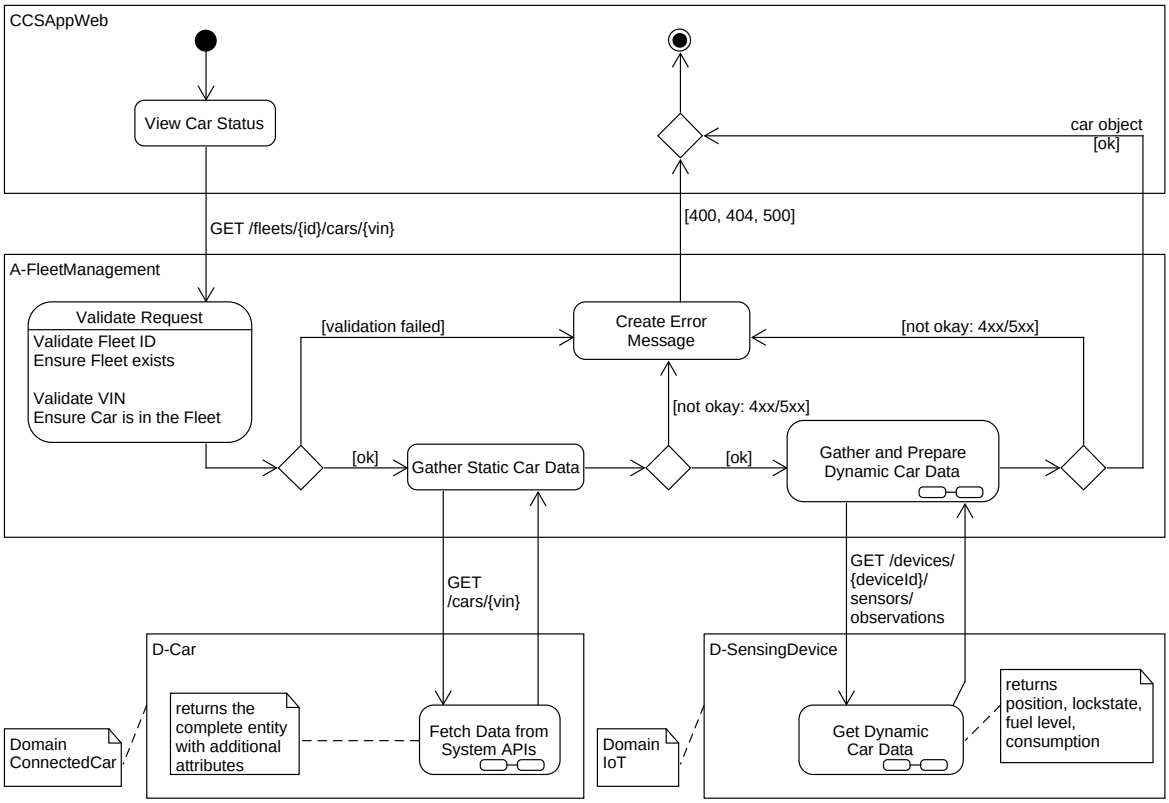


Abbildung 6.7: Orchestrierungs-Diagramm "View Car Status"

Die erhaltenen Daten werden anschließend vom Microservice A-FleetManagement entsprechend der gewünschten Funktionalität orchestriert. Basierend auf dem Anwendungsfall wird nur die letzte Beobachtung des entsprechenden Sensorwerts benötigt, weshalb nur dieser angefragt wird.

Die Verwaltung der IoT-Daten in der eigenen Domäne erlaubt aber auch die Bereitstellung weiterer Funktionalitäten, wie beispielsweise die Bereitstellung der Verlauf des Spritverbrauchs oder die Fahrstrecke des Fahrzeugs basierend auf den historischen Daten des Fahrzeugs. Ein entsprechender Fahrzeugnutzer müsste hier für die Nutzung der Daten noch seine Zustimmung einräumen. Dies wird im weiteren dieser Arbeit jedoch nicht betrachtet.

6.3 Zusammenfassung

In diesem Kapitel erfolgte eine detaillierte Aufstellung einer Querschnittsdomäne, die im weiteren Schritt bei der Anwendungsentwicklung eingesetzt wurde. Konkret wurde die Querschnittsdomäne des Internets der Dinge (Internet of Things, IoT) herangezogen und anhand des Domänenentwicklungsprozesses aufgestellt. Anhand verschiedener Quellen wie Standards und zu entwickelnder Anwendungen wurden die Fachlichkeit und die benötigten Inhalte festgehalten und die Microservice-Architektur für IoT erfasst. Dazu wurden Quellen analysiert und ein fachliches Modell abgeleitet. Grundlegend aus dem Querschnittsbereich waren hierbei IoT-Standards, die die benötigte Fachlichkeit beinhalteten.

Die daraus resultierende Context Map fasst auf der strategischen Ebene die Inhalte fachlich zusammen. Die Bounded Contexts wurden vom konzeptionellen Modell abgeleitet. Im nächsten Schritt wurde exemplarisch für den Bounded Context SensingDevice das taktische Modell aufgestellt und mit Domänen-Constraints präzisiert. Die daraus resultierende "Bounded Context Entity Relation View" wird in der Anwendungsreferenz weiter behandelt. Dazu wird der zuvor aufgestellte Anwendungsentwicklungsprozess um den IoT-Anteil erweitert. Als Resultat beinhaltet die Software-Architektur (im Beispiel die CCSApp) weitere Softwarekomponenten in der Form von Microservices. Durch die Orchestrierungs-Diagramme erfolgt letztendlich die Einordnung des dynamischen Bezugs zu den Microservices, die von der Querschnittsdomäne bereitgestellt werden. Durch das Erfassen der Querschnittsdomäne und deren Funktionalität werden eine Modellierung und Softwarebausteine bereit gestellt, die sich für die Entwicklung weiterer Anwendungen eignen.

7 Implementierungs- und Testkonzept für fortgeschrittene Web-Anwendungen

Der in den vorangegangenen Kapiteln eingeführte systematische Engineering-Ansatz liefert verschiedene Artefakte für die Anforderungen, den Entwurf sowie die Implementierung der Software, die für die Tests relevant sind. Hierbei ist es essenziell, dass das Vorgehen und die Artefakte so gestaltet sind, dass sich aus diesen strukturerhaltend und systematisch die Implementierung und die Tests ableiten lassen. Ein Beispiel für ein solches Artefakt sind die Constraints, die sich systematisch in die Implementierung und Tests überführen lassen. Weiterhin werden Herangehensweisen für die Erstellung der Tests anhand Richtlinien aufgegriffen, welche die Umsetzung der Tests erleichtern und systematisieren. Durch die Vorgabe einer konkreten Mikroarchitektur wird die Struktur der Implementierung und die Platzierung der Tests zudem weiter verfeinert. Dadurch ergibt sich eine Verbindung der Strukturen zur Implementierung, den Tests und der vorgegebenen Mikroarchitektur. Diese Mikroarchitektur eignet sich für die Testsystematik, da dort die Artefakte in der Architektur platziert werden. Zudem können Vorlagen geschaffen werden, die die gewünschte Struktur schon beinhalten und die Implementierung und das Testen fortgeschrittener Web-Anwendungen unterstützen.

Zu Beginn wird in Abschnitt 7.1 auf die Artefakte in Bezug auf die Tests eingegangen. Es wird erläutert, wie die verschiedenen Testarten (Unit-, Integrations- und Ende-zu-Ende-Tests) im Engineering-Ansatz abgeleitet werden. Dabei wird gemäß der Testpyramide von unten nach oben mit den Unit-Tests in Abschnitt 7.2 begonnen und auf die systematische Ableitung der Tests eingegangen. Innerhalb des Abschnittes wird zwischen der anwendungsagnostischen Domänenfunktionalität der Domänen-Microservices und der Anwendungslogik der Anwendungs-Microservices unterschieden. Nach der Umsetzung der Unit-Tests werden die Integrationstests in Abschnitt 7.3 und die Herleitung dieser aus den aufgestellten Artefakten betrachtet. Abschließend wird in Abschnitt 7.4 im Bereich der Ende-zu-Ende-Tests eine Möglichkeit zur systematischen Aufstellung dieser Tests aufgezeigt. Jeder der Abschnitte enthält die Ableitung der Testdaten sowie die Überführung dieser in die entsprechenden Tests und die automatisierte Ausführung in der CI/CD-Pipeline. Anhand des Fallbeispiels der Anwendung ConnectedCarServicesApplication (CCSApp) aus Kapitel 5 werden die eingeführten Konzepte vertieft. Hierbei werden exemplarisch Ausschnitte der Architektur, der Schnittstellen und Ausschnitte aus der Implementierung herausgezogen.

7.1 Implementierung und Testen im Engineering-Ansatz

Die Entwicklung einer Microservice-basierten Anwendung folgt entlang des eingeführten Microservice-Engineering-Ansatzes aus Kapitel 4. Ein wichtiger Punkt bei der Implementierung und dem Testen ist, dass die Ableitung des Codes und der Tests. In diesem Abschnitt wird geklärt, wie der Entwickler beim Testen mit den aufgestellten Artefakten systematisch vorgehen kann. Ein Teil ist die Klärung, welche Tests vom Entwickler wo und wann im Microservice-Engineering-Ansatz umgesetzt werden. Die Neuerung sind hierbei nicht die Tests an sich selbst, sondern die systematische Erfassung der Tests. Insbesondere wird dadurch geklärt, wie die verschiedenen Testarten in dem Engineering-Ansatz aus den Artefakten gezielt abgeleitet werden. Die Grundlage für die Tests werden hierbei schon in der Analysephase geliefert. Durch die Anforderungen werden bereits die Akzeptanzkriterien festgelegt, welche das Softwaresystem erfüllen soll. Daraus können die Entwickler die Ende-zu-Ende-Tests ableiten. Diese dienen einer gesamtheitlichen Überprüfung des Softwaresystems [Re19].

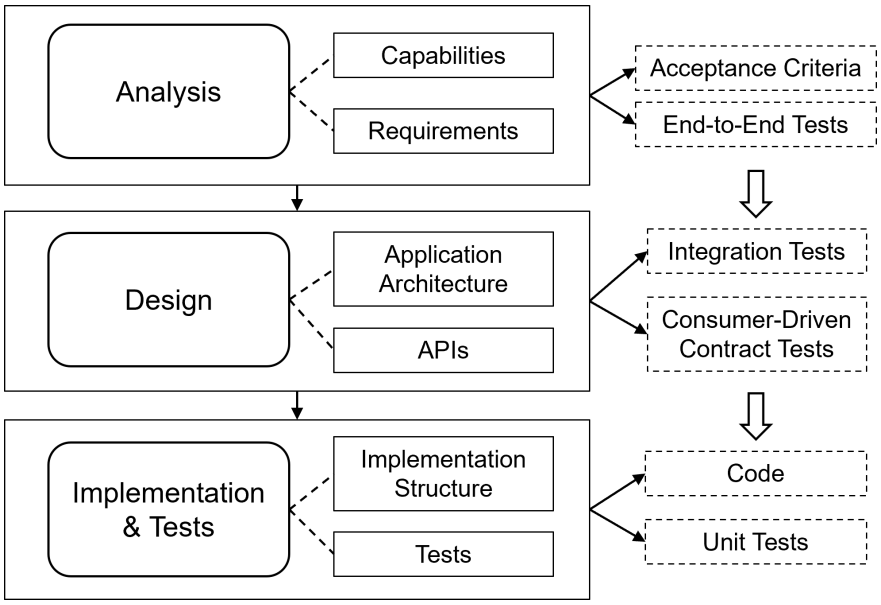


Abbildung 7.1: Entwicklungsphasen und Tests

Neben den Anforderungen hat auch die betrachtete Microservice-Architektur eine hohe Relevanz. Im Entwurf erfolgt die Aufteilung in die verschiedenen Microservice-Typen. Hierbei wird basierend auf den APIs die Integration der Microservices getestet. Im Falle der aufgestellten Architektur tritt hier die Besonderheit auf, dass zwischen den Anwendungs- und Domänen-Microservices unterschieden wird. Die Domänenlogik wird unabhängig von der Anwendung getestet, da diese Funktionalität von vielen Anwendungen wiederverwendet werden soll. Hierfür werden innerhalb der Microservices Unit-Tests platziert, um die kleinen Funktionsblöcke (beziehungsweise Methoden) isoliert zu testen. Die einzelnen Microservices können unabhängig voneinander ausgeliefert werden und müssen entsprechend

über die Schnittstellen angesprochen werden. Welche Tests für eine Anwendung benötigt werden, ergibt sich hierbei aus dem Microservice-Engineering-Ansatz und den Artefakten, die währenddessen erstellt werden.

7.1.1 Überblick über die Anwendung und den Testaufbau

Die Anforderungen legen die funktionalen Anforderungen fest, welche von einer Anwendung erfüllt werden müssen. Neue Anforderungen führen insbesondere auch zu neuen Testfällen [Wi16]. Hierbei ist nicht das Ziel, eine vollständige Testabdeckung zu erreichen, sondern die Anzahl an Fehlern möglichst zu reduzieren, da Tests die Anwesenheit von Fehlern nachweisen. Hierbei sollte mit den Tests möglichst früh begonnen werden, da diese bei einer späteren Entdeckung in der Regel höhere Kosten verursachen. Basierend auf den Anforderungen lassen sich bereits Testfälle für die Anwendung erfassen, die die Funktionalität der Anwendung (und insbesondere auch die Akzeptanz) sicherstellen. Eine Microservice-basierte Anwendung besteht aus mehreren verteilten Microservices. Um eine solche Anwendung im Gesamten zu testen, müssen für die Ende-zu-Ende-Tests alle benötigten Microservices zur Verfügung stehen. Üblicherweise erfolgt die Testdurchführung mittels einer Testumgebung. Die Ausführung der Tests wird über eine CI/CD-Pipeline automatisiert, so dass bei einer Änderung die bisherige gewünschte Funktionalität erhalten bleibt, aber auch die neue Funktion hinreichend getestet wird. Hierbei ist wichtig, dass der Entwickler bei der Entwicklung von Web-Anwendungen mittels einem Testkonzept unterstützt wird. Diese Arbeit fokussiert die Unterstützung der Entwickler der fortgeschrittenen, Microservice-basierten Web-Anwendungen. Faragó et al. [FS19] identifizierten Fehlerquellen in Microservice-Systemen anhand einer Umfrage. Diese Umfrage führt zu der Schlussfolgerung, dass eine auf Microservices basierende Architektur mehr Tests auf der oberen Ebene erfordert, insbesondere bei den Ende-zu-Ende-Tests, da die Interaktion zwischen den Microservices entscheidend ist. Daher werden die häufigsten Fehlerursachen von Microservice-Anwendungen diskutiert und eine geänderte Testverteilung vorgeschlagen, die Integrations- und Ende-zu-Ende-Tests in den Fokus des Testens hebt [FS19].

Abbildung 7.2 zeigt das in der Arbeit behandelte Testkonzept auf einer hohen Abstraktionsebene. Auf der Testebene stellen Unit-Tests die erste Ebene des Testkonzepts dar. Diese Tests sollen gewährleisten, dass die internen Funktionen eines Microservices ihre Funktionalität erfüllen und in der Testphase der Pipeline unmittelbar nach dem Schritt "Build" ausgeführt werden. Die Unit-Tests werden hier gemäß Olan [Ol03] frühzeitig in der Softwareentwicklung eingesetzt und häufig ausgeführt. Die Schnittstellen der Microservices (Endpunkte, APIs) werden mittels Integrationstests von außen getestet, um sicherzustellen, dass der Microservice die geforderten API-Funktionalität (API Contract) erfüllt. Um einen Microservice isoliert von anderen Microservices zu testen, wird auf das Konzept der konsumentengetriebenen Vertragstests (engl. Consumer-Driven Contract tests, CDC tests) verwendet. Diese Integrationstests werden ebenfalls in den CI/CD-Prozess einbezogen, um die

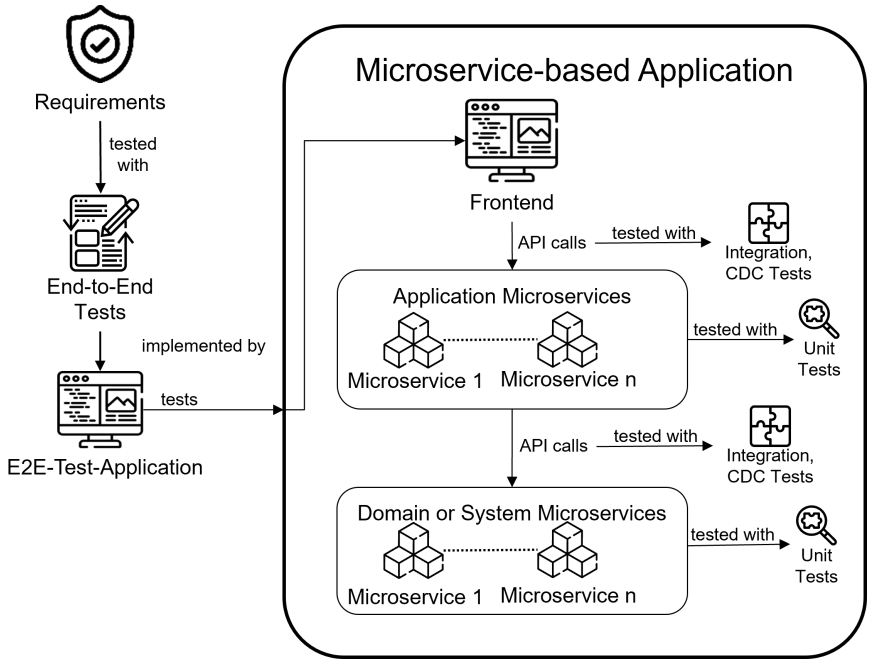


Abbildung 7.2: Überblick über das Test-Konzept für fortgeschrittene Web-Anwendungen

gewünschte API-Funktionalität der Microservices zu verifizieren. Die Anforderungen und die daraus resultierenden Akzeptanzkriterien werden durch Ende-zu-Ende-Tests überprüft. Mit diesen Tests wird zudem die Funktionalität des Frontends getestet. Hierbei unterscheidet sich die Platzierung der Tests von denen der Microservices. Während die Unit-Tests direkt bei der Implementierung des Frontends platziert sind, wird für die Akzeptanztests eine zusätzliche Test-Anwendung (E2E-Test-Application) eingeführt. Diese von Rahman und Gao [RG15] vorgeschlagene dedizierte Testanwendung trennt die Akzeptanzkriterien von der Implementierung der einzelnen Services, indem diese die Tests unabhängig von der Anwendung implementiert, wodurch die Akzeptanztests von der Anwendung getrennt werden.

Dadurch entsteht der Vorteil, dass die Ende-zu-Ende-Tests unabhängig von der Benutzeroberfläche implementiert werden. Insbesondere wenn verschiedene Benutzeroberflächen (Desktop, Mobil) umgesetzt werden, können Teile der Tests wiederverwendet werden. Was beachtet werden muss ist, dass zur Durchführung der Ende-zu-Ende-Tests der Zugriff auf die verschiedenen Elemente der Benutzeroberfläche benötigt wird. Beispiele hierfür sind Schaltflächen oder Formulare, die von einem Benutzer ausgefüllt oder betätigt werden. Für die Tests bedeutet dies, dass für die konkreten HTML-Elemente bekannt sein müssen, damit diese innerhalb der Tests (beispielsweise für Aufrufe in Cypress [Cyp-Jav]) angesprochen werden können.

Anhand des betrachteten Fallbeispiels aus Kapitel 5 werden die Konzepte genauer erläutert. Die betrachtete Anwendung CCSApp behandelt die Verwaltung von Fahrzeugen und die Überwachung

von dessen Komponenten. Eines der Ziele der Anwendung ist neben der Vermietung von Fahrzeugen die Informationsabfrage von bestimmten Fahrzeugdaten, wie beispielsweise der Tank- oder Ladestand. Abbildung 7.3 zeigt die konkrete Software-Architektur der Anwendung und die Platzierung der Tests auf den verschiedenen Ebenen der Architektur.

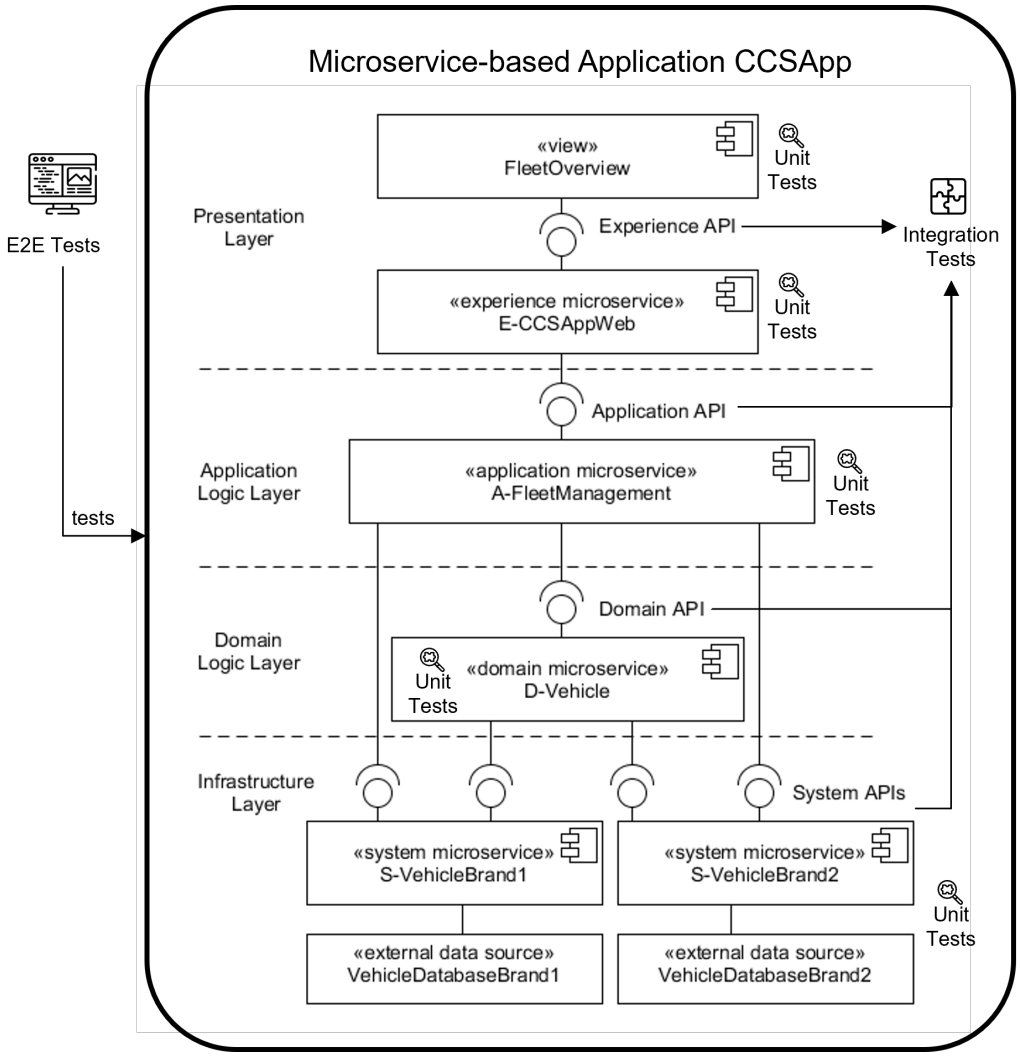


Abbildung 7.3: Testkonzept angewendet auf die Architektur der CCSApp

Jeder Microservice und die Benutzerschnittstelle werden mit Unit-Tests getestet. Die Integrationstests finden an den jeweiligen Schnittstellen der Microservices statt. Hierbei werden für alle Microservice-Typen entsprechende Integrationstests bzw. konsumentengetriebe Vertragstests eingesetzt. Die gesamte Anwendung (und somit das Zusammenspiel aller Microservices) wird anhand der Ende-zu-Ende-Tests überprüft.

Neben den Tests zur Erfüllung der Funktionalität können weitere Faktoren berücksichtigt werden. Beispielsweise werden die Testabdeckung und verschiedenen Werkzeuge zur Messung von nicht-

funktionalen Anforderungen (wie beispielsweise die Antwortzeit und Code-Qualität) berücksichtigt.

Ein weiterer Punkt ist die Dokumentation der Tests. Darunter fällt, welche Eingaben erfolgt sind, welche Testfälle ausgeführt wurden und welche (erwarteten) Ausgaben erzeugt wurden. Anhand eines Testprotokolls können die Ergebnisse der Tests festgehalten werden.

7.1.2 Ableitung der Tests aus den Artefakten

Die Entwicklung von Tests folgt einer durch die Testpyramide vorgegeben Reihenfolge, beginnend von unten mit den Unit-Tests auf der Domänenebene, über die Unit-Tests der Anwendungs-Microservices, über die Schnittstellentests hin zu den Ende-zu-Ende-Tests. Im Gegenzug zum Testansatz von [SR+18] werden die verschiedenen aufgestellten Artefakte in die Erstellung der Tests mit einbezogen. Eine Übersicht über die verschiedenen Arten von Tests und die zugehörigen Artefakte ist in Abbildung 7.4 dargestellt.

Bei der Entwicklung wird zwischen den Domänen- und Anwendungs-Microservices unterschieden. Da die Entwicklung von Microservice-basierten Anwendungen mit den Domänen-Microservices beginnt, werden die Unit-Tests für diese Microservices frühzeitig erstellt. Die Domänen-Microservices erfüllen meist Funktionalität, welche durch die Operationen Create, Read, Update und Delete (CRUD) ausgedrückt werden kann. Die entsprechenden Unit-Tests werden daher direkt aus den Domänen-Constraints abgeleitet. Bei der Implementierung von Anwendungs-Microservices werden dagegen zwei Arten von

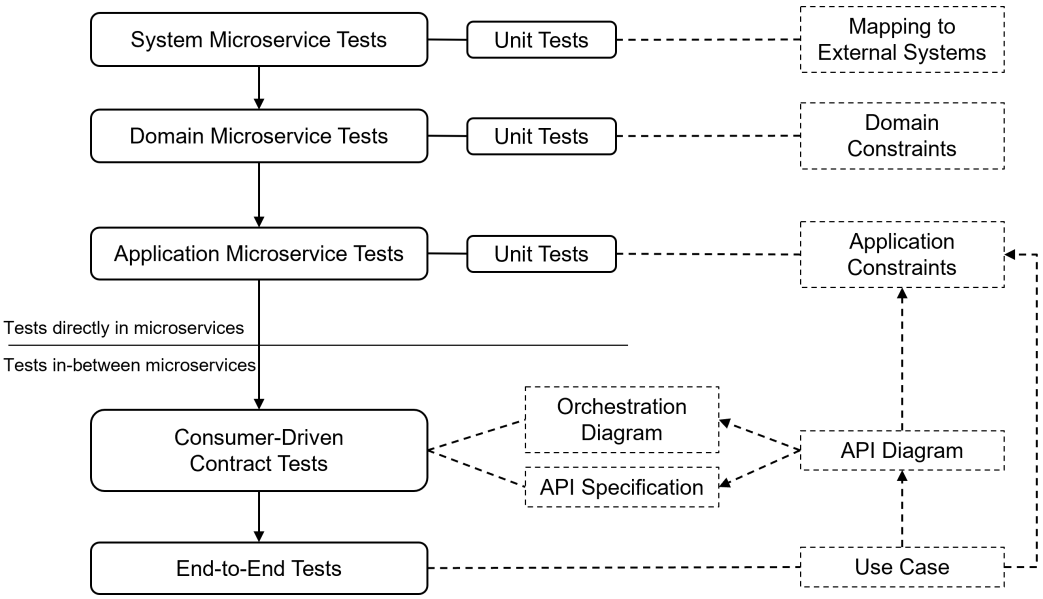


Abbildung 7.4: Testkonzept und beteiligte Artefakte

Tests entwickelt. Die Backend-Akzeptanztests und die Unit-Tests, die von den erstgenannten abgeleitet

werden. Die Backend-Akzeptanztests ergeben sich aus den einzelnen Anwendungsfällen. Nach der Implementierung von mindestens einem Domänen-Microservice beginnt die Entwicklung eines oder mehrerer Anwendungs-Microservices einschließlich ihrer Tests, wobei die Funktionalität aus den Anwendungsfällen abgeleitet wird. Wenn es mindestens zwei Microservices gibt, die miteinander kommunizieren, werden die Schnittstellen mittels Integrationstests, insbesondere konsumentengetriebene Vertragstests (Consumer-Driven Contract, CDC) getestet. Aus den erstellten Artefakten sind hierbei das Orchestrierungs-Diagramm (siehe Abschnitt 5.2.4) und die API-Spezifikation ausschlaggebend zur Ableitung der Tests. Ein Orchestrierungs-Diagramm definiert, welche Microservices miteinander kommunizieren und auf welche Daten sie zugreifen. Die API-Spezifikation hingegen zeigt, wie die Anfrage und die Antwort des API-Aufrufs spezifiziert werden. Die Ende-zu-Ende-Tests bilden die übergeordneten Tests, für die die meisten Microservices der Anwendung bereits umgesetzt sein müssen, bevor diese verwendet werden können, um die Anwendung zu testen. Die Ende-zu Ende-Tests lassen sich aus den Anwendungsfällen ableiten. Hierbei bilden jeder Fluss und alternative Fluss ein mögliches Szenario ab, welches durch Ende-zu-Ende-Tests getestet wird.

Eine weiterer Punkt ist, dass die gesamte Mikroarchitektur des Microservices, wie in Abbildung 7.5 dargestellt, durch die Tests ausreichend getestet wird. Jeder Microservice implementiert und stellt eine API bereit, die durch die API-Spezifikation dieses Microservices definiert ist.

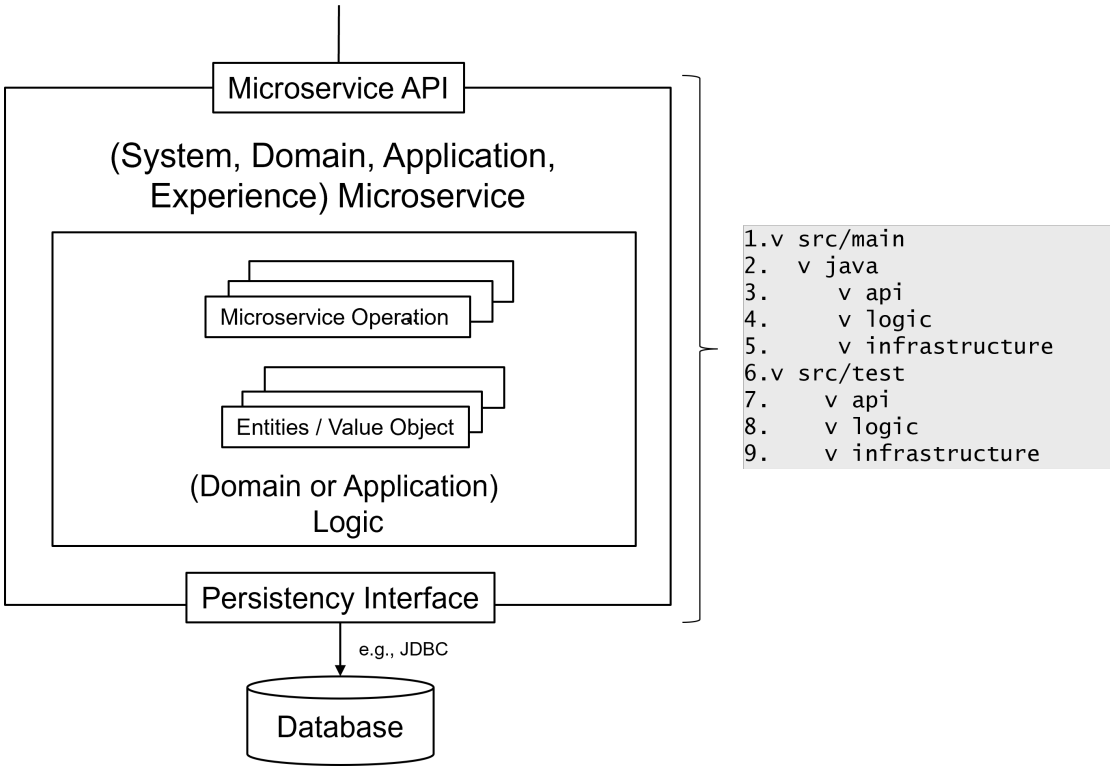


Abbildung 7.5: Mikroarchitektur eines Microservices

Ein zentraler Bestandteil der Mikroarchitektur eines Microservices ist die implementierte Domänenlogik oder Anwendungslogik. Diese Logik ist entsprechend den von der API definierten Operationen strukturiert. Im Falle eines Domänen-Microservice beziehen sich die Operationen auf eine Domänenentität und sind in der Regel CRUD-Operationen. Im Falle eines Anwendungs-Microservices implementiert eine Operation üblicherweise eine Benutzer-/System-Interaktion, die durch die Anforderungsspezifikation festgelegt ist. Hier steht ein Funktionsaufruf im Vordergrund, der auf eine REST-Operation abgebildet werden muss. Ein weiterer Bestandteil ist die Schnittstelle zu der Datenbank. Hierbei ist in der Mikroarchitektur ein Bereich vorgesehen, der diesen Aspekt behandelt. Diese Einheit bietet somit eine Entkopplung der Schnittstelle zur Datenbank, über die die verschiedenen Domänenobjekte persistiert werden.

Neben den einzelnen Microservices ist auch das Zusammenspiel und die Integration der Microservices ein zentraler Bestandteil des Testkonzepts. Hier sind insbesondere die Schnittstellen und die Korrektheit der Orchestrierung der verschiedenen Microservices zu testen. Ein wichtiges Kriterium für die Tests ist hierbei, dass möglichst viele der Microservice-Operationen abgedeckt werden. An dieser Stelle sind insbesondere die Interaktionen des Benutzers mit dem System wichtig.

Für die Implementierung der einzelnen Services kann dem Entwickler eine weitere Hilfestellung geboten werden. Hier können durch entsprechende Projektvorlagen die Implementierung und Umsetzung der Tests vereinfachen werden. Durch die Vorgabe der Strukturierung wird die Mikroarchitektur vorgegeben und die entsprechende Funktionalität und deren Tests werden an den entsprechenden Stellen implementiert. Hierdurch ergibt sich eine strukturierte Implementierung.

7.2 Unit-Tests zur Überprüfung der Domänen- und Anwendungslogik

Eine wichtige Eigenschaft des Testkonzepts ist, dass die Unit-Tests nicht willkürlich, sondern nach einem bestimmten systematischen Vorgehen abgeleitet werden. Für die Ableitung der Unit-Tests wird hierbei auf die im Microservice-Engineering-Ansatz aufgestellten Artefakte zurückgegriffen.

Nach dem ANSI/IEEE Standard 829 [ANS829] ist die Erstellung der Testdaten ein wichtiges Kriterium. Hierbei handelt es sich um den Ist-Stand der Daten, sowohl als auch der erwarteten Ausgabedaten. Gerade bei der Spezifikation der Testfälle sind diese Daten von zentraler Bedeutung. Die festgelegten Daten ermöglichen die Überprüfung des Systems auf die korrekte Ausführung der Funktionalität [Wi16].

7.2.1 Methodisches Vorgehen zur Ableitung der Unit Tests aus der Domänenlogik

Während bei den Anwendungs-Microservices mehr Unit-Tests in Richtung der Anwendungslogik benötigt werden, wird bei den Domänen-Microservices die Korrektheit des implementierten Domänenwissens überprüft. Hierbei spielt insbesondere die aufgestellte Bounded Context Entity Relation View (siehe Abschnitt 4.2.1) eine zentrale Rolle. Die Tests der Domänenlogik konzentriert sich auf die Domänenfunktionalität, die in der Mikroarchitektur in der Ebene der Domänenlogik umgesetzt ist. Ein weiterer Punkt ist die Persistenz der Entitäten, die über die entsprechende Schnittstelle zugegriffen wird.

Abbildung 7.6 zeigt den Zusammenhang der Artefakte und deren Ableitung. Die Domänen-Constraints werden anhand der Bounded Context Entity Relation View und dem etablierten Domänenwissens hergeleitet. Weiterhin wird aus der Bounded Context Entity Relation View ein API-Diagramm zur Ableitung der API-Spezifikation erstellt. Das zentrale Artefakt zur Ableitung der Unit-Tests für die Domänen-Microservices sind die aufgestellten Constraints. Aus diesen lassen sich verschiedene Tests ableiten. Hieraus lassen sich ebenfalls parametrisierte Tests gewinnen, wodurch eine einzige Testmethode mehrfach mit unterschiedlichen Parametern ausführen werden kann wodurch verschiedene Code-Pfade durchlaufen und getestet werden können [TS05]. Zusätzlich lassen sich aus der API-Spezifikation Unit-Tests ableiten, da diese von der implementierten Logik unterstützt werden soll.

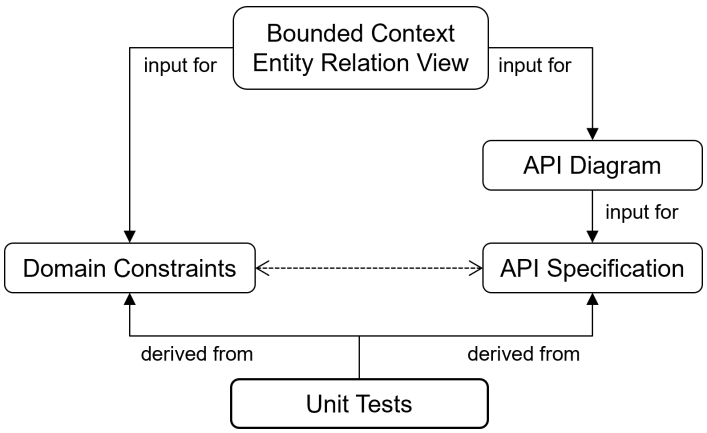


Abbildung 7.6: Ableitung der Unit-Tests im Rahmen der Domänen-Microservices

Ein Domänen-Microservice implementiert das anwendungsagnostische Domänenwissen, welches bestimmte Constraints beinhaltet. Diese beinhalten Invarianten für die Attribute sowie Vor- und Nachbedingungen für Funktionsaufrufe. Beispielsweise muss die Fahrzeugidentifikationsnummer (Vehicle Identification Number, VIN) einem bestimmten Format folgen. Die Domänenlogik wird im entsprechenden Microservice implementiert und getestet. Hierbei können verschiedene Arten von Contraints auftreten. Attribute dürfen nicht leer sein, Invarianten müssen Bestand haben, Vor- und

Nachbedingungen müssen erfüllt und bestimmte Grenzen müssen eingehalten werden. Die Constraints werden bei der Analyse der Domäne festgehalten, so dass diese für das Testen herangezogen werden können. Die Object Constraint Language (OCL) wird zur formalen Spezifikation der Constraints [OMG-OCL] verwendet.

```
1 context Car inv:
2   allInstances()->forAll(c1, c2: Car | c1.vin <> c2.vin)
3   and self.vin.matches(^[A-HJ-NPR-Z0-9]{13}[0-9]{4}$)
```

Listing 7.1: Sicherstellung einer korrekten VIN

Listing 7.1 zeigt eine Invariante der Entität Car. Die Constraints beschreiben die Regeln der Domäne. In diesem Fall muss die VIN einem bestimmten Format entsprechen. Werden die Domänenbeschränkungen verletzt, muss die Ausführung der Methode gestoppt und der Benutzer des Systems benachrichtigt werden. Die Constraints Beschränkungen in der Regel das Ergebnis einer Methode.

Die korrekte Implementierung der Constraints in der Domänenlogik wird durch entsprechenden Unit-Tests sichergestellt. Hierbei eignen sich die Constraints um die Unit-Test abzuleiten und zu entwickeln um die korrekte Implementierung der Domänenlogik zu überprüfen. Ein weiterer Vorteil der sich hieraus ergibt, ist, dass die Constraints eine systematische Ableitung der Testdaten erlauben. Genauer lassen sich aus den spezifizierten Invarianten fehlererzeugende und korrekte Testfälle ableiten. Neben den Invarianten sind auch die definierten Methoden ein wichtiger Ausgangspunkt. Bei einem OCL-Contraint werden hier immer eine Vor- und eine Nachbedingung definiert.

Eine Reduzierung der Unit-Tests lässt sich weiterhin durch Anlehnung an Äquivalentklassentests [BH+13] erreichen. Hierbei werden durch wenige Unit-Test die Wertebereiche abgedeckt. Bei dem Verfahren werden nicht die Grenzfälle der Invarianten einzeln betrachtet, sondern die Überschneidungen werden berücksichtigt. Bei der Umsetzung eines Unit-Tests werden die Überschneidung entsprechend zusammengelegt. Im Beispiel des Tankstands eines Autos resultieren drei Bereiche. Der Bereich für korrekte Testdaten umfasst die Zahlen Null (leer) bis einhundert (voll). Ein negativer Tankstand oder ein vollerer Tankstand stellen den Bereich der fehlererzeugenden Testdaten dar.

Sollte für ein Attribut kein Constraint existieren und es werden keine weiteren Einschränkungen gemacht, existiert keine konkrete Vorgabe an die Testdaten. Anhand der Testdatenspezifikation sollte entsprechend das Datum vorgegeben werden. Beispielsweise existiert für das Attribut der Fahrzeugmarke (brand) der Entität "Car" kein Domänen-Constraints. Ein entsprechender Test wäre somit auch mit dem Beispielwert "Muster" zulässig. Für sinnvolle Tests sind verständliche, konkrete Werte vorzuziehen, so dass weitere involvierte Personen ebenfalls die Semantik und den Inhalt der Tests verstehen.

Vorgehen zur Implementierung und zum Testen eines Domänen-Microservices

Wie in Abbildung 7.7 dargestellt, wird analog zum Vorgehen der Implementierung der Mikroarchitektur (siehe Abschnitt 5.3.2) die Implementierung umgesetzt. Ergänzend werden an den entsprechenden Stellen die Tests vorgesehen. Die Bounded Context Domain Entity View bzw. das erstellte API-Diagramm wird auf die Code-Ebene abgebildet. Für jedes Entitäts- und Wertobjekt wird zuerst eine eigene Klasse mit den jeweiligen Attributen und Methoden angelegt. Die zum Modell gehörenden Operationen eines Domain Microservice werden anschließend implementiert. Die Bezüge zu den Tests wird anhand der Domänen-Constraints hergestellt. Hierbei werden die zuvor abgeleiteten Testfälle an den entsprechenden Stellen implementiert. Constraints bezüglich des Domänenwissens werden zu den einzelnen Operationen vorgesehen. Hierbei werden die zuvor abgeleiteten Testfälle verwendet.

Die Persistenzschnittstelle definiert die Methoden für den Zugriff auf und die Manipulation von Entitäten. Danach wird die API-Schnittstelle erstellt, die die API-Endpunkte für die CRUD-Operationen enthält. Die ankommenden HTTP-Anfragen werden in dem API-Controller verarbeitet und stellen eine RESTful-API für die angegebenen Operationen (im Rahmen der Arbeit in OpenAPI) bereit. Der APIController implementiert die API-Schnittstelle. Bei der Umsetzung der Unit-Tests zu der API-Spezifikation werden die Testfälle hierzu herangezogen.

Eine Persistenz-Entität stellt eine in einer Datenbank gespeicherte Tabelle dar. Die verschiedenen Mapper bilden die Persistenz-Entitäten auf die im Modell implementierten Domänen-Entitäten (oder umgekehrt) ab. Die definierte Persistenzschnittstelle wird zuletzt implementiert. Unit-Tests zur Absicherung der korrekten Transformation finden auf dieser Ebene statt.

7.2.2 Methodisches Vorgehen zur Ableitung der Unit-Tests aus der Anwendungslogik

Für die Ableitung der Tests für die Anwendungslogik wird ein ähnliches Vorgehen hinsichtlich der Ableitung der Testdaten eingesetzt. Im Vergleich zu der Ableitung der Tests für die Domänen-Microservices (Abbildung 7.6) werden weitere Artefakte bei der Erstellung der Tests berücksichtigt. Für die Anwendung liefern die Artefakte der Analyse und des Entwurfs die Grundlage für die umzusetzenden Tests. So stellen die Anwendungsfälle aus der Analysephase für die Entwurfsphase zunächst das zentrale Artefakt dar. Aus diesen werden das API-Diagramm, die API-Spezifikation und die Orchestrierungs-Diagramme aufgestellt. Die beteiligten Artefakte sind in Abbildung 7.8 in Beziehung gesetzt. Insbesondere beschreiben die Anwendungsfälle in Kombination mit den Orchestrierungs-Diagrammen den Ablauf innerhalb eines Anwendungs-Microservices. Basierend auf den Abläufen lassen sich Unit-Tests ableiten. Die Unit-Tests sollen hierbei die interne Funktionalität des Microservices anhand der beschriebenen Funktionalitäten abdecken. Durch die Ableitung der Tests können auch Fehler in den spezifizierten Anwendungsfällen entdeckt und behoben werden.

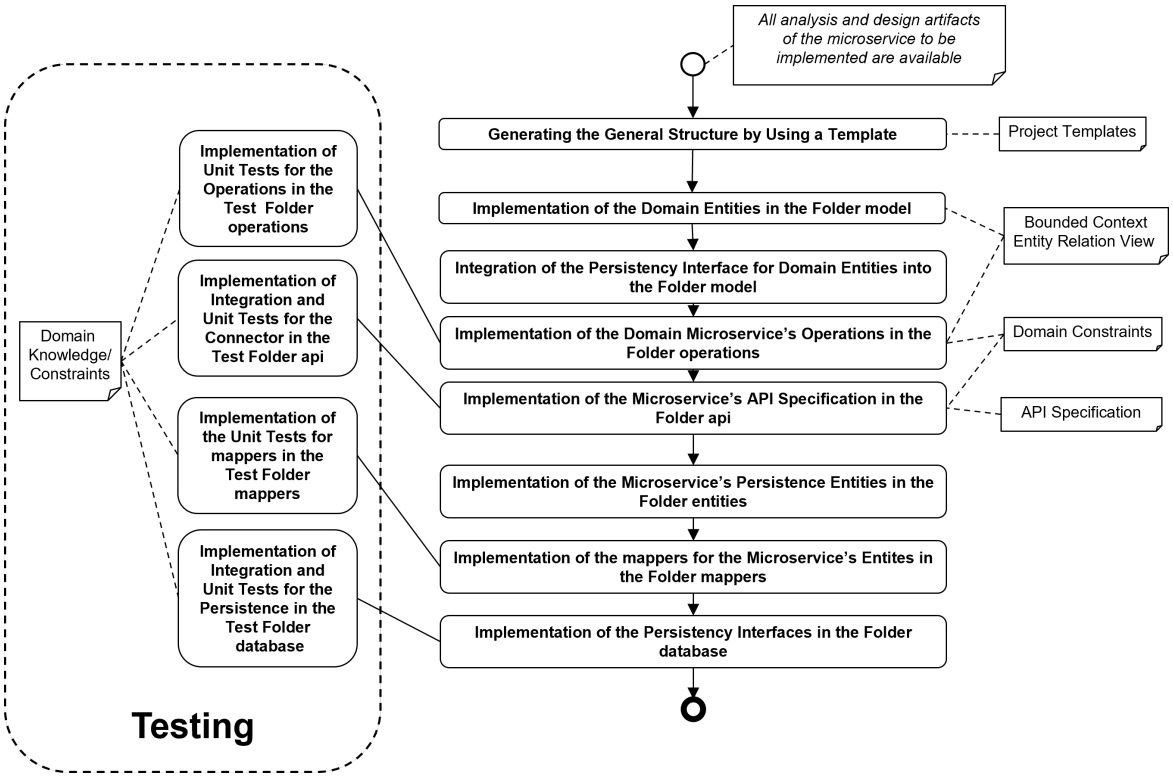


Abbildung 7.7: Prozess zur Implementierung der Domänen-Microservices

Die Constraints auf der Anwendungsebene bilden gemeinsam mit den Anwendungsfällen die Grundlage für die Ableitung der Testdaten und können analog zu den Domänen-Constraints betrachtet werden.

Vorgehen zur Implementierung und Testen eines Anwendungs-Microservices

Abbildung 7.9 greift den Prozess zur Implementierung auf. Hierbei wird durch die erste Aktion, die im vorigen Abschnitt beschriebene Mikroarchitektur durch die Verwendung einer Vorlage erzielt.

Der Implementierungs- und Testprozess eines Anwendungs-Microservices beginnt mit der Implementierung der anwendungsbezogenen Logik. Zur Ableitung wird das API-Diagramm zur Ableitung der wichtigsten Entitäten verwendet. Das Modell mit den Entitäten und Wertobjekte und deren Beziehungen bilden die Grundlage und werden daher aus dem zugehörigen API-Diagramm zuerst implementiert. Weiterhin werden die Microservice-Operationen implementiert. Die Logik hieraus ergibt sich aus der Anforderungsanalysephase und den Anwendungsfällen. Neben der Implementierung der Funktionalität werden ebenfalls die Unit-Tests abgeleitet und implementiert.

Nach der Implementierung der Logik erfolgt die Implementierung der API-Spezifikation. Hierbei

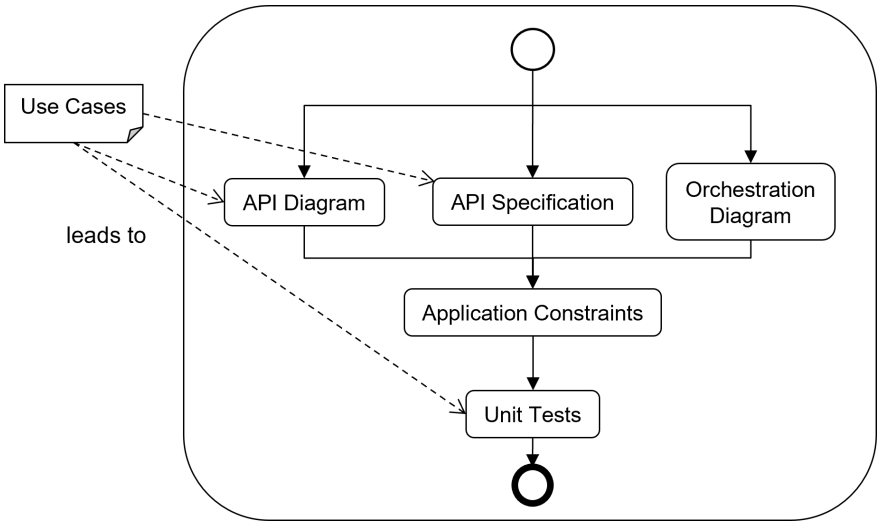


Abbildung 7.8: Ableitung der Unit-Tests im Rahmen der Anwendungs-Microservices

nimmt der API-Controller die API-Anfragen entgegen. Als zentrales Artefakt zur Implementierung wird die API-Spezifikation verwendet. Anhand von vorhandenen Werkzeugen (wie beispielsweise "oapi-codegen" [OAP-Ope] für die Programmiersprache Go) lassen sich aus der API-Spezifikation die Methodenrümpfe generieren. Für den Controller werden Unit-Tests geschrieben, wobei die erfassten Testdaten als Beispieldaten eingesetzt werden. Anhand der Orchestrierungs-Diagramme werden die von anderen Microservices oder externen Services (z.B. FROST-Server) benötigten Ressourcen zwischen AM-RentalManagement und DM-Car modelliert. Dadurch werden die benötigten Konnektoren abgeleitet. HTTP-Anfragen an externe Ressourcen werden durch Mapper auf das Format der für die Implementierung verwendeten Technologie abgebildet. Zu diesen Abbildungsvorschriften werden Unit-Tests und White-Box-Integrationstests implementiert. Sie ähneln einander insofern, als die zu testenden Funktionen mit Testdaten gefüttert werden und die erhaltenen mit den erwarteten Daten abgeglichen werden. Der Unterschied besteht darin, dass Unit-Tests die einzelnen Funktionen isoliert testen, während die Integrationstests einen größeren Teil des Systems erfordern.

7.2.3 Automatisierte Ausführung der Unit-Tests in einer CI/CD-Pipeline

Abbildung 7.2 zeigt die CI/CD-Pipeline im Überblick. Die Unit-Tests werden in der Pipeline nach dem Schritt "Build" ausgeführt. Schlagen die Tests fehl, dann wird in diesem Schritt die Pipeline beendet. Die Ergebnisse der Testausführung werden über die Pipeline zur Verfügung gestellt, so dass die Entwickler im Fehlerfall den Fehler basierend auf dem fehlgeschlagenen Test identifizieren können. Grundlegend werden die Unit-Tests zuvor lokal ausgeführt.

Bei den Anwendungs-Microservices werden für die Ausführung bestimmter Funktionalitäten ggf. die orchestrierten Domänen-Microservices benötigt. Für das Testen der Funktionalität innerhalb der

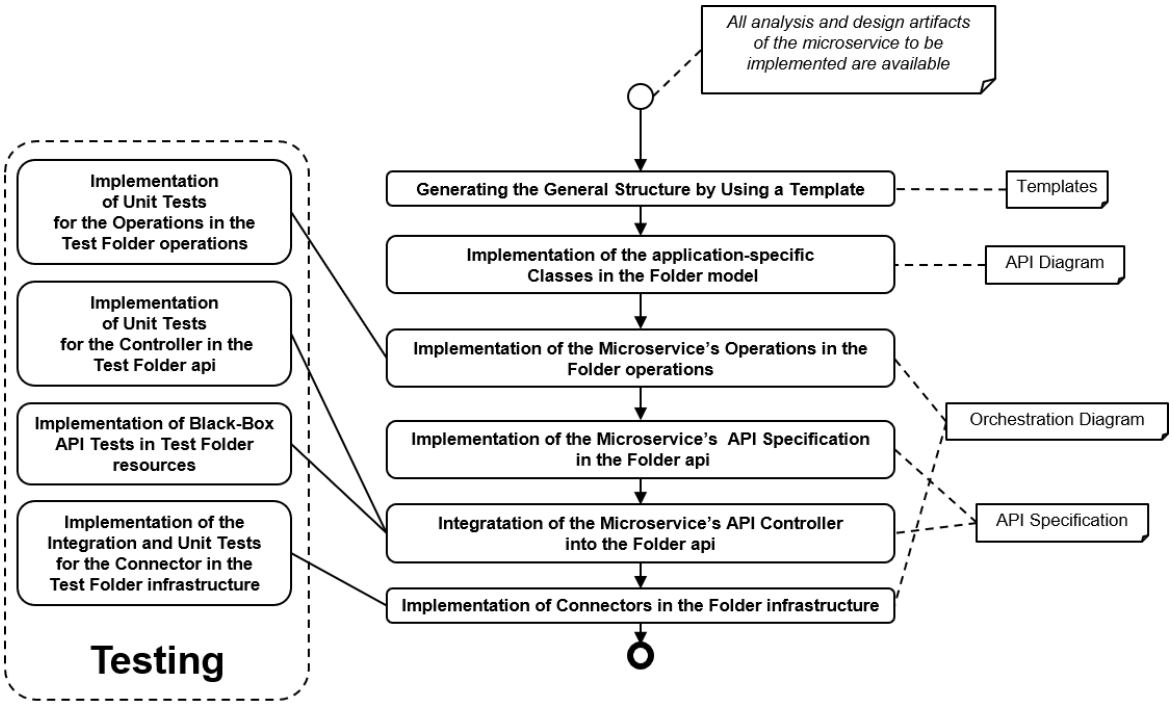


Abbildung 7.9: Prozess zur Implementierung der Anwendungs-Microservices

Services werden die benötigten Daten per Mock-Objekte bereitgestellt.

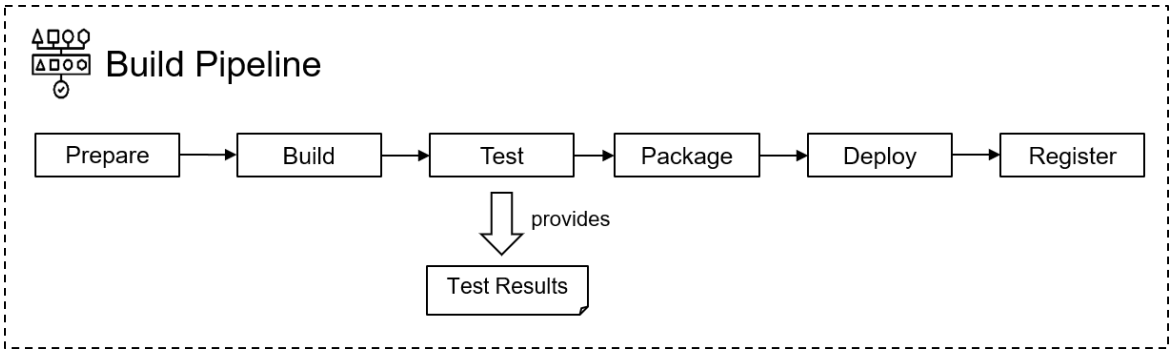


Abbildung 7.10: Ausführung der Unit-Tests in der CI/CD-Pipeline

Die Testergebnisse werden anschließend von der Pipeline gespeichert und sind für die Entwickler einsehbar. Im Falle eines Fehlschlags des Tests, wird der Entwickler automatisch benachrichtigt.

7.2.4 Konzeptionelle Unterschiede zwischen Domänen- und Anwendungslogik

Ein wichtiger Unterschied zwischen der Domänenlogik und der Anwendungslogik ist, dass die Anforderungen (wie die Anwendungsfälle) nicht bei der Domänenlogik betrachtet werden. Das hat

den Grund, dass die Domänen-Microservice anwendungsagnostisch zu betrachten sind. Daher wird die in den Anwendungsfällen spezifizierte Funktionalität nicht direkt betrachtet. Hingegen werden insbesondere die API-Spezifikation und die aufgestellten Domänen-Constraints herangezogen, um die Testfälle abzuleiten.

Im Gegensatz hierzu findet auf der Ebene der Anwendungslogik (und somit auf der Ebene der Anwendungs-Microservices) die Implementierung der Anwendungslogik statt. Diese greift bei der Ableitung der Testfälle auf die Anwendungsfälle zurück. Ein weiteres wichtiges Artefakt ist hier das Orchestrierungs-Diagramm, da bei diesem die Interaktion mit den zu orchestrierenden Microservices beschrieben wird. Bei der Umsetzung der Unit-Tests für die Anwendungs-Microservices werden die benötigten Daten der Domänen-Microservices durch Platzhalter (Mocks) ersetzt.

7.3 Testen der Integration zwischen Microservices

In diesem Abschnitt wird auf die Integration der beteiligten Microservices eingegangen. Die Konsumenten-getriebenen Vertragstests stellen hierbei das zentrale Element des Testkonzepts dar. Im ersten Schritt wird auf die systematische Ableitung des Vertrags zwischen dem bereitstellenden und dem konsumierenden Microservice eingegangen. Im nächsten Schritt wird die Aufstellung der Testfälle für den bereitstellenden und konsumierenden Microservice erläutert, bevor diese in die Implementierung überführt werden. Für die automatische Ausführung wird die Integration der Tests in eine CI/CD-Pipeline aufgezeigt.

7.3.1 Systematische Erstellung von Vertragstests zwischen Microservices

Die Schnittstellen zweier Microservices werden mit Konsumenten-getriebenen Vertragstests (engl. Consumer-Driven Contract Tests, CDC) isoliert zueinander getestet. Diese Tests bieten die Möglichkeit, die Integration zweier Microservices zu testen, ohne die Notwendigkeit, dass beide Microservices während der Tests verfügbar sind. Dies ermöglicht schnellere Testlaufzeiten und weniger Fehlbedienungen während des Tests [LM+19]. Ein Microservice ist hierbei der Konsument (engl. Consumer) und der andere der Anbieter (engl. Provider). Abbildung 7.11 zeigt zwei Vertragstests zwischen dem Anwendungs-Microservice A-FleetManagement und den zwei Domänen-Microservices D-Car und D-SensingDevice. Sowohl der Konsument als auch der Anbieter werden isoliert voneinander getestet.

Um die Interaktion zu testen, wird ein Vertrag aufgesetzt. Dieser Vertrag definiert die beabsichtigte Anfrage des Konsumenten und die erwartete Antwort des Anbieters. Zusätzlich kann der Vertrag weitere Informationen wie den Zustand des Anbieters enthalten. Dies ermöglicht es, spezielle Szenarien mit spezifischen Antworten zu testen. Der Vertrag wird bei CDC-Tests in der Regel von den

Konsumenten gesteuert. Jedoch haben die angebotenen APIs der darunterliegenden Microservices ebenfalls einen Einfluss auf die CDC-Tests. Insbesondere bei den Domänen-Microservices wird die Schnittstelle von der Fachlichkeit geprägt.

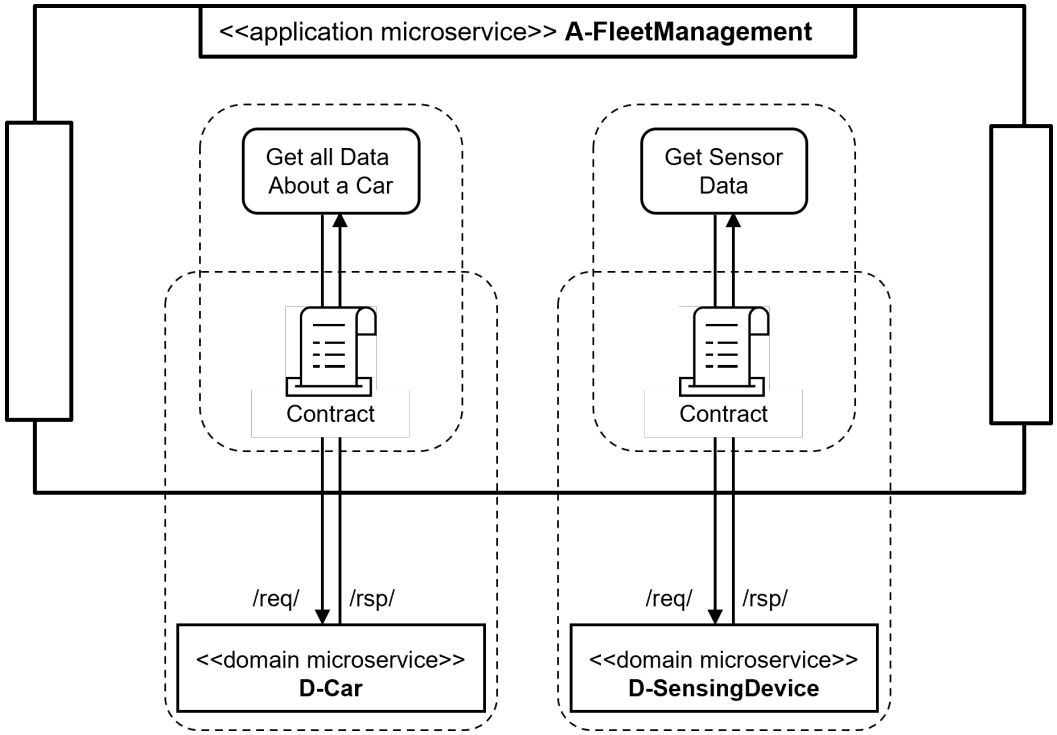


Abbildung 7.11: Verträge zwischen dem Anwendungs-Microservice und den zwei Domänen-Microservices

Die wichtigste Anforderung an das Testkonzept ist hierbei, wie die Ableitung der benötigten Testdaten aus den Artefakten des Microservice-Engineering-Ansatzes durchgeführt werden kann. Die wichtigsten Artefakte sind hierbei die Anwendungsfälle, die API-Spezifikation und dem zugehörigen Orchestrierungs-Diagramm. Aus diesen Artefakten wird bestimmt, was die Inhalte des Vertrages sind und wie die Verträge gestaltet werden. Anhand der Microservice-Operationen lassen sich die Inhalte der Verträge ableiten.

7.3.2 Inhalt und Aufstellung des Vertrags

Der Vertrag ist das zentrale Artefakt der CDC-Tests. Inhaltlich besteht der Vertrag aus drei Teilen. Der erste Teil besteht aus der beabsichtigte Anfrage und die erwartete Antwort, welche auf die Anfrage gesendet werden soll. Im zweiten Teil des Vertrags werden Regeln festgehalten. Diese Regeln geben an, wie die tatsächlichen Anfragen und Antworten mit den erwarteten Antworten abgeglichen werden. Der letzte Teil des Vertrags beinhaltet Metadaten, die zusätzliche Informationen über den

Vertrag enthalten, wie die Namen des konsumierenden und anbietenden Microservices, den Namen der Kommunikation sowie die Version des Vertrags. Die Version des Vertrags muss mit den verwendeten API-Versionen gepflegt werden. Ändert sich etwas an der Spezifikation, dann muss der Vertrag entsprechend aktualisiert werden und die Versionsnummer des Vertrags ändert sich analog zu der API-Spezifikation.

Die systematische Ableitung des Vertrags aus den aufgestellten Artefakten erfolgt durch den Engineering-Ansatz. Dadurch werden systematisch die benötigten Inhalte für die CDC-Tests aus [LM+19] hergestellt. Abbildung 7.12 verdeutlicht, welche Engineering-Artefakte zur Ableitung von welchem Teil des Vertrags benötigt werden.

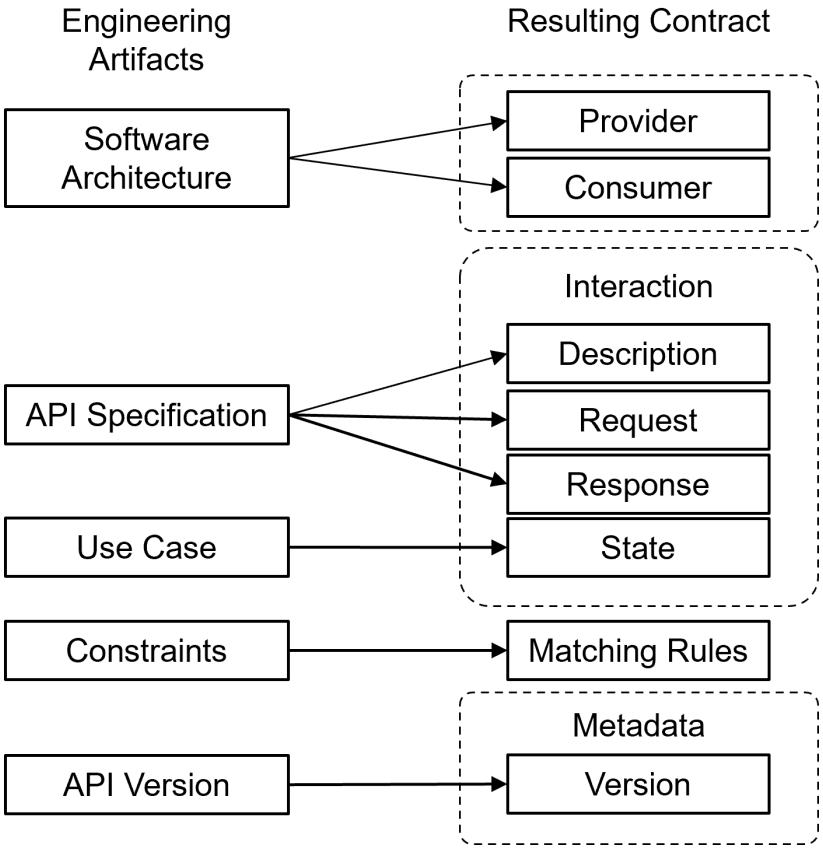


Abbildung 7.12: Ableitung der Vertragsinhalte aus den Artefakten des Engineering-Ansatzes

Aus der Software-Architektur lassen sich alle bereitstellenden und konsumierenden Microservices ableiten. Zwischen jedem konsumierenden und jedem bereitstellenden Microservice wird genau ein Vertrag spezifiziert. Der Vertrag zwischen zwei Microservices enthält abhängig von der API-Spezifikation mehrere Interaktionen. Eine Interaktion enthält eine Beschreibung sowie die Anfrage und die Antworten, die zwischen den beiden Microservices ausgetauscht werden. Jede Interaktion wird somit durch eine Kommunikation mit dem bereitstellenden und konsumierenden Microservice

bestimmt. Jede Interaktion eines Vertrages wird wiederum aufgeteilt in die Zustände des anbietenden Microservices (providerState) mit den dazugehörigen Anfragen und Antworten.

Die Anfrage (request) bestimmt, welche Daten benötigt werden. Diese Information wird aus der API-Spezifikation abgeleitet, welche das API-Diagramm und die Anwendungsfälle als Grundlage verwendet. Das Format der Felder wird durch die Randbedingungen der Anwendung und der Domäne bestimmt, die übereinstimmen sollten. Für jedes Feld sollte, analog zur API-Spezifikation, ein Beispielwert angegeben werden.

Die zugehörige Antwort (response) enthält die notwendigen Werte, die der konsumierenden Microservice für seine weitere Verarbeitung benötigt. Der Zustand des anbietenden Microservices ist wichtig, um den spezifischen Zustand des anbietenden Microservices für einen Test festzulegen. Ein solcher Zustand könnte zum Beispiel sein, dass der Anbieter ein bestimmtes spezifisches Fahrzeug in seiner Datenbank verwendet. Diese Aspekte lassen sich aus den zugehörigen Anwendungsfällen ableiten. In Anhang 10.2 befindet sich eine beispielhafte Ableitung eines Vertrags.

Geteilte Vertragsbestandteile Bei der Ableitung eines Vertrags ist es möglich, dass dieser Vertrag in Teilen mit einem anderen, bereits existierenden Vertrag identisch ist. Dies kann insbesondere dann der Fall sein, wenn für einen Domänen-Microservice und dessen API-Verträge mit zwei oder mehreren Anwendungs-Microservices erstellt werden. In diesem Fall würden die erstellten Verträge redundante Informationen über das Verhalten der Schnittstelle der Domänen-Microservices enthalten.

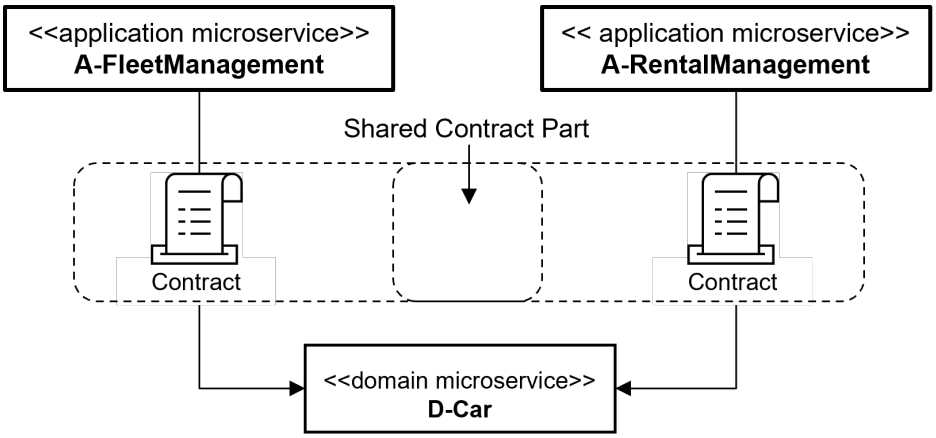


Abbildung 7.13: Gemeinsame Vertragsbestandteile

Abbildung 7.13 zeigt exemplarisch, wie ein gemeinsamer Vertragsteil auftreten kann. In diesem Beispiel testet der erste Vertrag die Kommunikation zwischen Microservice dem Domänen-Microservice Car und den Anwendungs-Microservices A-FleetManagement und A-ManagementOfRental. Der

gemeinsame Vertragsteil besteht aus Testfällen, die den Zugriff auf die gleichen Endpunkte der API-Spezifikation des Domänen-Microservices abdecken. Da diese Testfälle Teile von beiden Verträgen sein können, ist die Gefahr, dass redundante Testfälle entstehen. Der Umgang mit solchen gemeinsam genutzten Vertragsteilen hat einen Einfluss auf die Testqualität. Zum Beispiel sorgen die Redundanzen für einen höheren Wartungsaufwand, insbesondere dann, wenn sich die API-Spezifikation ändert. Das hat die Konsequenz, dass die Vertragsdefinitionen aller konsumierenden Services konsistent geändert werden müssen, was für Aufwand bei den Entwicklungsteams sorgt. Daher müssen die Verträge entsprechend verwaltet werden. Das trifft insbesondere dann zu, wenn CDC-Tests in einer CI/CD-Pipelines der jeweiligen Microservices integriert werden.

7.3.3 Methodisches Vorgehen zur Ableitung der CDC-Testfälle

Nach der Spezifikation des Vertrags müssen der bereitstellende und anbietende Microservice Tests gegen den Vertrag implementieren. Wie die Ableitung der Testfälle aus der spezifizierten API erfolgen kann, wird in Abbildung 7.14 gezeigt. Für jede im Vertrag definierte Interaktion werden Testfälle erstellt. Eine Interaktion entspricht einer Anfrage und einer erwarteten Anfrage. Bei dem bereitstellenden Microservice wird zuerst eine fehlerfreie Anfrage ausgeführt. Hierbei wird auf die korrekten Beispiele zurückgegriffen, welche bereits definiert wurden. Bei der Antwort des bereitstellenden Microservices wird das gelieferte Ergebnis mit dem erwarteten Ergebnis verglichen. Hierzu werden die Vergleichsregeln aus dem Vertrag verwendet.

Neben den positiven Testfällen werden ebenso negative Testfälle getestet. Negative Testfälle können auf verschiedene Arten generiert werden. Ein einfacher Fall ist beispielsweise die Adressierung einer nicht vorhandenen Ressource. Für die Ableitung der fehlerhaften Testdaten erfolgt zusätzlich aus den korrekten Testdaten. Ein Weg sind hier die aufgestellten Constraints, gegen die verstoßen wird. Ein solcher Test überprüft, ob die Constraints korrekt implementiert wurde. Dazu werden Test geschrieben, welche gegen die Invarianten verstoßen. Insbesondere Randfälle (z. B. bei Vergleichsoperatoren) sind hier zu berücksichtigen. Die Parameter für den Testfall am Rande werden von den aufgestellten Constraints ermittelt. Weiterhin sollten Tests für falsch übermittelte Parameter (wie beispielsweise eine nicht korrekte VIN) festgelegt werden. Ein weiterer Testpunkt inkludiert Parameter, welche nicht von dem Endpunkt vorgesehen sind. Diese müssen ebenfalls mitberücksichtigt werden.

Bei REST APIs ist ein weiterer wichtiger Punkt, dass die entsprechenden zurückgelieferten HTTP-Statuscodes überprüft werden. Die entsprechenden erwarteten Fehler sollen hierbei zu den erwarteten Statuscodes als Antwort führen. Ist eine Ressource beispielsweise nicht vorhanden, dann wird der Statuscode 404 als Antwort erwartet. Bei den API-Tests sollte hier für jeden erwarteten Statuscode (betrifft alle erwarteten 4xx-Statuscodes) ein entsprechender Testfall erfasst und umgesetzt werden.

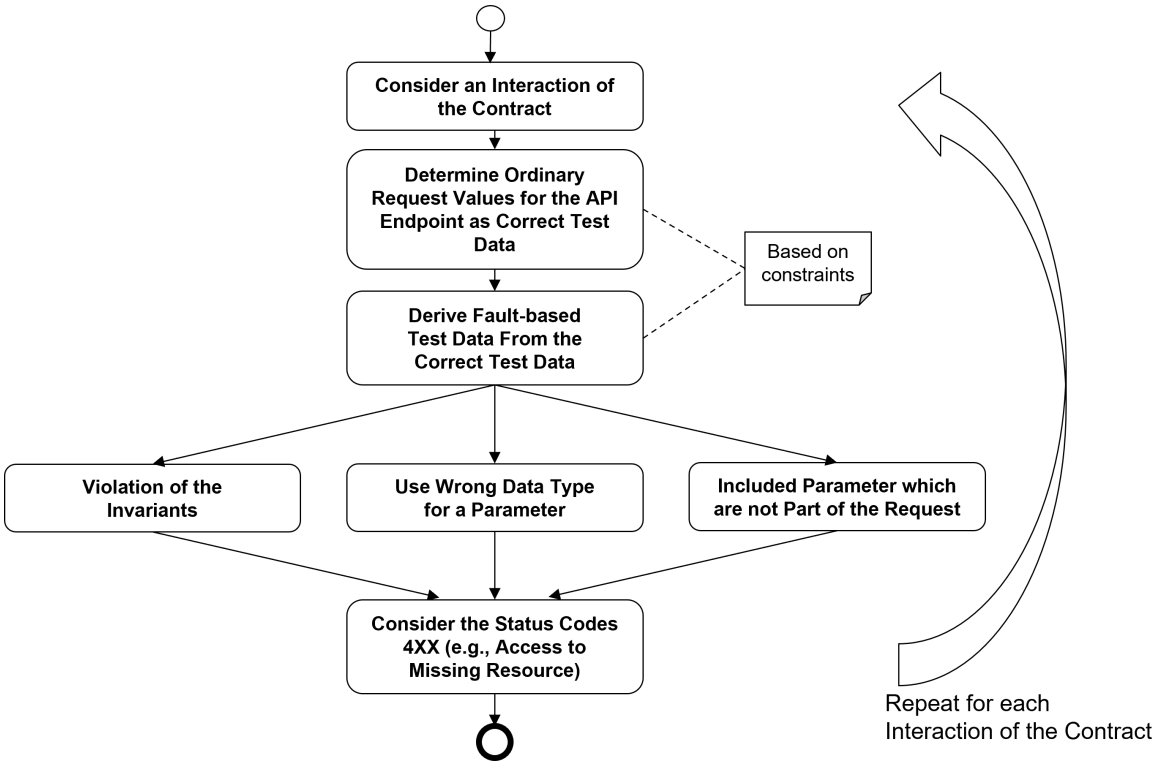


Abbildung 7.14: Ableitung der Testdaten für den bereitstellenden Microservice

Für jeden API-Aufruf zu einer Interaktion im Vertrag werden Schnittstellentests benötigt, damit sichergestellt werden kann, dass die Schnittstelle des Anwendungs-Microservice die Anforderungen erfüllt.

Bei dem konsumierenden Microservice wird analog vorgegangen. Anstatt der Anfragen werden nun die zurückgegebenen Antworten und deren Inhalte simuliert. Dadurch wird getestet, ob der konsumierende Microservice die korrekten Anfragen stellt und mit den erwarteten Antworten korrekt umgeht.

7.3.4 Implementierung der Schnittstelle und der Integrationstests

Eine manuelle Pflege der Verträge ist hierbei schwierig, da Änderungen ständig manuell gepflegt werden müssen. Die von Lehvä et al. [LM+19] verwendete und in der Branche gängige Methode ist die Verwendung des Werkzeugs Pact Broker. Der Pact Broker ist eine Anwendung für die gemeinsame Nutzung von verbrauchergesteuerten Verträgen und Verifizierungsergebnissen [Pac-Doc]. Pact unterstützt viele Programmiersprachen und bietet eine Komponente namens "Pact Broker" für den automatischen Austausch von Verträgen, sogenannten Pacts, die vom ConsumerTester während des Tests zwischen Konsument und Anbieter erzeugt werden. Dies ermöglicht eine vollautomatische

Testausführung und die Integration in eine CI/CD-Pipeline. Als Ergebnis werden die Verträge an den Anbieter im Repository weitergeleitet.

Der Teilprozess für die Implementierung der CDC-Tests beginnt mit zwei parallelen Schritten. Der erste Schritt ist die Implementierung des ConsumerTesters, der beim konsumierenden Microservice implementiert wird. Der Vertrag wird verwendet, um einen Mock des anbietenden Microservice zu erstellen. Der konkrete anbietende Microservice wird nicht benötigt. Der zweite Schritt bei der Implementierung ist die Bereitstellung des sogenannten "Pact Broker". Der "Pact Broker" ist eine Anwendung, welcher den Austausch und die Versionierung von Verträgen zwischen dem konsumierenden und anbietenden Microservice ermöglicht. Der ConsumerTester stellt bei der Ausführung der Tests den neuesten Vertrag bereit. Der ProviderTester wiederum verwendet diesen Vertrag, um den anbietenden Microservice zu testen. Die Implementierung des ProviderTester erfolgt daher erst im Anschluss an den ConsumerTester.

Auf die Implementierung des ProviderTesters folgt die Integration in die Teststufen der Pipelines in den verschiedenen Repositories, um eine automatische Ausführung der Consumer- und Provider-Tests bei Änderungen in der Implementierung zu ermöglichen.

7.3.5 Automatisierte Ausführung der Integrationstests in der CI/CD-Pipeline

Um die Tests bei Änderungen an den betroffenen Microservices automatisch auszuführen, werden die Tests innerhalb der CI/CD-Pipeline ausgeführt. Die ConsumerTester und ProviderTester erhalten jeweils eigene Schritte in der Pipeline. Abbildung 7.15 zeigt die Pipeline mit den ergänzten Schritten für den bereitstellenden Microservice.

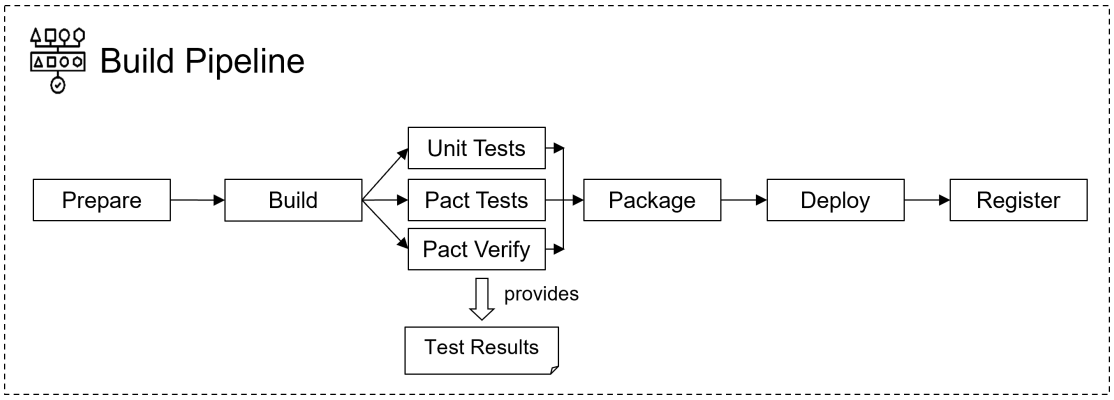


Abbildung 7.15: Ausführung der CDC-Tests in der CI/CD-Pipeline

Die CDC-Tests benötigen für die Ausführung in der Pipeline nur den aktuellen Microservice. Hierbei wird der konsumierende und der bereitstellende Microservice unabhängig voneinander getestet. Die Durchführung der Tests beginnt mit dem konsumierenden Microservice und ist in drei Schritten

aufgeteilt. Der erste Schritt führt die Verbrauchertests aus. Anschließend werden diese über sogenannte PACT-Dateien veröffentlicht.

Anschließend kann der bereitstellende Microservice getestet werden. Hier besteht die Pipeline aus zwei Schritten. Im ersten Schritt werden die Provider-Tests ausgeführt. Die Testergebnisse werden über den PactBroker veröffentlicht. Im zweiten Schritt wird analog zum Test des konsumierenden Microservice geprüft, ob der Microservice bei der Bereitstellung möglicherweise die Kommunikation mit einem anderen Microservice unterbrechen könnte. Falls dies der Fall ist, wird die Bereitstellung verhindert.

7.3.6 Zusammenhang zu den Ende-zu-Ende-Tests

CDC-Tests sind Teil des Testansatzes, der die Entwicklungsreihenfolge von Unit-Tests, Integrationstests und E2E-Tests festlegt. Der Ansatz sieht eine Aufteilung der CDC-Testentwicklung entlang der Schichtenarchitektur vor. Zunächst werden die CDC-Tests für die Schichten der Infrastruktur und der Domäne implementiert. Das bedeutet, dass die API-Konformität zwischen den Domänen-Microservices und den entsprechenden System-APIs durch die Definition von Constraints (basierend auf dem API-Diagramm), die Ableitung von Testfällen (basierend auf den Anwendungsfällen) und die Implementierung dieser Tests getestet werden sollte. Der Grund dafür ist, dass CDC-Tests oft auf Constraints basieren und sich die Constraints auf der Anwendungs- und Präsentationsebene häufig zu den Constraints auf der Infrastruktur- und Domänenebene ähneln.

Der zweite Teil befasst sich mit dem Testen der Anwendungs-Microservices. Auch hier sind die wichtigsten Artefakte für die Ableitung der Testfälle die Anwendungsfälle. Die Tests der Anwendungs-Microservices werden in Orchestrierungstests und Verarbeitungstests unterteilt. Die Orchestrierungstests stellen sicher, dass die Anwendungs-Microservices die erwarteten Daten erhalten, während die Verarbeitungstests die korrekte Implementierung der Anwendungslogik überprüfen. Schließlich wird geprüft, ob der Anwendungs-Microservices überhaupt die erwarteten Daten empfängt und ob die Ausgabe den Erwartungen und den Constraints entspricht.

Der dritte Teil befasst sich mit den Tests der Präsentationslogik sowie mit den Ende-zu-Ende- Tests (E2E-Tests) im Allgemeinen. Ein Experience-Microservice fungiert als Backend For Frontend (BFF) für die Anwendung, indem sie alle API-Endpunkte der zugrundeliegenden Anwendungs-Microservices so bereitstellt, dass diese von einer bestimmten Benutzerschnittstelle konsumiert werden kann.

7.4 Ende-zu-Ende-Tests der Anwendung

Die Ende-zu-Ende-Tests prüfen die Akzeptanzkriterien der Anwendung. Das bedeutet, dass die aufgestellten Anforderungen erfüllt werden und die Software das Verhalten aufzeigt, welches der Benutzer erwartet. Eine wichtige Eigenschaft ist die Ableitung der Ende-zu-Ende-Tests auf den im Engineering-Ansatz erstellten Artefakte. Bei dem in dieser Arbeit vorgestellten Ansatz wird ein separates Repository für die Implementierung der Ende-zu-Ende-Tests verwendet. Hierfür wird der Ansatz aus [RG15] adaptiert. Der Vorteil eines solchen Test-Repositories ist, dass eine geringere Wartung der Tests notwendig ist. Das Test-Repository kann nicht auf die interne Implementierung der Microservices zugreifen, was bedeutet, dass die gesamte Anwendung als Blackbox behandelt werden muss. Bevor die Ende-zu-Ende-Tests ausgeführt werden können, muss jeder Microservice implementiert werden und lauffähig sein. Die beiden wichtigsten Fragen, die bei Ende-zu-Ende-Tests zu beantworten sind, sind, was und wie getestet wird. Die Ende-zu-Ende-Tests sollten nicht dazu dienen, eine hohe Abdeckung für alle Pfade in einer Anwendung zu erreichen, sondern sie sollten Beispiele für das Verhalten der Software beschreiben [Sm14].

7.4.1 Ableitung der Ende-zu-Ende-Tests aus den Anwendungsfällen

Die Akzeptanztests werden in in einer ähnlichen Weise, wie es die verhaltensgetriebene Softwareentwicklung (Behavior-Driven Development, BDD) vorsieht, abgeleitet. Dort wird mit der Sprache Gherkin die Anforderungen als Features beschrieben. Die Features enthalten Szenarien, die aus mehreren Schritten bestehen. Diese Schritte werden als Schrittdefinitionen (engl. step definitions) implementiert und werden (beispielsweise mit Cucumber) gegen das System ausgeführt [WH12]. Wenn ein Schritt fehlschlägt, werden die restlichen zum Szenario gehörenden Schritte übersprungen und der Test gilt als fehlgeschlagen. Werden alle Schritte erfolgreich durchlaufen, dann gilt der Test als bestanden. Bei der Microservice-basierten Architektur bedeutet dies, dass alle beteiligten Microservices korrekt zusammen agieren müssen.

Da bei dem vorgestellten Entwicklungsansatz keine Gherkin-Features zur Anforderungsspezifikation erstellt werden, kann dieses Vorgehen nicht direkt übernommen werden. Das Testkonzept für die Ende-zu-Ende-Tests sieht jedoch ein ähnliches Vorgehen vor. Ein Verfahren zu Formalisierung von Gherkin als UML-Anwendungsfalldiagramm wird in der Literatur bereits behandelt [GR+17]. Bei dem hier verfolgten Ansatz wird aus den modellierten Anwendungsfällen die Testfälle abgeleitet. Abbildung 7.16 skizziert die Ableitung der Testfälle aus einem Anwendungsfall. Hierbei wird ein konkreter Anwendungsfall bei der Umsetzung durch eine Test-Datei repräsentiert.

Der Ausgangszustand, zu dem das System vor der Ausführung des Tests gebracht werden muss, ist in der Vorbedingung beschrieben. Im Vergleich zu Gherkin wird hier der Schritt "Given" für den

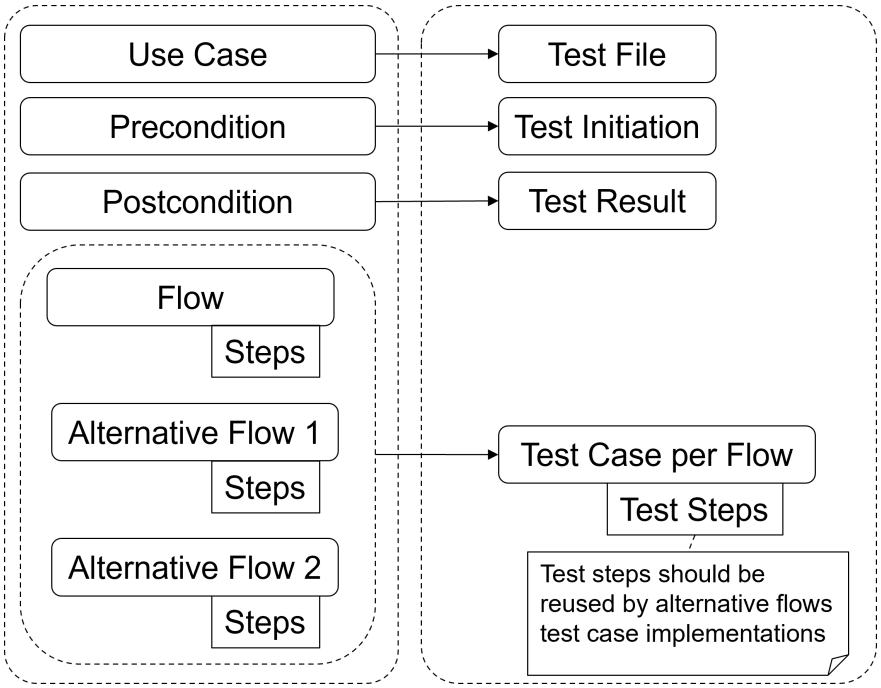


Abbildung 7.16: Ableitung der Ende-zu-Ende-Tests aus den Anwendungsfällen

Ausgangszustand verwendet. Die durchzuführenden Aktionen, werden durch die Flüsse beschrieben. In der Analogie zu Gherkin handelt es sich um den Schritt "When" der die Aktion beschreibt, die der Benutzer ausführt um einen Zustandsübergang zu erreichen. Hierbei besteht ein Fluss aus verschiedenen Schritten, die jeweils in einzelne Testschritte überführt werden müssen. Aus jedem Schritt, der einer Benutzerinteraktion zugehört, wird daher ein entsprechender Testschritt abgeleitet. Um die Wiederverwendbarkeit zu erhöhen, bzw. zu ermöglichen, werden für die einzelnen Schritte eigene Methoden definiert. Dadurch können bereits implementierte Schritte bei weiteren Flüssen (insbesondere bei alternativen Flüssen) wiederverwendet werden. Das Endresultat des Tests muss anschließend mit der Nachbedingung übereinstimmen. In diesem Fall ist der Test erfolgreich durchgelaufen, andernfalls schlägt der Test fehl. Bei Gherkin wird der Schritt "Then" verwendet, um das gewünschte Ergebnisse zu beschreiben. Bei den Anwendungsfällen ergibt die Nachbedingung (engl. postcondition) den gewünschten Systemzustand an. Ein konkretes Beispiel der systematischen Umsetzung mit den Artefakten befindet sich im Anhang 10.3.

Struktur des Test-Repositories

Wie bereits erwähnt, wird für die Umsetzung der Ende-zu-Ende-Tests ein eigenes Repository verwendet, angelehnt an [RG15]. Ein weiterer Vorteil für die Nutzung eines Test-Repository ist, dass die Tests in einer anderen Programmiersprache als die Benutzerschnittstelle geschrieben werden kann.

In Listing 7.2 ist die Mikroarchitektur des Test-Repositories gegeben. Das Repository sortiert die verschiedenen Tests der einzelnen Anwendungsfälle in die Capabilities ein. Unter den flowDefinitons werden die vom Anwendungsfall ausgehende Schritte (analog zu Schrittdefinitionen mit Gherkin) definiert.

```
|- test
|  |- e2e
|  |  |- capability1
|  |  |  |- use_case1
|  |  |  |- use_case2
|  |  |- capability2
|  |- pageObjects
|  |  |- navigation.page.ts
|  |- flowDefinitions
|  |  |- flow.ts
```

Listing 7.2: Mikroarchitektur des Test-Repositorys

7.4.2 Richtlinien zur Gestaltung von Ende-zu-Ende-Tests

Die Qualität von Tests muss unter zwei Gesichtspunkten betrachtet werden. Zum einen von der Testspezifikation, d.h. die Testschritte müssen sich aus der Anforderungsanalyse ableiten lassen, sowie die Testimplementierung, d.h. die Umsetzung der abgeleiteten Testschritte. Um einen hohen Wartungsaufwand zu vermeiden, sollten diese imperativ geschrieben werden. Dadurch werden UI-spezifische oder andere irrelevante Informationen vermieden.

Tests sollen den Entwicklern eine Rückmeldung geben, ob eine Änderung die Anwendung beeinflusst oder nicht. Wenn die Entwickler lange warten müssen, bis die Testausführung abgeschlossen ist, beeinträchtigt dies Produktivität. Außerdem muss nicht jeder Randfall durch einen Ende-zu-Ende-Test verifiziert werden, diese sollten nach Wynne et al. mit Unit-Tests getestet werden [WH12].

Automatisierte Ende-zu-Ende Tests können erfolgreich sein, aber der gleiche Test kann bei der nächsten Testausführung ohne ersichtlichen Grund fehlschlagen. Wettlaufsituationen (engl. race conditions) können hierfür der Auslöser sein. Gerade in Web-Anwendungen laufen viele Anfragen asynchron ab, so dass die Reihenfolge, in der die Aufrufe beantwortet werden, nicht im Voraus bekannt ist. Eine Lösung hierfür ist die Verwendung von Wartezeiten mit fester Länge. Dies ist jedoch keine geeignete Lösung, da sich dadurch die Testausführungszeit erhöht und auf der anderen Seite nur die Möglichkeit einer Wettlaufsituation verringert wird. Stattdessen sollten bedingte Wartezeiten verwendet werden [WH12]. Ein ähnliches Problem kann auftreten, wenn Tests einen persistenten Zustand ändern, diesen aber nicht zurücksetzen. In diesem Fall würde der Erfolg eines Tests von der

Ausführungsreihenfolge abhängen. Dieser Nebeneffekt kann zu falschen negativen Testergebnissen führen, d. h. die Funktionalität funktioniert, aber der Test schlägt fehl. Um dies zu verhindern, sollte ein solcher Zustand immer zurückgesetzt werden oder von einem gezielten Ausgangszustand ausgegangen werden.

7.4.3 Automatisierte Ausführung der Ende-zu-Ende-Tests mithilfe einer CI/CD-Pipeline

Werden an der Anwendung Änderungen vorgenommen (beispielsweise Änderungen an einem der Microservices), dann muss überprüft werden, ob die gesamte Anwendung noch die gewünschte Funktion fehlerfrei erbringt. Durch die automatische Ausführung der Ende-zu-Ende-Tests in einer CI/CD-Pipeline und einer Testumgebung können die Anforderungen überprüft werden. Eine Ausführung der Tests von der gesamten Anwendung erfordert, dass alle Microservices für die Tests bereitstehen. Abbildung 7.17 zeigt, wie ein Commit auf einen Microservice der Anwendung alle beteiligten Microservices der Anwendung verteilt und in einer Testumgebung bereitstellt.

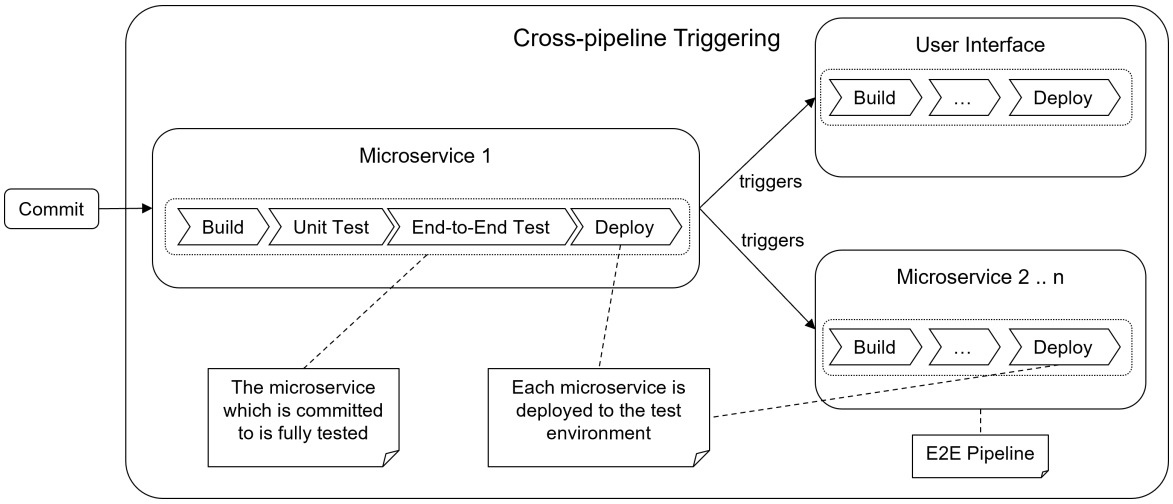


Abbildung 7.17: Ausführung der verschiedenen Pipelines für die Ende-zu-Ende-Tests

Hierzu wird eine sogenannte Multi-Stage-Pipeline [Git-Doc] verwendet, welche vor der Ausführung der Ende-zu-Ende-Tests alle benötigten Bestandteile der Anwendung in einer Testumgebung bereitstellt. Das beinhaltet die verschiedenen Microservice-Typen, wie auch die Benutzerschnittstelle. Zudem müssen auch die benötigten Datenbanken für die Tests initialisiert und bereitgestellt werden. Der Microservice, der geändert wurde, erfährt umfangreiche Tests. Alle weiteren benötigten Microservices werden für die Testausführung ebenfalls mit einem definierten Datenbankzustand bereitgestellt. Nachdem die Bestandteile der Anwendung bereitstehen, wird anhand der E2E-Testanwendung die Ende-zu-Ende-Tests ausgeführt.

7.5 Zusammenfassung

Ziel des Kapitels war es, die Implementierung und das Testen im Engineering-Ansatz vorzusehen, so dass die Problemstellung P4 (siehe Abschnitt 1.4) gelöst werden kann. Das Kapitel entspricht dem dritten Forschungsbeitrag (B3) zum Implementierungs- und Testkonzept, um eine qualitätsgesichertes Software-System zu erhalten. Vorgestellt wurde eine Systematik, die es erlaubt, das Microservice-basierte Software-System zu implementieren und systematisch basierend auf den bisher erstellten Artefakten zu testen. Die Basis hierfür wird durch den in Kapitel 4 vorgestellten systematischen Prozess zur Entwicklung von fortgeschrittenen Web-Anwendungen und den verschiedenen Artefakten gelegt. Die verschiedenen Artefakte (wie das API-Diagramm und die Constraints) werden gezielt für die Ableitung der funktionalen Tests verwendet. In Abschnitt 7.1 wurde die Einordnung der benötigten Artefakten zu den Tests geklärt. Diese liefert zusammen mit dem Testkonzept die Systematik zur Ableitung der verschiedenen Testarten.

Zu Beginn wird beim Implementierungs- und Testkonzept in Abschnitt 7.2 mit den Unit-Tests begonnen, um die internen Funktionalitäten eines Microservices sicherzustellen. Über die Integrationstests in Abschnitt 7.3 wird die Funktionalität der Microservices über die Schnittstelle als Blackbox getestet. Den Abschluss des Testkonzepts bilden die Ende-zu-Ende-Tests in Abschnitt 7.4. Für die Ableitung der verschiedenen Tests wird hierbei jeweils ein Vorgehen erläutert, welches die gezielte Aufstellung der Tests ermöglicht. Das Gesamtkonzept liefert ein Vorgehen, das der Entwickler gezielt für die Ableitung der Tests verwenden kann. Dadurch wird geklärt welche Teile des Softwaresystems wie zu testen sind und die Testfälle werden nicht nach dem Zufall generiert. Zu jeder Testarten wird die Integration in eine CI/CD-Pipeline unterstützt. Dadurch ist die kontinuierliche Entwicklung anhand eines CI/CD-Prozesses abgedeckt.

8 Validierung der Beiträge

Die domänengetriebene Entwicklung von fortgeschrittenen Web-Anwendungen ermöglicht eine systematische Entwicklung von Softwaresystemen unter Berücksichtigung der Domäneninhalte und deren Extraktion in separate Microservices, um wiederverwendbare Softwarebausteine zu generieren. Die Systematik des Entwicklungsansatzes unterstützt hierbei alle Phasen der Softwareentwicklung, angefangen bei der Erhebung der Anforderungen in der Analyse, der Überführung dieser in den Entwurf, bei der Etablierung der Software-Architektur und der benötigten Schnittstellen, sowie bei der Überführung in die Implementierung und dem Testen des Softwaresystems.

Um die Eignung des Ansatzes zu zeigen, werden verschiedene Validierungsarten der empirischen Validierung durchgeführt. Die Basis liefern hierfür Experimente, welche als Ziel die Tragfähigkeit der vorgeschlagenen Konzepte zur Entwicklung der Softwarelösungen verfolgen. Die Experimente selbst werden unter anderem von Studierenden der Informatik ausgeführt. Die erzielten Ergebnisse werden begleitend analysiert und zur Verbesserung des systematischen Ansatzes eingesetzt.

In Abschnitt 8.1 werden die verschiedenen Validierungsarten der empirischen Validierung betrachtet. Diese beschreiben, wie die Validierung durchgeführt werden muss, um die Eignung der Beiträge zu zeigen. Im Kontext dieser Arbeit ergeben sich die folgenden Punkte:

- (1) Vergleich der Umsetzung ohne vorgegebene Systematik
- (2) Bewertung der Forschungsbeiträge in Bezug auf den Anforderungskatalog
- (3) Durchführung der Systematik an einem durchgängigen Beispiel
- (4) Einsatz der Systematik an einem konkreten Industriekontext mit einem Industriepartner

In Abschnitt 8.2 werden verschiedene Einschränkungen und Gefahren, die sich für die Validität ergeben können, aufgegriffen. In den nachfolgenden Abschnitten wird basierend auf den Validierungsarten die Validierung durchgeführt. Diese beginnt in Abschnitt 8.3 mit der eigentlichen Machbarkeit der Forschungsbeiträge in Bezug auf den Anforderungskatalog (2). Dort wird geprüft, ob die Ergebnisse, die mit der Systematik erzielt werden, den Anforderungen gemäß Abschnitt 3.1 gerecht werden.

Die Eignung des Ansatzes gemäß der Typ-1-Validierung wird in Abschnitt 8.4 anhand eines durchgängigen Beispiels durchgeführt (1)(3). Verglichen werden hierbei die erzielten Ergebnisse mit den Ergebnissen einer Kontrollgruppe, die den systematischen Ansatz nicht vorliegen haben. Anschließend folgt in Abschnitt 8.5 die Bewertung der Systematik anhand eines Industrieprojekts (4). Dies erfolgt,

indem die Systematik bei der Entwicklung einer Anwendung gemeinsam mit dem Kooperationspartner im betrachteten Industrieprojekt eingesetzt und bewertet wird. Abschließend wird in Abschnitt 8.6 eine Zusammenfassung und ein Fazit der Validierung gegeben.

8.1 Empirische Validierung

Durdik [Du16] verfolgt vier verschiedene empirische Validierungsarten, welche ebenfalls in dieser Arbeit aufgegriffen werden. Typ 0 beschreibt die generelle Machbarkeit des Ansatzes, während Typ I auf die Eignung des Ansatzes anhand eines durchgängigen Beispiels eingeht. Typ II zeigt die Anwendbarkeit im Industriekontext und der Validierungstyp III sieht ein Kosten/Nutzen-Vergleich vor. Im Folgenden werden die einzelnen Typen im Kontext der Arbeit näher erläutert.

Typ 0 (Machbarkeit)

Die Typ 0-Validierung der Machbarkeit ist eine einfache Validierung, welche basierend auf einfachen und fiktiven Beispielen die Machbarkeit der Ansätze validiert. Diese Validierung erfolgt in dieser Forschungsarbeit durch den exemplarischen Vergleich der Soll-Situation (durch den Einsatz des systematischen, domänengetriebenen Ansatzes zur Entwicklung von fortgeschrittenen Web-Anwendungen) mit einer (hypothetischen) Ist-Situation, die ohne die Anwendung des Einsatzes entsteht. Konkret bedeutet dies, dass die erzielten (Teil-)Ergebnisse, die den Handlungsbedarf abdecken mit den aufgestellten Anforderungen aus Abschnitt 3.1 verglichen werden. Die Ist-Situation lässt sich hierbei aus dem aktuellen Forschungsstand und dem Vergleich der Entwicklungsarbeit ohne den Einsatz der Systematik aufstellen. Anschließend werden der vorgestellte Ansatz und die erzielten Resultate diesem gegenüber gestellt. Das Verständnis in Bezug auf die Systematik kann mit Hilfe von Umfragen erhalten werden [Gi18, Du16]. Hierbei wird erfasst, inwieweit die Systematik die Entwicklung vereinfacht. Weiterhin lässt sich ebenfalls ein Vergleich mit den Ansätzen aus der bestehenden Literatur herstellen, indem die Ergebnisse auf dessen Basis durchgeführt und bewertet werden. Damit die Ergebnisse repräsentativ sind, sollte die Personengruppe aus Entwicklern mit etwa gleichem Kenntnisstand bestehen. Diese Validierungsart ist im Vergleich zu den anderen Typen der empirischen Validierung einfach umzusetzen, da der Aufwand der Validierung überschaubar ist [Gi18, Du16]. Abbildung 8.1 zeigt den Zusammenhang des Aufwands der verschiedenen Typen, der Validierung und der externen Validität.

Typ I (Eignung)

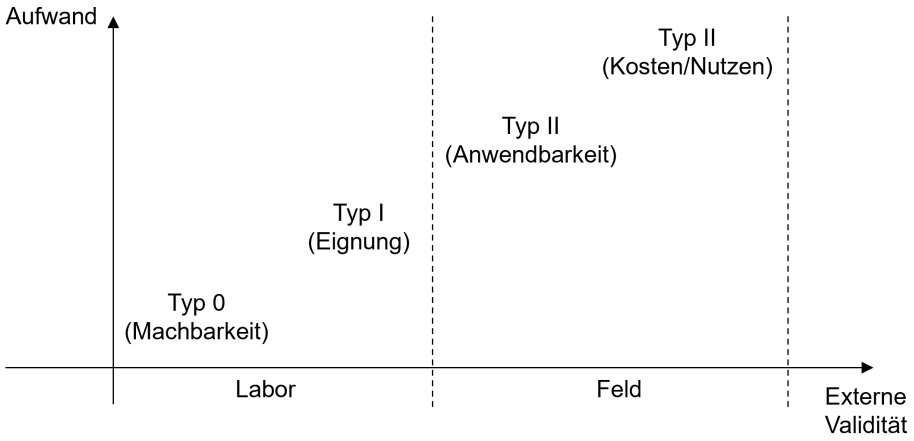


Abbildung 8.1: Validierungsarten der empirischen Validierung und dem Aufwand nach [Gi18]

Im Kontrast zur Typ 0-Validierung, wird die Eignung der aufgestellten Systematik durch ein praxisnahes und durchgängiges Beispiel überprüft. Dieses Beispiel kann nach [Du16] einen fiktiven Charakter besitzen. Ein realitätsnahes Beispiel eignet sich jedoch besser, da hierbei die externe Validität höher ist. Diese Validierung lässt sich daher wieder im Labor erfassen. Im Kontext der Arbeit wird eine realistische Situation nachgestellt, indem verschiedene Entwicklerteams die Forschungsbeiträge nutzen und deren Eignung zur Erreichung des Zieles anhand der Entwicklung einer fortgeschrittenen Web-Anwendung zeigen. Die verschiedenen Konzepte und Bestandteile des systematischen Ansatzes werden hierbei aufgegriffen und genutzt. In die Bewertung fließen die Ergebnisse ein, die durch den Ansatz entstehen. Diese werden analog zu Durek [Du16] mit Ergebnissen verglichen, die ohne die Verwendung des Ansatzes entstanden sind. Ein wichtiger Punkt in dieser Forschungsarbeit ist hierbei, dass die verschiedenen Artefakte strukturerhaltend ineinander überführt werden können. Für die Erprobung werden insbesondere Studierende der Informatik herangezogen, die sich für die Validierung solcher Ansätze eigenen [Ti00].

Die Auswertung und Bewertung der Ergebnisse erfolgt durch eine Beobachtung der Ergebnisse, Interviews und Rückfragen an die Entwickler. Die Interviews selbst werden durch einen zuvor präparierten Fragebogen geleitet. Insgesamt ist der Aufwand der Typ 1-Validierung etwas höher (siehe Abbildung 8.1 als der Aufwand der Typ 1-Validierung, die Aussagekraft bezüglich der externen Validität ist aber entsprechend höher.

Typ II (Anwendbarkeit)

Die Validierung des Typs 2 evaluiert die Anwendbarkeit der domänengetriebenen Entwicklung von fortgeschrittenen Web-Anwendungen, indem die angestrebte Zielgruppe den Ansatz erproben

und in einem realen Projektkontext anwenden. Die Kontrollgruppen sind hierbei Software-Architekten und Entwickler, sowie Studierende der Informatik. Der reale Projektkontext wird von einem Industriepartner gestellt, der gemeinsam mit Kunden an Softwarelösungen arbeitet. Durch den Praxisbezug kann die Anwendbarkeit des systematischen Ansatzes direkt am realen Beispiel erprobt werden. Dieser Bezug zum realen Projekt sorgt ebenfalls für eine hohe externe Validität, wie es entsprechend in Abbildung 8.1 dargestellt ist. Die zu entwickelnde Anwendung im Projektkontext ist eine Web-Anwendung, die die entsprechend der Ziel-Architektur aus Abschnitt 4.1 verwendet. Nach [Gi18] und [Hi21] ist ein Bewertungsbogen geeignet, um die Ergebnisse der Umsetzung festhalten, so dass ein repräsentatives Ergebnis zur Anwendbarkeit gewonnen werden kann. Zudem kann eine Analyse (Ist-Soll-Vergleich) der entstandenen Artefakte einen weiteren Aufschluss auf die Qualität der Ergebnisse in Bezug auf die Systematik geben.

Typ III (Kosten/Nutzen)

Bei diesem Typ der Validierung wird ein Vergleich zwischen aktuell eingesetzten Entwicklungsvorhaben mit dem in der Forschungsarbeit betrachteten Ansatz anhand von Kosten und Nutzen durchgeführt. Für eine aussagekräftige Validierung, wird somit die Entwicklung des gleichen Softwaresystems einmal mit und einmal ohne die Systematik auf dem Feld benötigt. Der Vergleich der erzielten Ergebnisse unter der Berücksichtigung der entstandenen Kosten ermöglichen eine Bewertung der Vor- und Nachteile des vorgeschlagenen Entwicklungsansatzes. Weiterhin liefern wiederverwendbare Microservices insbesondere Vorteile bei der Entwicklung weiterer Softwaresysteme, weshalb dies bei dem Kosten/Nutzen-Vergleich ebenfalls berücksichtigt werden müssten. Unter diesen Gegebenheiten und mit der Einschätzung von Giessler [Gi18], ist diese Art der Validierung nur mit einem hohen Aufwand und hohen Kosten verbunden und wird daher nicht oft eingesetzt [He08]. Aus den genannten Gründen wird die Typ III-Validierung nicht in dieser Forschungsarbeit durchgeführt.

8.2 Einschränkungen und Bedrohungen der Validität

Keine Validierungsart ist optimal und unterliegt verschiedenen Einschränkungen und Bedrohungen, welche Auswirkungen auf die Bewertung der erzielten Forschungsergebnisse dieser Arbeit haben. Um diese Einschränkungen und Bedrohungen zu identifizieren, werden bekannte Klassifizierungen der Validität herangezogen und die verschiedenen Validierungsarten und gemäß den erzielten Ergebnissen untersucht. Die Validität der Forschung ist gegeben, wenn die gemessenen Ergebnisse mit den gewünschten Forschungsparametern übereinstimmen. Wohlin et al. stellen [WR+12] hierfür eine Klassifizierung der verschiedenen Arten der Validierung und deren Bedrohungen auf. Es wird unterschieden zwischen der Validität der Ergebnisse, der Validität der Konstruktion, der internen Validität

und der externen Validität. Anhand der verschiedenen Typen der Validität werden die durchgeführten Experimente bewertet und diskutiert.

Validität der Ergebnisse Bei der Validität der Ergebnisse wird erfasst, inwieweit ein Ergebnis sich von den verschiedenen Eingabeparameter beeinflussen lässt. Gerade bei der Softwareentwicklung sind die Vorkenntnisse der Entwickler einschlagende Eingabeparameter. Aber auch der Zeitaufwand und die zur Verfügung stehenden Stunden sind solche Parameter.

Validität der Konstruktion Die Validität der Konstruktion bezieht sich auf den Aufbau des Experiments und dem angewandten Konzept. Hierbei ist sicherzustellen, dass die Konzepte korrekt angewandt werden. Das bedeutet, dass die Umsetzung der Theorie in die Praxis bei der Durchführung des Experiments angewandt wird. Fehler können hier entstehen, wenn der Aufbau des Experiments nicht klar definiert ist oder Raum für Alternativwege gelassen wird, was eine ungültige Abbildung vom theoretischen Konzept auf den praktischen Aufbau darstellt.

Interne Validität Bei der internen Validität steht die Qualität der Ergebnisse im Vordergrund, die bei der Durchführung der Methodik erzielt wird. Hierbei sind die Ergebnisse valide, wenn die erzielten Ergebnisse mit dem gewünschten oder erwarteten Ergebnis übereinstimmen. Die Durchführung selbst wird hierbei durch verschiedene Variablen beeinflusst. Bei den Experimenten ist es daher wichtig, dass der Einfluss der Variablen nicht das Ergebnis gefährden. Nach [WR+12] erfasst die interne Validität die kausale Abhängigkeit zwischen dem durchgeführten Experiment und den erzielten Ergebnissen.

Externe Validität Die Generalisierung des entwickelten Ansatzes wird durch die externe Validität beschrieben. Während bei der internen Validität die gemessenen Ergebnisse mit den zu erwartenden Ergebnissen übereinstimmen sollen, wird bei der externen Validität die Anwendbarkeit des Ansatzes auf andere Projektkontexte angewandt. Hierbei ist von Interesse, inwieweit welche Teile des Ansatzes ihre Gültigkeit haben und dort verwendet werden können.

8.3 Typ 0-Validierung - Machbarkeit

Die Validierung der Machbarkeit beruht in dieser Forschungsarbeit auf den Forschungsfragen und dem daraus resultierenden Anforderungskatalog aus Abschnitt 3.1. Somit entspricht die Typ-0-Validierung dem (informellen) Nachweis der Erfüllung der in Kapitel 3 gestellten Anforderungen. Die Anforderungen (A1) bis (A7) werden im Folgenden auf die Erfüllung durch die Forschungsbeiträge geprüft. Das Ziel der Validierung ist hierbei zu bestätigen, dass die Systematik anwendbar ist und die

erzielten Ergebnisse eine Verbesserung in Bezug auf die Anforderungen im Vergleich zum bisherigen Stand der Forschung erbringen.

	A1 - Strukturerhaltende Überführung der Analyseartefakte in den Entwurf	A2 - Unterstützung beim Entwurf von Microservice-basierten Anwendungen	A3 - Etablierung des Domänenwissens	A4 - Separierung von Domänenlogik und Anwendungslogik	A5 - Systematische Ableitung und Strukturierung der Schnittstelle	A6 - Einbezug des Testens in das Engineering	A7 - Systematische und nachvollziehbare Ableitung von Tests und Testdaten
[BN+19] Towards a Model-Driven Architecture Process for Developing Industry 4.0 Applications	●	◐	◐	○	◐	/	/
[RS+18] Challenges of Domain-Driven Microservice Design: A Model-Driven Perspective	○	●	◐	◐	◐	/	/
[Gi18] Domänengetriebener Entwurf von ressourcenorientierten Microservices	◐	●	●	◐	●	◐	○
[SL+17] An Open IoT Framework Based on Microservices Architecture	○	◐	○	◐	◐	/	/
[SR+18] Microservice Test Process: Design and Implementation	/	/	/	/	○	●	○
[LM+19] Consumer-Driven Contract Tests for Microservices: A Case Study	/	/	/	/	◐	◐	◐
Vorliegende Forschungsarbeit	●	●	●	●	●	●	●

Tabelle 8.1: Ergebnis der Validierung des Typ 0 in Bezug auf den Anforderungskatalog

Aus der Bewertung zu den Anforderungen ergibt sich Tabelle 8.1. Im Folgenden wird auf die einzelnen Anforderungen eingegangen und die Machbarkeit näher erläutert. Eine vollständig erfüllte Anforderung wird mit dem Zeichen ● angegeben. Ist eine Anforderung nur zum Teil oder bedingt erfüllt, dann ist diese mit dem Zeichen ◐ gekennzeichnet. Eine nicht erfüllte Anforderung ist mit dem Zeichen ○ gekennzeichnet. Nicht anwendbare Anforderung sind mit dem Zeichen "/" markiert.

Strukturerhaltende Überführung der Analyseartefakte in den Entwurf Die strukturerhaltende Überführung der Analyseartefakte in den Entwurf stellt sicher, dass die Analyseartefakte gezielt in die gewünschte Ziel-Architektur überführt werden können. Durch die Festlegung der Terminologie, der beteiligten Akteure, sowie der beteiligten Systeme werden die notwendigen Grundlagen gelegt. Eine vorgegebene Strukturierung der Anwendungsfälle anhand des Anwendungsfall-Diagramms liefert die Ausgangsbasis (siehe Abschnitt 4.3.1) und gruppiert die verschiedenen Capabilities. Die benötigten Daten werden anhand der Informationsanforderungen in den spezifizierten Anwendungsfällen festgelegt. Anschließend erfolgt die systematische und strukturerhaltende Überführung der Artefakte in die vorgegebene Ziel-Architektur. Anhand einer festgelegten Reihenfolge der Aktivitäten und Richtlinien zur Überführung kann eine systematische Überführung der Artefakte durch die Entwickler erfolgen.

Unterstützung beim Entwurf von Microservice-basierten Anwendungen Die Anforderung zur Unterstützung beim Entwurf wird von der Systematik auf verschiedene Arten erfüllt. Durch die Vorgaben bei der Erstellung der Anforderungen wird eine strukturerhaltende Überführung in den Entwurf ermöglicht. Dadurch erhält der Software-Architekt bereits eine grundlegende Strukturierung der Software-Architektur. Durch die frühzeitige Einordnung der Funktionalitäten in die Capabilities wird der Aufwand weiter reduziert. Anhand der Systematik lassen sich die benötigten Artefakte nacheinander gezielt aufstellen. Dies beinhaltet die Extraktion des Domänenwissens, die Modellierung der Anwendungslogik mit der Möglichkeit zur Ableitung der API, sowohl als die Überführung in die Implementierung. Bei der Etablierung des Domänenwissens und insbesondere der Wiederverwendbarkeit der Softwarebausteine wird der Entwickler durch die nachfolgenden Anforderungen unterstützt.

Etablierung des Domänenwissen Durch die Etablierung des Domänenwissens wird die wiederverwendbare Geschäftslogik herauskristallisiert. Hierfür wird ein eigenständiger Prozess eingeführt, der unabhängig vom Anwendungsentwicklungsprozess stattfindet und die Gegebenheiten in der Domäne analysiert. Hierzu wird die Terminologie der Domäne erfasst und diese wird in Subdomänen strukturiert. Die beteiligten Entitäten und die Geschäftsfunktionalität wird durch die Bounded Contexts eingefangen. Ohne diese Überlegungen wäre nicht gewährleistet, dass die Domäne durchdrungen und die (benötigte) Geschäftslogik korrekt verstanden wird. Das Herausziehen der Funktionalität liefert insbesondere in Bezug auf die Wiederverwendbarkeit einen großen Beitrag, da ohne dieses extrahierte Wissen im Kontext der Entwicklung weiterer Softwaresysteme sonst nicht verfügbar wäre, bzw. direkt im Anwendungskontext eingebettet wäre. Ein weiterer Schritt ist die anschließende Formalisierung, welche sicherstellt, dass die Domäneninhalte auch in die Implementierung überführt werden können. Dazu werden als Basis Diagramme aus UML verwendet, die durch ein UML-Profil (siehe Abschnitt 4.2.2) entsprechend erweitert werden.

Separierung von Domänenlogik und Anwendungslogik Nach der Aufstellung der Domäneninhalte ist insbesondere beim Entwurf wichtig, dass diese Domänenlogik sauber von der Anwendungslogik getrennt wird. Wird Anwendungslogik in die angedachten Microservices der Domäne integriert, besteht die Gefahr, dass diese die Charakteristik der Wiederverwendbarkeit verlieren. In Abschnitt 6.2 wird die Orchestrierung solcher Microservices zum Anwendungsbezug anhand der Task-Prozesse aufgezeigt und exemplarisch durchgeführt. Neben der eigentlichen Domäne, in der sich die Anwendung bewegt, werden hier insbesondere die weiteren Querschnittsdomänen von der Anwendungslogik separiert.

Systematische Ableitung und Strukturierung der Schnittstelle Die Schnittstelle ist eines der zentralen Elemente eines Microservices. Die Unterstützung des Software-Entwicklers und Architekten beim Ableiten der Schnittstelle ist essenziell um keine willkürlichen Schnittstellendefinitionen, sondern qualitativ hochwertige Schnittstellen zu erhalten. Das eingeführte API-Diagramm in der Entwurfsphase hilft hierbei die beteiligten Entitäten und Funktionalitäten zu erfassen. Anhand der Ableitungsregeln und der Vorgaben der Systematik (siehe Abschnitt 4.11) werden die wichtigen Bestandteile einer REST-basierten API-Spezifikation aufgestellt. Dies beinhaltet die Extraktion der HTTP-Methode, die Identifikation des Endpunktes und der Ressource, die benötigte Anfrage (Request) und Antwort (Response) inklusive des Datentyps, sowie die Extraktion der Statuscodes. Um eine Wiederverwendbarkeit innerhalb der API zu erhalten, wird zudem zwischen den Datentypen und den Operationen unterschieden, um den Wiederverwendungsgrad innerhalb der API-Spezifikation zu erhöhen.

Einbezug der Implementierung und des Testens in das Engineering Mit dem Abschluss des Entwurfs steht die Implementierung und das Testen der Softwarelösung an. Während das Testkonzept direkt bei den Analyseartefakten einsetzt, wird der Entwurf gezielt in die Implementierung überführt. Hierzu wird der Code entlang der vorgeschlagenen Implementierungs-Mikroarchitektur strukturiert. Analog zu der eigentlichen Implementierung werden ebenfalls die Tests einsortiert. Dadurch wird eine Strukturierung erreicht, die insbesondere Entwickler unterstützt, die neu in das Projekt einsteigen. Der Einstieg in existierende Implementierung anderer Microservices wird dadurch ebenfalls vereinfacht. Neben der eigentlichen Implementierung werden die Tests auf den verschiedenen Ebenen erstellt. Die Ende-zu-Ende-Tests resultieren direkt aus den Anwendungsfällen (siehe Abschnitt 7.4 von den Anforderungen nach der Systematik abgeleitet und erlauben das Testen der gesamten Anwendung. Die Integration der verschiedenen Microservices kann unabhängig davon getestet werden. Hier steht der definierte Vertrag aus Abschnitt 7.3 im Vordergrund. Die Tests auf der untersten Ebene nach der Testpyramide (siehe Abschnitt 2.6.1) werden direkt im Microservice implementiert. Dadurch lässt sich gewährleisten, dass die gesamte Anwendung, aber auch die einzelnen Teilsysteme getestet werden.

Systematische und nachvollziehbare Ableitung von Tests und Testdaten Das Festlegen der Testdaten und der Ableitung der zugehörigen Tests ist ein wichtiger Faktor, um die Entwickler zu unterstützen und ein gezieltes Testen der etablierten Softwarelösung zu ermöglichen. Es reicht nicht aus nur die verschiedenen Testarten zu beleuchten, sondern die Testfälle und Testdaten müssen ebenfalls erfasst werden. Die Testfälle selbst lassen sich mittels der Systematik ableiten. Die Testdaten werden anhand der vorhergesehenen Beispiele definiert und festgehalten. Anhand dieses Vorgehens sind die Testdaten bereits im Voraus bekannt und die Ergebnisse der Testausführung sind bekannt.

8.4 Typ I-Validierung - Eignung

Die Eignung des Ansatzes für fortgeschrittene Web-Anwendungen wird anhand einer Typ-I-Validierung durchgeführt. Anhand der Erprobung der Systematik durch die Entwicklung verschiedener Anwendungen aus der Domäne ConnectedCar wie die ConnectedCarServicesApplication (CCSApp) wird die Eignung des Ansatzes festgestellt. Mithilfe von Studien, welche durch Praktikumsarbeiten in verschiedenen Semestern durchgeführt wurden, wurde die Systematik erprobt und weiter verfeinert. Hierbei wurden realitätsnahe Szenarien verwendet. Die Beispielanwendung CCSApp wird bereits in den verschiedenen Kapiteln 5, 6 und 7 verwendet, um die Konzepte bei der Entwicklung zu erklären. Dadurch wird bereits die Typ-I-Validierung durchgeführt. Ein ähnliches Vorgehen verfolgt Giessler [Gi18] und Durdik [Du16]. Weiterhin wurde das Vorgehen auch an einer weiteren Geschäftdomäne, der Healthcare-Domäne erprobt. Die hieraus entstandene Context Map, durch die die Healthcare-Domäne fachlich strukturiert wird, ist in Abbildung 8.2 dargestellt.

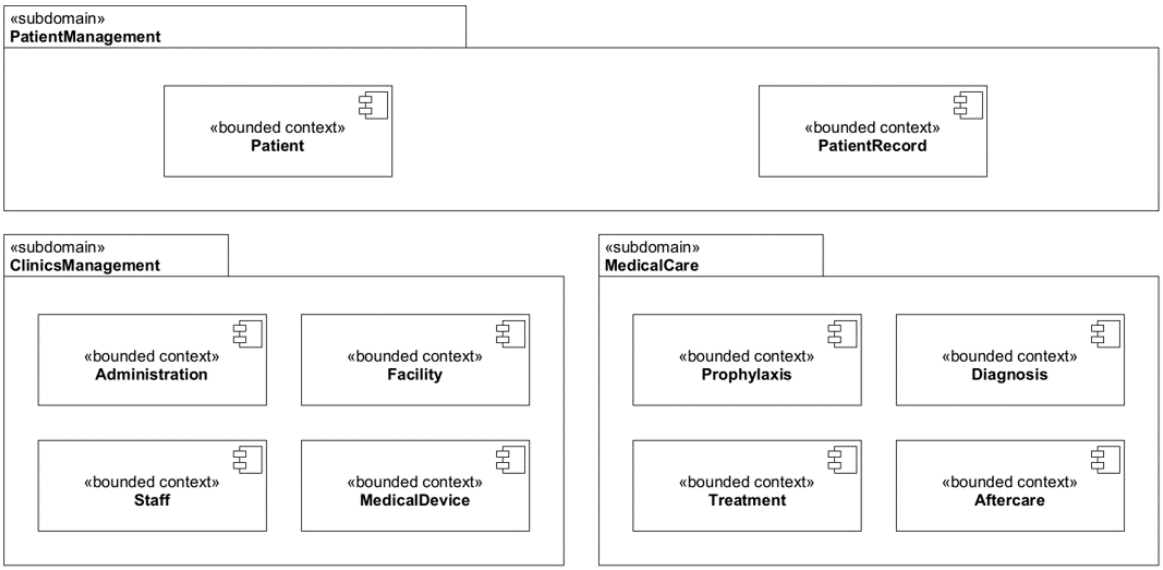


Abbildung 8.2: Conext Map der Domäne Healthcare

Der Patient steht im Mittelpunkt der Aktivitäten im Gesundheitswesen, da das zentrale Ziel des Gesundheitsdienstleisters ist es, den Patienten zu heilen. Diese Subdomäne enthält die zwei Bounded Contexts Patient und PatientRecord. Der Bounded Context Patient beschäftigt sich mit der Kernentität der Subdomäne und modelliert den Patienten. Der Bounded Context PatientRecord verarbeitet und hält die bisherigen medizinische Behandlungen und den medizinischen Status des Patienten. Ein wichtiger Teil der Domäne ist die Verwaltung der Ressourcen des Gesundheitsdienstleisters, z. B. des Personals oder der medizinischen Geräte. Daher deckt diese Subdomäne administrative Aktivitäten und wichtige administrative Entitäten im Bereich des Gesundheitswesens ab. Die Subdomäne MedicalCare kann als das Zentrum der Domäne betrachtet werden. Sie enthält Bounded Contexts, die medizinische Aktivitäten für den Patienten in der Gesundheitsdomäne bereitstellen. Hierdurch wurde auch gezeigt, dass sich der Ansatz für weitere Domänen einsetzen lässt. Eine Anwendung, die auf der Grundlage dieser Domäne entwickelt wurde, ist die Anwendung ClinicsAssetManagement, die die Verwaltung medizinischer Geräte umfasst.

Für die Umsetzung der Anwendung, setzten die Studierenden als Grundlage Leitfäden und bewährte Praktiken, die im Rahmen der Erkenntnisse dieser Forschungsarbeit erstellt wurden. Diese unterstützten die Studierenden bei den anfallenden Arbeiten zur Aufstellung der Analyseartefakte, dem Entwurf, der Ableitung der Schnittstelle, sowie der Implementierung und dem Testen der Softwarelösung. Während des Praktikums wurden die Studierende befragt, wie verständlich und hilfreich die Leitfäden bei der Umsetzung waren. Zusätzlich wurden die Ergebnisse der Studierenden evaluiert, indem die erstellten Lösungen auf die angewandte Systematik, die Struktur und Umsetzung untersucht wurden. Hierbei wurde darauf geachtet, dass die Vorgaben und Konventionen eingehalten wurden. Ein weiterer Gesichtspunkt der Auswertung waren die Anzahl an Rückfragen und Unklarheiten, die die Studierende bei der Anwendung der Systematik hatten. Diese Rückfragen wurden für die Schärfung der Konzepte verwendet.

In verschiedenen Praktikumsarbeiten wurden in den Semestern in den Jahren 2021 und 2022 verschiedene Anwendungen basierend auf den Ergebnissen der Forschungsarbeit erarbeitet. Als Beispiele dienen hier die Anwendungen ElectricCarCharcher, welche den Autofahrer bei der Suche nach freien Ladesäulen für Elekrofahrzeuge unterstützt und die Anwendung PredictiveCarMaintenance, die eine vorausschauende Wartung basierend auf Sensorinformationen ermöglicht. Hierbei haben Studierende der Informatik, welche sich nach der Evaluation von Tichy [Ti00] für eine solche Evaluation eignen, die Anwendungen im Umfeld der vernetzten Fahrzeuge entwickelt.

8.4.1 Ergebnisse der Typ I-Validierung

Die Ergebnisse der Studierenden, die die Systematik genutzt haben, wurden in den Vergleich zu den Entwicklungsarbeit ohne den systematischen Ansatz gesetzt. Verglichen wurden die erstellten Artefakte gemäß den gegebenen Vorgaben. Die Resultate ohne die Verwendung des Ansatzes waren weniger

strukturiert, die Gestaltung der API wurde sehr unterschiedlich durchgeführt und die Implementierung war unterschiedlich strukturiert, was die Einarbeitung in den Code erschwerte. Weiterhin wurden die Tests willkürlich eingesetzt, was zu keinem zielgerichteten Ergebnis führte. Mit der vorgegebenen Systematik waren die Artefakte systematisch aufgestellt, unter anderem waren die Datenformate viel geordneter, die erstellten API-Spezifikationen von den verschiedenen Teams waren vergleichbar. Ein weiterer Vorteil ergab sich bei der Domänenmodellierung. Hier konnte auf die bereits erstellten Modellierungen zurückgegriffen werden, was die Entwicklungszeit der angedachten Anwendungen weiter reduzierte. Auf Basis der etablierten Domänen-Microservices in der ConnectedCar-Domäne konnten die Domänen-Microservices bei der Entwicklung neuer Anwendungen wiederverwendet werden. Änderungen gab es hier insbesondere bei der Erweiterung der Funktionalität. Bei der Überführung in die Implementierung wurde der Code strukturiert und die Funktionalitäten gemäß den Anforderungen getestet. Als Resultat lässt sich ausdrücken, dass die systematische Ableitung zu gezielteren und systematischen Ergebnissen führt.

	SoSe 21	WS 21/22	SoSe22	WiSe 22/23
Studierende	6	13	5	12
	Rückfragen	Rückfragen	Rückfragen	Rückfragen
Allgemeines Vorgehen	12	9	6	4
Modellierung der Domäne	7	4	4	0
Aufteilung der Fachlichkeit in Microservices	5	6	3	2
Entwurf zur API-Spezifikation	7	9	5	0
Einbezug der Querschnittsfunktionalität	10	16	4	1
Insgesamt	41	44	26	8
Befund der API-Spezifikation	zufällig	zufällig	systematisch	systematisch
Befund Implementierung bzgl. Micro-Architektur	Keine	vorhanden	vorhanden	vorhanden
Befund Tests	Wenig	zufällig	systematisch	systematisch

Tabelle 8.2: Auswertung über die verschiedenen Semester

Tabelle 8.2 zeigt die Ergebnisse der Rückfragen der Studierenden, sowie die Ergebnisse der Evaluation der erstellten Artefakte. Die Ergebnisse basieren auf der Grundlage von Rückfragen der Studierenden an die Betreuer, aufgetretene Probleme und abgelegte Code-Artefakte der DevOps-Plattform GitLab. Diese sind mit Vorsicht zu betrachten, weshalb die Bedrohung der Validierung in Abschnitt 8.4.2 erläutert werden. Insgesamt liefert die Evaluation aber den Trend, dass durch entsprechende Leitfäden die systematische Vorgehensweise bei der Entwicklung Klarheit liefert und weniger Komplikationen auftreten. Die unten stehenden Ergebnisse in der Tabelle zu der Evaluation der Artefakte zeigt diesen Trend ebenfalls auf.

8.4.2 Bedrohung der Validierung

Als Grundlage für die Bedrohung der Validität werden die zuvor betrachteten Bedrohungen aus Abschnitt 8.2 herangezogen. Bei der Betrachtung wird die Ausgangssituation, die Variablen, sowie die Ergebnisse betrachtet und auf Einschränkungen bzw. Gefahren bewertet. Anschließend erfolgt ein Fazit über die Validität der Experimente zur Validierung der Forschungsarbeit.

Validität der Ergebnisse In jedem Semester waren die Ausgangsbedingungen für die Experimente vergleichbar. Jedes Entwicklerteam bestand in der Regel aus 4-5 Studierenden, wobei in einigen Semestern mehr als ein Entwicklerteam zur Verfügung stand. Jedes Entwicklerteam arbeitete an einem eigenständigen Projekt. Die Anzahl an Personenstunden für die Umsetzung der Projekte war dadurch ausgeglichen. Ein Unterschied bestand in den Vorkenntnissen und Erfahrungen. Das Experiment selbst startete immer mit einer Einarbeitung in den Ansatz, wodurch der Wissensstand weitestgehend ausgeglichen wurde. Die geleisteten Stunden wurden mittels einem Stundenzettel erfasst und in verschieden kategorisiert. Dadurch lässt sich die eingesetzte Zeit für Besprechungen und der Entwicklungsarbeit einsehen. Problematisch ist hierbei, dass die Eintragungen auf einer Selbstdisziplin der Studierenden beruhen und keine Garantie besteht, dass diese immer korrekt waren.

Interne Validität Die größte Variable bei der internen Validität sind die Studierenden. Der Grund dafür ist, dass jeder Studierende unterschiedliche Fähigkeiten und Vorkenntnissen besitzt. Die Anzahl der Fragen bzgl. dem Vorgehen hat sich bei den Studierenden zwar reduziert, aber dies kann aufgrund der Vorkenntnisse und Faktoren persönlicher Natur wie Schüchternheit ebenfalls beeinflusst sein. Wie bei der Arbeit von Hippchen [Hi21] festgestellt, lassen sich diese Einflüsse nicht bei den Experimenten abschalten und sind immer gegeben. Daher ist eine Kausalität dieser Abhängigkeiten gegeben. Die Tendenz über die Semester zeigt insgesamt, dass es weniger Rückfragen hinsichtlich des Vorgehens von den Entwicklern gab.

Validität der Konstruktion Bei der Durchführung der domänengetriebenen Entwicklung der fortgeschrittenen Web-Anwendungen wurde auf die Einhaltung der vorgegebenen Systematik und der Richtlinien Wert gelegt. Hierbei wurden die erzielten Ergebnisse basierend auf Dokumentationen und Reviews bewertet und anschließend besprochen. Dieser Zyklus fand in einem wöchentlichen, bzw. zweiwöchigen Rhythmus statt. Dadurch wurde sichergestellt, dass die Vorgehensweise inklusive der Systematik eingehalten wurde und die erzielten Ergebnisse der Studierenden nicht verfälscht wurden. Lediglich die Vorkenntnisse und die Erfahrungen der Studierenden im Bereich der Softwareentwicklung haben sich unterschieden. Bei den Rückfragen wurde insbesondere die Kommunikation über die verwendeten Kommunikationsmittel ausgewertet. Hierbei ist es problematisch, dass diese

Kommunikation wenig strukturiert vorlag. Die allgemeine Tendenz ist aus der Auswertung aber zu erkennen.

Externe Validität Die Entwicklung einer Anwendung wird in der Regel von Softwarearchitekten und Softwareentwicklern mit Erfahrung durchgeführt. Die durchgeführten Studien hatten verschiedene Anwendungen aus der ConnectedCar-Domäne als Lösungsziel, wodurch die Erprobung des Ansatzes an verschiedenen Softwarelösungen durchgeführt wurden. Die Ergebnisse selbst wurden von Studierenden erzielt, die meistens keine große Erfahrung bei der Entwicklung von komplexen Softwaresystemen besitzen. Mit den bereitgestellten Mitteln und den Beschreibungen des Ansatzes mussten die Studierenden klarkommen und ein umfangreicheres Softwaresystem entwickeln, was den Studierenden unter Anwendung des systematischen Ansatzes gelungen ist. Unter dieser Betrachtung ist die Anwendung des Ansatzes und unter Berücksichtigung der anderen Validierungsarten ein Fortschritt, insbesondere da erfahrene Entwickler den Ansatz erfolgreich bei der Entwicklung einfließen lassen können sollten.

8.5 Typ II-Validierung - Anwendbarkeit

Die Validierung der Anwendbarkeit bestimmter Aspekte des Ansatzes wird durch den Einsatz im industriellen Kontext erprobt. Hierbei ist als Kooperationspartner die iC Consult GmbH beteiligt, von deren die Rahmenbedingungen der Projekte gestellt werden. Aufgrund der Gegebenheiten lassen sich nicht alle Aspekte umsetzten, aber ein Teil der Forschungsbeiträge lassen sich dadurch validieren. Im Kontext mit der iC Consult GmbH wird die systematische Aufstellung der Anwendungsarchitektur und die Separierung der Geschäfts- und Querschnittsdomänen vorgenommen. Zudem werden die Schnittstellen nach dem Vorgehen aufgestellt. Die Überführung in die Implementierung und dem Testen kann nur teilweise dargestellt werden, da die Implementierung zum Zeitpunkt noch nicht abgeschlossen wurde und der Quellcode aufgrund der einer Vertraulichkeitsvereinbarung (engl. Non-Disclosure Agreement, NDA) nicht veröffentlicht werden darf. Die Ergebnisse werden in Abschnitt 8.5.1 diskutiert.

Zu Beginn der Studie wurde, ähnlich wie in [Hi21], die Kenntnisse der einzelnen Entwickler anhand eines Fragebogens erfasst. Tabelle 8.3 gibt einen Überblick über die eingenommenen Rollen und die Berufserfahrung der einzelnen Teilnehmer. Die eigentlichen Erfahrungen in Bezug auf die relevanten Themenbereiche, die vom systematischen Ansatz der domänengetriebenen Entwicklung von Web-Anwendungen vorhanden waren, werden in Abbildung 8.3 dargestellt. Für jede der Antwortmöglichkeiten wurde eine Skala von 1-5 vorgegeben. Der obere Wert (5) gibt an, dass viele Erfahrungen und Erkenntnisse diesbezüglich vorhanden waren, während der untere Wert angibt, dass die Erfahrungen gering waren.

Rolle	Anzahl
Projektleiter	1
Softwareentwickler (Senior)	2
Softwareentwickler (Junior)	4

Berufsjahre	Anzahl
<2 Jahre	3
2-5 Jahre	2
>5 Jahre	2

Tabelle 8.3: Übersicht und Erfahrung der Beteiligten

Zu den abgefragten Themengebiete gehörte der Umgang mit Anwendungsfällen im Bereich der Analyse, die bisherigen Kenntnisse im Bereich des domänengetriebenen Entwurfs, der Modellierung allgemein und in Bezug auf die Modellierungssprache UML, die Aufstellung einer API-Spezifikation im Format Open API, sowie das generelle Verständnis über Microservices. Wie Abbildung 8.3 zeigt, sind insbesondere die Kenntnisse in Bezug auf den domänengetriebenen Entwurf (Domain-Driven Design, DDD) eher gering. Dieser Umstand wird gerade bei der Validierung der Systematik in Richtung des separaten Entwicklungsprozesses zur Separierung der Domäneninhalte interessant. Für die Umsetzung der weiteren Konzepte hatte die Kontrollgruppe der Entwickler bereits eine höhere Erfahrung.

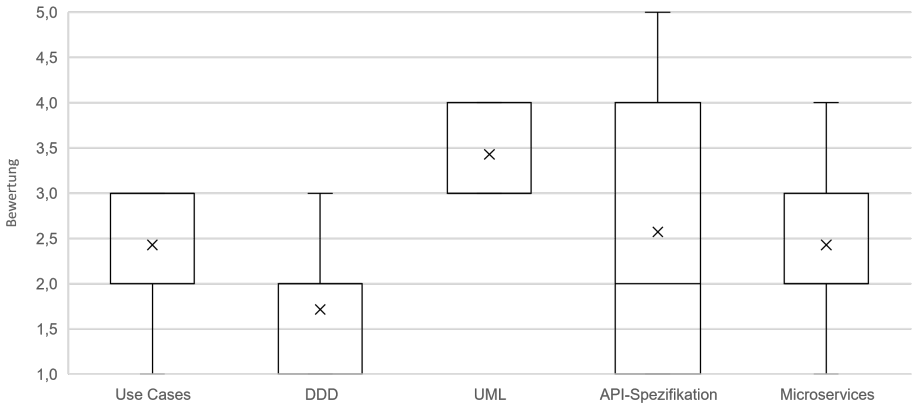


Abbildung 8.3: Erfahrungen im Rahmen der Themengebiete und der Entwicklung

Die Bewertung ist gemäß dem Bewertungsbogen (siehe Anhang 10.1.1 folgendermaßen aufgeschlüsselt. Der hohe Wert (hier 5) gibt eine sehr hohe Erfahrung an. Der niedrige Wert (hier 1) gibt an, dass keine bzw. kaum Erfahrungen in diesem Bereich vorhanden sind.

8.5.1 Industriekontext CustomerPortal

Das Ziel des Industrieprojekts ist es, dass ein Produktionsbetrieb (dessen Namen aufgrund der NDA nicht genannt wird) über ein Kundenportal (engl. customer portal) verschiedene Funktionalitäten für die Kunden bereitstellt. So soll mit der Anwendung CustomerPortal die Kunden ihre Aufträge und Sendungen verwalten können. Hierbei wird ein bisheriges System abgelöst und die neue Anwendung

namens "CustomerPortal" soll über leichtgewichtige Web-Schnittstellen (Web-APIs) verbunden werden. Aus diesem Grunde wurde sich für eine Microservice-Architektur entschieden, welche im Grundsatz der in Abschnitt 4.1 vorgestellten Architektur gleicht.

Für die Entwicklung der Anwendung wurde auf der in dieser Arbeit vorgestellte Engineering-Ansatz zurückgegriffen. Abweichungen, die stattfanden, werden an den entsprechenden Stellen angemerkt.

Durchführung der Analyse Die Anforderungen an die Anwendung wurden in einem Workshop gemeinsam mit dem Kooperationspartner ermittelt und festgehalten. Die Herangehensweise und Dokumentation war vergleichbar mit dem in Abschnitt 5.1 beschriebenen Vorgehen zur CCSApp. Grob lassen sich die Anforderungen wie folgt zusammenfassen: Mit dem Kundenportal kann ein Kunde den Produktkatalog durchsuchen. Wenn der Kunde bestimmte Produkte kaufen möchte, kann er eine Bestellung aufgeben. Eine Bestellung umfasst neben den Produkten auch die Auswahl der Versanddetails, wie z. B. das Versandziel. Nachdem eine Bestellung aufgegeben wurde, kann der Kunde die Bestellung stornieren oder die Sendung verwalten. Die Verwaltung der Sendung umfasst die Änderung des Bestimmungsortes, der Lieferzeitpunkt oder die Anzeige des Sendungsstatus.

Beispielsweise besteht ein Anwendungsfall darin, den Kunden die Möglichkeit zu geben, den Status ihrer Bestellungen und die Versanddaten zu den bestellten Produkten einzusehen. Hierbei wurden ebenfalls die Informationsanforderungen aufgestellt, die das System zur Ausführung der Funktionalität benötigt. Ein weiterer wichtiger Punkt ist, dass die Bestell- und Versandinformationen bereits über ein monolithisches Enterprise-Resource-Planning-System (ERP-System) zur Verfügung stehen. Hierzu war ein detailliertes Verständnis des ERP-Systems und der zugrundeliegenden Domäne erforderlich. Ein wichtiger Teil für die Entwicklung des Systems war daher zu Beginn die Analyse des ERP-Systems. Die benötigten Inhalte auf der Systemebene wurde, wie in Abschnitt 5.1.3 erläutert, vom Team anhand der Anforderungsfälle extrahiert. Bei der Extraktion beschränkte sich Team nur auf die notwendige Information zur Erfüllung der Anforderungen. Die semantische Abbildung der Informationen aus dem ERP-System auf den System-Microservice wurde tabellarisch festgehalten.

Für die Bewertung des Ansatzes wird ein Fragebogen und der Austausch mit den Entwicklern vorgenommen. Die herangezogenen Metriken belaufen sich hierbei auf die Verständlichkeit und der Nachvollziehbarkeit des Ansatzes, die praktische Anwendbarkeit und dem zeitlichen damit verbundenen Aufwand. Anschließend wurde noch die Korrektheit der erarbeiteten Ergebnisse geachtet. Hierbei wurden Ausschnitte der Ergebnisse herangezogen und ein Soll-Ist-Vergleich wurde durchgeführt. Abbildung 8.4 zeigt die Auswertung der Befragung der verschiedenen Entwickler bzgl. der Analysephase.

Das Team gab an, dass es entsprechend dem Vorgehen detailliertere Analyseartefakte als üblich generiert hat. In Bezug darauf wurde angegeben, dass der zeitliche Aufwand höher als sonst war, da die Einarbeitung in die Systematik und die Aufstellung der Artefakte aufwendiger war. Es wurde aber

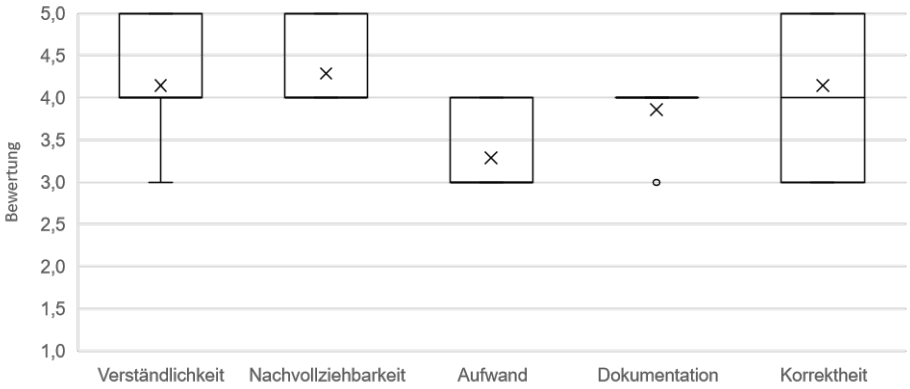


Abbildung 8.4: Rückmeldungen zur Analysephase

auch angegeben, dass sich dieser Aufwand im Hinblick auf die Systematik vermutlich lohnen wird. So wurde die Verständlichkeit und die Nachvollziehbarkeit des Vorgehens als hilfreich bis sehr hilfreich eingestuft. Die Korrektheit der Ergebnisse sah das Team ebenfalls gegeben. Dies hat sich bei der Überführung in den Entwurf bestätigt. Die erstellten Analyseartefakte wurden bei der Überführung in den Entwurf geringfügig angepasst. Hier gab das Team durchweg an, dass sich die Artefakte dafür gut eignen.

Erfassung und Modellierung der Domäne Das Konzept zur Erfassung der Domäne unabhängig vom Anwendungsentwicklungsprozess war für das Entwicklerteam neu. Daher wurde zu Beginn ein Workshop veranstaltet, bei dem die Konzepte von DDD und das Vorgehen erklärt wurden. Das Domänenwissen und die Domäneninhalte wurden gemeinsam mit den Domänenexperten (dem Kunden) extrahiert und festgehalten. Zum Beginn wurde die Domäne in die verschiedenen Bereiche strukturiert. Ein Ausschnitt der resultierenden Context Map ist in Abbildung 8.5 dargestellt. Die für die Erfüllung der geforderten Funktionalitäten der Anwendung CustomerPortal lässt dich von Bounded Contexts aus den drei Subdomänen CustomerManagement, ProductManagement und OrderManagement erfüllen. Im Zentrum steht hierbei die Bestellung (engl. Order) des Kunden, welche an diesen ausgeliefert werden soll.

Für jeden der für die Anwendung benötigte Bounded Context wurde anschließend eine Bounded Context Entity Relation View (BCERV) modelliert, welche die eigenen und geteilten Entitäten enthält. Abbildung 8.6 zeigt die BCERV des Bounded Contexts Order. Im Zentrum des Bounded Contexts Order steht die Entität Order, welche vom Kunden ausgelöst wird. Ein Kunde kann hierbei verschiedene Bestellungen aufgeben, welche verschiedene Produkte enthalten kann. Die Bestellung selbst beinhaltet verschiedene Versandinformationen. Insbesondere die Adresse, an die die Bestellung versendet werden soll, ist hier wichtig.

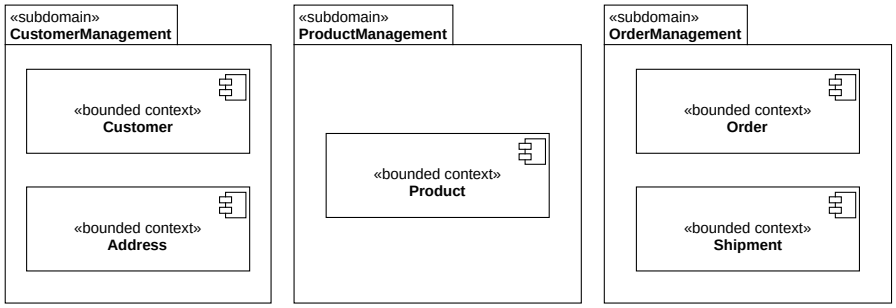


Abbildung 8.5: Context Map der ERP-Domäne

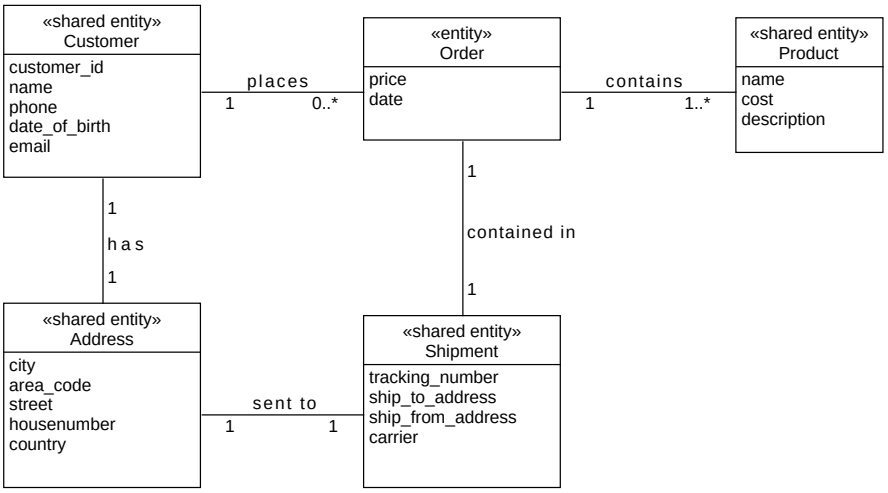


Abbildung 8.6: Context Map des Bounded Contexts Order

Die herangezogenen Metriken belaufen sich hierbei auf die Nachvollziehbarkeit und Verständlichkeit des Ansatzes zur Extraktion des Domänenwissens anhand des separaten Domänen-Entwicklungsprozesses aus Abschnitt 4.2. Es ist anzumerken, dass nur vier der Entwickler sich mit der separaten Domänenmodellierung beschäftigt haben.

Unter Berücksichtigung der eher geringen Vorerfahrung der Entwickler mit den DDD-Konzepten lassen sich aus den Ergebnissen positive Rückschlüsse für die Systematik ziehen. So gaben die Entwickler an, auch ohne Vorkenntnisse einen guten Einstieg gefunden zu haben. In diesem Zusammenhang wurde auch untersucht, inwieweit die Wiederverwendbarkeit der erstellten Domäneninhalte von den Entwicklern gesehen wird. Abbildung 8.7 zeigt die Auswertung der Umfrage in Bezug auf die Domänenmodellierung. Die Verständlichkeit des Ansatzes wurde von den Teilnehmern als hoch oder sehr hoch eingeschätzt. Der Aufwand hingegen wurde von den Entwicklern teilweise als eher mäßig bis hoch eingestuft. Es wurde angegeben, dass die Gespräche mit den Domänenexperten

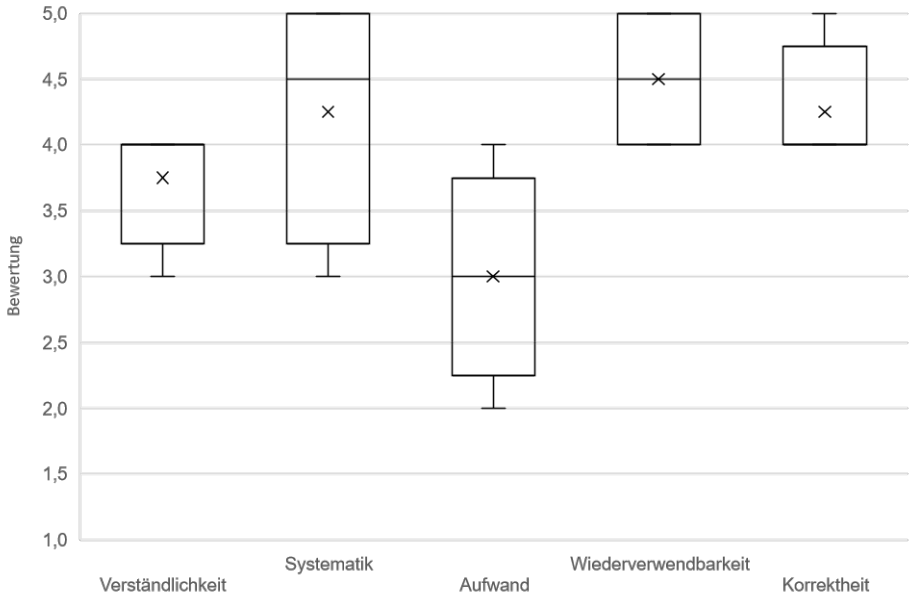


Abbildung 8.7: Ergebnisse zur Domänenmodellierung

insbesondere zu Beginn aufwändig waren, bis die Domäne in einem gewissen Umfang erfasst war. Das modellierte Domänenwissen wurde von allen Entwicklern als positiv angesehen. Es gab es eine eindeutige Rückmeldung, dass dieses und die umgesetzten Services ein großes Potential zur Wiederverwendung haben. Bei der Korrektheit der Ergebnisse hatten die Entwickler den Eindruck, dass die Ergebnisse stimmig sind. Da sich die Untersuchung in diesem Kontext auf eine Anwendung beschränkt, ist die konkrete Wiederverwendbarkeit der Domäne nicht überprüfbar. Hierzu müsste ein weiteres Experiment in dieser Domäne durchgeführt werden.

Aufstellung der Architektur Nach dem die Anforderungen erfasst waren und die Verbindung des externen Systems geklärt war, wurde die Software-Architektur aufgestellt. Diese basiert auf der Domänenmodellierung und der in der Anforderungsanalyse erarbeiteten Anwendungsfalldiagramm. Anstatt der in Kapitel 4 eingeführten Namensgebung des Anwendungs-Microservices wurde im Kontext des Industriepartners sich dazu entschieden, die Microservices als Prozess (engl. process microservice) zu bezeichnen. Analog wurde das Kürzel P- für diesen Microservice-Typen eingeführt. Abbildung 8.8 zeigt die resultierende Softwarearchitektur der Anwendung CustomerPortal. Auf der Infrastrukturebene wird die Schnittstelle des ERP-Systems konsumiert und über eine REST-Schnittstelle vom System-Microservice S-ERP zurückgegeben. Der System-Microservice wird erweitert, sobald zusätzliche Anforderungen neue Informationsanforderungen liefen. Aufgrund des komplexen monolithischen Charakters des eingesetzten ERP-System und unter Betrachtung des domänengetriebenen Entwurfs, wurde auf der Ebene der Domäne zwei Domänen-Microservices eingeführt. Die Domäne selbst ist bisher nur einem gewissen Grad eingeführt, soll aber tiefer strukturiert werden, sobald weite-

re Bounded Contexts benötigt werden. Der benötigte Ausschnitt der Gesamtdomäne aus Abschnitt 8.5.1 inkludiert für die Architektur die Domänen-Microservices D-Order und D-Shipment. Bei Bedarf werden die Domänenausschnitte weiter modelliert. So wurde insbesondere bei der Betrachtung der weiteren Funktionalität noch weiteres Domänenwissen extrahiert, welches in die Domänenmodellierung mit eingeflossen ist. Auf der Anwendungsschicht wird für den oben eingeführten genannten

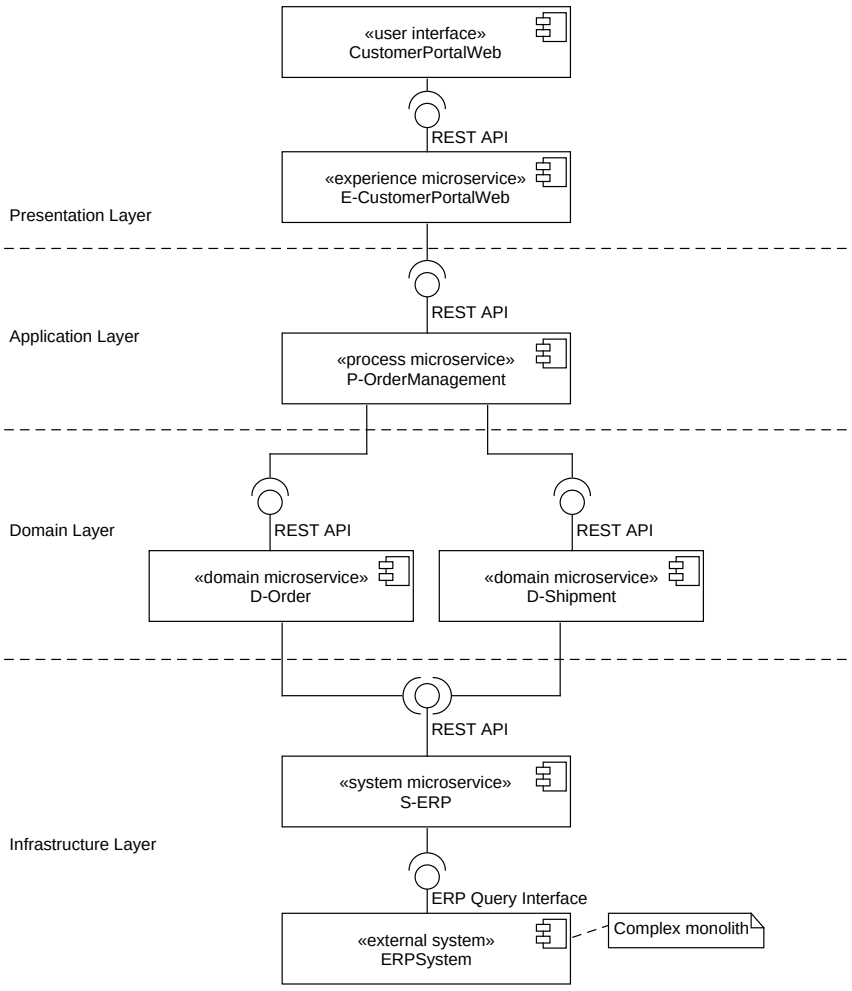


Abbildung 8.8: Architektur der Anwendung CustomerPortal

Anwendungsfall der Microservice P-OrderManagement für die benötigte Geschäftslogik abgeleitet. P-OrderManagement orchestriert die darunterliegenden Domänen-APIs D-Orders und D-Shipments. Entsprechend der Anforderungen werden die Daten verarbeitet und dem Experience Microservice E-CustomerPortalWeb bereitgestellt, der wiederum die Antwort entsprechend den Bedürfnissen der Benutzeroberfläche CustomerPortalWeb aufbereitet.

Die dargestellte Software-Architektur ist vereinfacht dargestellt. So wurden beispielsweise die zusätz-

lichen Proxies zur Absicherung der einzelnen Microservices ausgelassen. Die Software-Architektur wurde anhand der bisherigen Artefakte aufgestellt. Ein Kritikpunkt an den Ansatz von den Entwicklern aus war, dass die Authentifizierung und Autorisierung nicht Teil des Ansatzes sind. Hier hätten sich die Entwickler noch weitere Unterstützung gewünscht. Weiterhin gab es Fragen zu der benötigten Proxies, um die Autorisierung durchzuführen.

Entwurf des Anwendungs-Microservices P-OrderManagement Der Entwurf der Microservices aus der Anwendungsschicht ist erfolgt, nachdem die Software-Architektur modelliert und die Domäne initial erfasst war. Im Kontrast zum eigentlichen Entwicklungsansatz wurden keine expliziten API-Diagramme für die Domäne erstellt, sondern die Domänen-API-Diagramme wurden direkt im API-Diagramm der Anwendung mit modelliert. Abbildung 8.9 zeigt einen Ausschnitt des API-Diagramms.

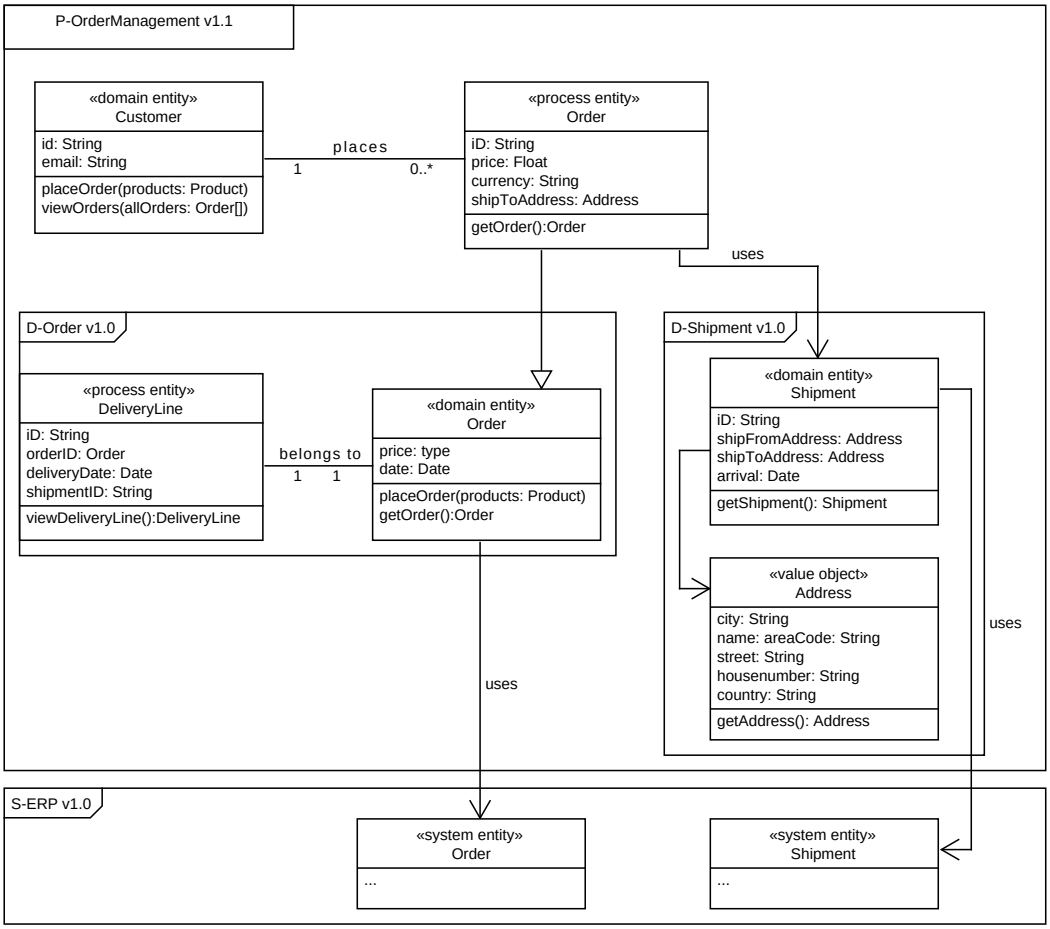


Abbildung 8.9: Ausschnitt aus dem API-Diagramm zu P-OrderManagement

Ein Nachteil, der sich hieraus ergibt, ist, dass das Domänenwissen zur API-Ableitung bei der Anwendung dokumentiert ist. Der Hintergrund war eine pragmatischere Lösung und die Idee diese Information separat zu modellieren, sobald diese auch wirklich gebraucht wird.

Die wichtige Entität Order erbt hierbei die Informationen aus dem Domänen-Microservice. Gemeinsam mit den Informationen, die aus dem Domänen-Microservice D-Shipment bezogen werden, werden die benötigten Informationen zur Lieferanschrift der Bestellung orchestriert. Die notwendigen Funktionen sind in den Entitäten enthalten und beschränken sich in dem dargestellten Diagramm auf eine kleine Auswahl. Die Verknüpfung zum ERP-System und dem dazugehörenden System-Microservice ist nur grob skizziert.

Analog zum Entwicklungsansatz und dem Vorgehen aus Abschnitt 4.3.2 wurde anschließend die API aus dem API-Diagramm abgeleitet und spezifiziert. Hierbei wurden zuerst die Schnittstellen der Domänen-Microservices D-Order und D-Shipment entwickelt. Daraufhin wurde die API des Anwendungs-Microservices P-OrderManagement etabliert. Abbildung 8.10 gibt einen Ausschnitt der beiden Domänen-Microservices wieder.

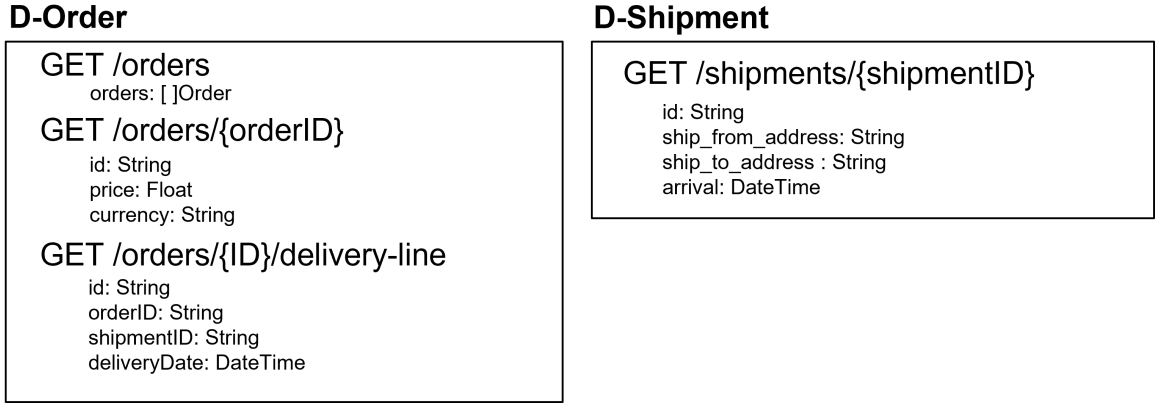


Abbildung 8.10: Abgeleitete API-Endpunkte und Schemas der Domänen-Microservices D-Order und D-Shipment

Die Erstellung eines API-Diagramms und dessen systematische Ableitung war für die Entwickler neu. Bisher wurde die API-Spezifikation insbesondere unter Berücksichtigung der API-Richtlinien erstellt und eine systematische Ableitung aus einem Entwurfsartefakt fand nicht statt. Es wurde angegeben, dass das systematische Vorgehen zu einer Zeiteinsparung führte, da sich die API-Spezifikation deutlich schneller in einem stabilen Zustand befand und die Anforderungen direkt mit berücksichtigt waren. Dieses Ergebnis korreliert mit den Ergebnissen des Fragebogens (siehe Abbildung 8.11) und dem damit verbundenen Aufwand.

Die Verständlichkeit des Ansatzes wurde für diesen Teil des Ansatzes mit einer durchschnittlichen Bewertung mit ungefähr 4,3 Punkten angegeben. Entsprechend hatten die Entwickler auch den

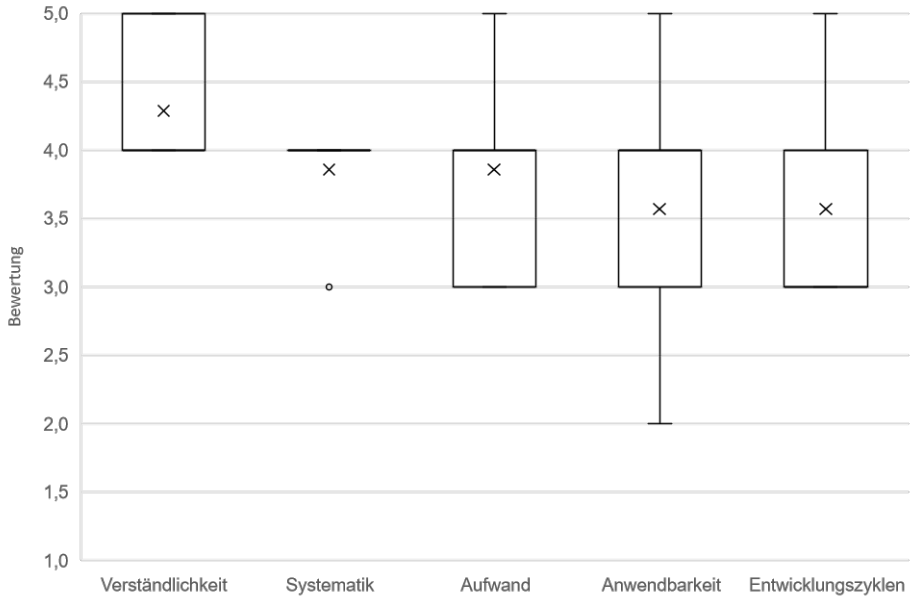


Abbildung 8.11: Ergebnisse der Befragung zur Modellierung und dem API-Entwurf

Eindruck, ein klares Verständnis über die Systematik erhalten zu haben. Was bei der Systematik in Richtung der API-Spezifikation angesprochen wurde, war, dass eine weiterführende Unterstützung gemäß den Statuscodes für die Fehlerfälle wünschenswert gewesen wäre. Der Aufwand wurde vom Entwicklerteam als angemessen eingestuft. Hinsichtlich der Anwendbarkeit gab es ein Feedback, dass das API-Diagramm redundant zu den anderen Artefakten ist und dass die API-Spezifikation auch daraus abgeleitet werden kann. Die API-Spezifikation lässt sich zwar aus den verschiedenen Artefakten ableiten, jedoch fehlt hierbei die Systematik, die das API-Diagramm mit sich bringt. Die Anzahl der Entwicklungszyklen bzgl. des Ansatzes war im üblichen Rahmen und hat für das Entwicklungsteam gepasst.

Überführung in die Implementierung und Testen Bei der Überführung in die Implementierung wurde die vorgeschlagene Mikroarchitektur aus Abschnitt 4.4 verwendet, diese wurde aber noch um eigene Strukturelemente erweitert. Neben der Implementierung wurde hier noch eine Struktur zur Platzierung der benötigten Inhalte für die Ausführung in der Pipeline und der Auslieferung ergänzt. Die Prinzipien des Testkonzepts gemäß 7 wurden angewendet.

Abbildung 8.12 zeigt, wie die Bewertung der Entwickler bezüglich dem Implementierungs- und Testkonzept. Die Verständlichkeit des Konzepts wurde als hoch bis sehr hoch angesehen. Der Aufwand bezüglich des Konzepts wurde mäßig bis moderat eingestuft. Die durchgeführte Umsetzung zeigt analog zu der Rückmeldung, dass sich die Konzepte gut umsetzten ließen. Mit dem Ergebnis der Implementierung waren die Entwickler zufrieden. Dies beinhaltet ebenso die umgesetzten Tests und

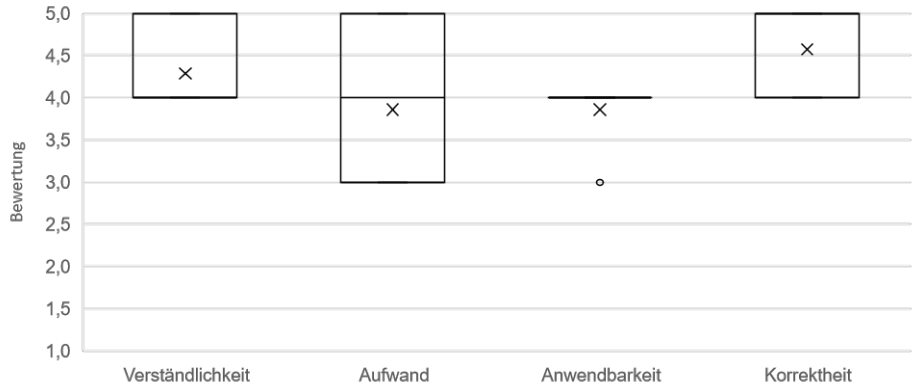


Abbildung 8.12: Ergebnis der Umfrage zur Implementierung und Tests

deren Anzahl.

Bewertung des Gesamtansatzes Abschließend wird der Gesamtansatz im Hinblick auf die durchgeführten Entwicklungsarbeiten betrachtet. Abbildung 8.13 zeigt einen Überblick der Auswertung des Bewertungsbogens.

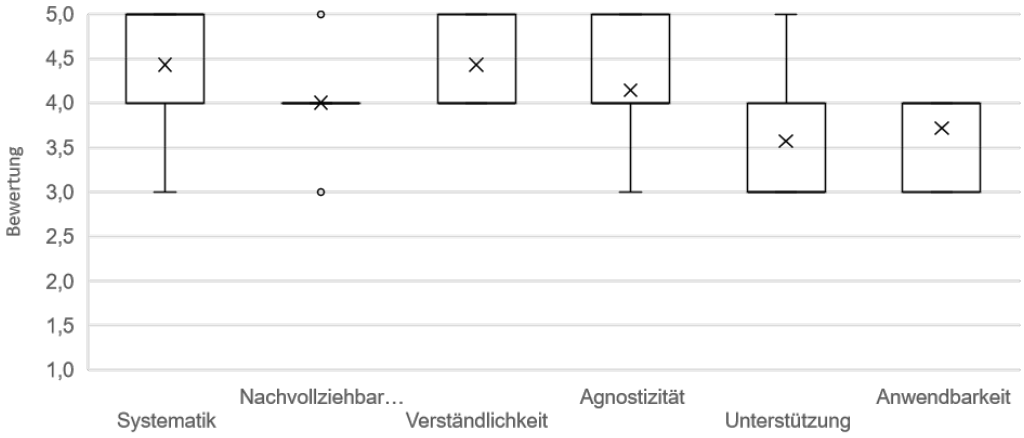


Abbildung 8.13: Ergebnis der Umfrage zum Gesamtansatz

Das Entwicklerteam hat den Umgang mit der Systematik als sehr hoch eingeschätzt. Bei der Nachvollziehbarkeit gab es ebenfalls ein positives Feedback. Weiterhin sehen die Entwickler, dass keine Abhängigkeiten zu bestimmten Technologien erfolgt. Als ein Defizit wurde genannt, dass nur REST-basierte APIs unterstützt werden. Insbesondere in einem funktionsorientierten Paradigma wäre die Ableitung von sogenannten RPC-APIs (Remote Procedure Call) wünschenswert gewesen. Basierend auf dieser Rückmeldung wurde zusätzlich die Ableitungsmöglichkeit solcher APIs aus dem API-Diagramm untersucht.

8.5.2 Bedrohung der Validität

Dieser Abschnitt behandelt die Einschränkungen und Bedrohungen der Validität für die Typ-II-Validierung.

Ergebnisvalidität Die Durchführung der Studie wurde gemeinsam in einer Kooperation mit einem Industriepartner durchgeführt. Beteiligt war der Kooperationspartner und der Kunde des Kooperationspartners. Für die Studie selbst stand ein eingeschränkter Zeitraum, sowie eine festgelegte Personenzahl zur Verfügung. Zudem waren Studierende der Informatik bei der Durchführung beteiligt. Die Gesamtanzahl der beteiligten Personen der Studie im Projektrahmen belief sich auf sieben Personen. Alle Beteiligten waren bereits mit der Entwicklung von Web-Anwendungen in einem gewissen Maße vertraut, gerade beim Kunden des Projektpartners waren die Kenntnisse jedoch geringer. Neu war für die Entwickler die Domänenmodellierung mit DDD und die Betrachtung dieser als eigenständiger Prozess. Zudem wurde in der bisherigen Entwicklung keine gezielte Systematik eingesetzt. Aufgrund der relativ geringen Anzahl der Studienteilnehmer fehlt gegebenenfalls die Aussagekraft der Daten, die erzielten Ergebnisse wurden aber im realen Kontext generiert und wurden gemäß der Systematik erarbeitet. Die Ergebnisvalidität kann daher als mäßig eingestuft werden.

Interne Validität Bei der interne Validität sind die unterschiedliche Berufserfahrung der Teilnehmer eine Gefahr. Dies ist aber bei Softwareprojekten oftmals der Fall, dass erfahrene und weniger erfahrene Entwickler zusammen arbeiten. Dies zeigt auch die Auswertung des Bewertungsbogens in Tabelle 8.3. Auch die Einführung der neuen Systematik kann aufgrund der Einstellungen der Entwickler zu Frustrationen führen. Insbesondere war der Kenntnisstand in Bezug auf den domänengetriebenen Entwurf gering. So war bei der eigenständigen Betrachtung der Domäne zu Beginn die Skepsis der Entwickler hoch, da diese als rein zusätzlicher Aufwand angesehen wurde. Die Motivation der Entwickler beeinflusst und letztendlich auch die Studiendaten. Das führte auch dazu, dass der Ansatz reduziert umgesetzt wurde, indem das API-Diagramm für die Domäne nicht explizit modelliert wurde. Insgesamt betrifft dies aber nur die Dokumentation der Artefakte, welches durch eine weitere Iteration verbessert werden kann. Insgesamt wird die interne Validität im Bezug auf die erzielten Ergebnisse als

Konstruktvalidität Die Teilnehmer können einen unterschiedlichen Wissensstand haben. Dieser basiert auf der Berufserfahrung und verschiedenen praktischen Erfahrungen. Um diesen Stand einzufangen, wurden Befragungen durchgeführt, um Erfahrungen diesbezüglich zu sammeln und Ausreißer auszuschließen. Bei der Umsetzung der Softwarelösung wurde auf die Einhaltung der Systematik geachtet. Teile der Systematik sind kreative Leistungen, aber die Einhaltung der Systematik ist erkennbar und führt zu Ergebnissen, die weiterverwendet werden können. Gerade im Praxisbezug

wurden kleinere Abweichungen in Bezug auf Konzepte vorgenommen. So wurde beispielsweise kein eigenes API-Diagramm für die Domäne erstellt, sondern dieses wurde in das API-Diagramm für die darüberliegenden Process Microservices integriert. Diese Abweichungen lassen sich aber vertreten, da diese kaum bis keine Auswirkungen auf die Systematik haben. Die Konstruktvalidität wird daher als moderat bis hoch eingestuft.

Externe Validität Es ist eine hohe externe Validität anzunehmen, da die Studie im Rahmen eines Industrieprojekts durchgeführt wurde, welcher zum Ziel die Entwicklung einer fortgeschrittenen Web-Anwendung hatte. Eine Bedrohung der Validität stellt die Durchführung der verschiedenen Schritte des Ansatzes dar. Hierbei sind insbesondere Abkürzungen anzumerken, welche bei der Entwicklung vorgenommen wurden. Dies war dem Zeitdruck und dem begrenzten zeitlichen Rahmen geschuldet, die sich hier auswirken.

8.5.3 Zusammenfassung und Fazit über die Typ II-Validierung

Bei der Durchführung der Validierung wurde geprüft, ob die Ansätze zur Entwicklung einer fortgeschrittenen Web-Anwendung auch im industriellen Kontext anwendbar sind. Es wurde darauf geachtet, wie gut die Entwickler mit der Systematik zurechtkommen und wo die Schwächen des Ansatzes liegen. Mit Hilfe von Bewertungsbögen wurde ein Feedback vom Entwicklerteam eingeholt. Die Rückmeldungen gaben keinen Rückschluss auf größere Probleme mit dem systematischen Vorgehen. Bei der Analysephase wurden kurz die nicht-funktionalen Anforderungen angemerkt. Diese waren aber kein Bestandteil des Ansatzes und werden deshalb vernachlässigt. Beim Entwurf der Domäne und der Extraktion der Fachlichkeit hatte das Team nur einen hohen zeitlichen Aufwand angemerkt. Die Systematik selbst wurde nichts beanstandet. Es wurde bei der API-Ableitung aus Gründen der Einfachheit kein eigenes API-Diagramm für die Domänen-Microservices erstellt. Die Überführung des Entwurfs in die Implementierung verlief ebenfalls ohne Rückfragen. Als Gesamtergebnis lässt sich somit sagen, dass die Umsetzung des Entwicklungsansatzes ein Erfolg war. Es lässt sich folgern, dass die Systematik das Entwicklerteam ganzheitlich unterstützt.

8.6 Zusammenfassung der Validierung

Für die Validierung der Forschungsbeiträge wurde die empirische Validierung herangezogen und orientiert sich an den vorgestellten Validierungsarten von Durdik [Du16]. Im Vordergrund der Validierung stehen die Typ 0- (Machbarkeit), Typ I- (Eignung) und Typ II-Validierung (Anwendbarkeit). Die Typ III-Validierung wurde aufgrund des hohen Aufwands und der hohen Kosten nicht betrachtet. Die Durchführung der Validierung fand durch die Entwicklung verschiedener Anwendungen unter

der Verwendung der in der Arbeit vorgestellten Konzepte statt. Hierbei wurden Anwendungen aus der Domäne ConnectedCar im Labor und Anwendung namens CustomerPortal gemeinsam mit einem Industriepartner entwickelt. Durch das Heranziehen der empirischen Validierungsarten wurde geprüft, ob die verschiedenen Prozessschritte die Entwickler bei den Entwicklungstätigkeiten einer fortgeschrittenen Web-Anwendung unterstützen.

Der Vergleich bestehender Arbeiten mit den Konzepten der hier betrachteten Forschungsarbeit war Bestandteil der Typ 0-Validierung. Der Anforderungskatalog aus Abschnitt 3.1 lieferte hierzu die Grundlage. In der Validierung wurde gezeigt, dass die Beiträge der Forschungsarbeit die offenen Lücken der bestehenden Arbeiten im Vergleich zu den Anforderungen schließen.

Die Typ-I Validierung wurde im Rahmen von Entwicklungsarbeiten gemeinsam mit Studierenden über mehrere Semester hinweg durchgeführt. Die Typ I-Validierung betrachtet hierbei auf alle der Forschungsbeiträge dieser Arbeit. Hierzu wurden Projektteams gebildet, die verschiedene Anwendung aus der Domäne ConnectedCar umgesetzt haben. Die erzielten Ergebnisse aus den verschiedenen Semestern wurden genutzt, um Defizite zu beheben und die Systematik weiter zu schärfen.

Als dritte Validierungsart wurde die Typ II-Validierung durchgeführt. Hierbei wurde gemeinsam mit dem Industriepartner eine Anwendung entwickelt. Die Entwickler erhielten hierbei die Systematik für die domänengetriebene Entwicklung von fortgeschrittener Web-Anwendungen. Anhand dieser Systematik wurde die Entwicklung durchgeführt und die Eindrücke der Entwickler mittels eines Bewertungsbogens festgehalten. Ein besonderer Fokus der Studie lag auf dem Entwurf und dem separaten Entwicklungsprozess für die Domäne, weshalb diese Artefakte tiefer beleuchtet werden. Im Gegensatz dazu wird die Implementierung und das Testen nur kurz beleuchtet.

9 Fazit und Ausblick

Ein immer wieder auftretendes Problem bei der Softwareentwicklung ist die Wiederverwendbarkeit von Softwarebausteinen. Ein Problem ist hierbei, dass verschiedene Funktionalität der Software immer wieder neu entwickelt wird. Diese Arbeit liefert einen Beitrag zur Etablierung wiederverwendbarer Softwarebausteine während der Anwendungsentwicklung im Rahmen fachlicher Domänen. Hierbei wird in Abschnitt 9.1 ein Fazit über die Beiträge und Ergebnisse erhoben. Im Abschnitt 9.2 folgen mögliche Fragestellungen, die durch zukünftige Arbeiten behandelt werden sollten.

9.1 Fazit

Der Beitrag zur Verbesserung der Wiederverwendbarkeit bei der Entwicklung von fortgeschrittenen Web-Anwendungen extrahiert die anwendungsagnostischen Inhalte in eigene Softwarebausteine. Bei den bestehenden Arbeiten wird dieser anwendungsagnostische Kern mit in die Anwendungsentwicklung integriert, wodurch die Wiederverwendbarkeit nicht gegeben ist.

Ein wichtiges behandeltes Defizit des Standes der Forschung ist die Extraktion der anwendungsagnostischen Fachlichkeit. Durch den zentralen Beitrag des systematischen Engineering-Ansatzes zur Entwicklung von fortgeschrittenen Web-Anwendungen in dieser Arbeit wird das Defizit angegangen. Zu Beginn wurde eine Zielarchitektur festgelegt, welche vom Microservice-Engineering-Ansatz erreicht werden soll. Durch die Einführung eines zweiten Teilprozesses, den Domänenentwicklungsprozess, wird der anwendungsagnostische Anteil extrahiert und systematisch erfasst. Dadurch werden die wiederverwendbare Softwarebausteine abgeleitet, die sich bei der Entwicklung verschiedener Anwendungen einsetzen lassen. Durch den Ansatz lassen sich neben der fachlichen Domäne auch die Inhalte für Querschnittsdomänen ableiten. Am Beispiel des Internets der Dinge (Internet of Things, IoT) wurde dies aufgezeigt.

Für die Implementierung und das Testen wurde ebenfalls eine Systematik erstellt, die es den Entwicklern erlaubt aus den resultierenden Artefakten des Engineering-Ansatzes gezielt funktionale Tests für die zu entwickelnde Anwendung abzuleiten. Im folgenden erfolgt ein Fazit für die einzelnen Beiträge.

B1 - Systematischer Engineering-Ansatz zur Aufstellung der Anwendungsarchitektur

Der zentrale Beitrag dieser Arbeit ist ein systematischer Engineering-Ansatz zur Entwicklung von fortgeschrittenen Web-Anwendungen aus Kapitel 4. Hierbei ist der Ansatz in zwei Teilprozesse, den Domänenentwicklungsprozess und den Anwendungsentwicklungsprozess, aufgeteilt. Der Domänenentwicklungsprozess läuft neben dem Anwendungsentwicklungsprozess als eigenständiger Prozess und dient der Erfassung der Domäne und deren Logik. Das hier erreichte Ergebnis ist, dass die Domäne analysiert und in eigenständige, fachliche Bereiche eingeteilt wurde. Dadurch lassen sich die anwendungsagnostischen Inhalte extrahieren und eigene Softwarebausteine entwerfen, die für viele Anwendungen wiederverwendbar sind. Begonnen wird mit der Analyse der Domäne und der strategischen Aufteilung der Fachlichkeit gemäß DDD anhand der sogenannten Context Map. Die Context Map strukturiert die Domäne in verschiedene Unterdomänen und ordnet die Geschäftslogik in sogenannte Bounded Contexts ein. Jeder etablierte Bounded Context ist ein Kandidat für einen Microservice. Für jede zu entwickelnde Anwendung aus der Domäne liefern die aufgestellten Domäneninhalte die Grundlage. Soll eine neue Anwendung innerhalb der Domäne entwickelt werden, lassen sich die bereits etablierten Services der Domäne wiederverwenden. Dadurch wurde das Ziel eine verbesserte Wiederverwendbarkeit zu erreichen erhöht, da erfasste Domäneninhalte nicht erneut modelliert und implementiert werden müssen. Bei der Entwicklung einer solchen fortgeschrittenen Anwendung sind auch die auftretenden Querschnittsdomänen wie das Internet der Dinge berücksichtigt. Hierbei werden die immer wieder benötigten Funktionalitäten aus den Microservices der Anwendung herausgezogen und so erfasst, dass diese auch bei der Entwicklung von anderen Anwendungen wiederverwendet werden können.

Beim Anwendungsentwicklungsprozess wird mit der Erfassung der Anforderungen begonnen, welche strukturerhaltend in den Entwurf überführt werden. Der behandelte Ansatz beinhaltet die strukturerhaltende Überführung in die Software-Architektur während der Entwurfsphase. Beim Entwurf der Software-Architektur selbst wird zwischen Anwendungs- und Domänenmodellierung unterschieden. Zu Beginn wird eine Zielarchitektur festgelegt, welche vom Microservice-Engineering-Ansatz erreicht werden soll. Hierdurch wird erreicht, dass eine gezielte und strukturerhaltende Überführung der Analyseartefakte in den Entwurf erfolgt. Ein weiterer wichtiger Aspekt ist die Herstellung des Bezugs der Domäne zu der Anwendungsentwicklung. Der Anwendungsentwicklungsprozess selbst wird in Bezug zu den Inhalten der Domäne vorgenommen. Neben der Trennung der Fachlichkeit ist auch die Etablierung der Schnittstellen wichtig. Der Entwurf ist so gestaltet, dass die Schnittstellen systematisch aus den erstellten Diagrammen abgeleitet werden können. Für die anstehende Implementierung wurde eine systematische Strukturierung vorgeschlagen. Kapitel 5 greift den Ansatz auf und illustriert die Anwendbarkeit der eingeführten Konzepte wodurch die Orchestrierung der von den beiden Teilprozessen entstandenen Microservices gezeigt wurde. Dazu wurde ein Diagramm eingeführt, welches die Verknüpfung der verschiedenen Softwarebausteine ermöglicht.

B2 - Separierung der Geschäfts- und Querschnittsdomäne

Die Trennung der Querschnittsdomäne wurde am Beispiel der IoT-Domäne anhand Kapitel 6 vorgestellt. Hierbei wurden verschiedene Quellen analysiert und die Fachlichkeit der Domäne erfasst. Durch die Ableitung eines konzeptionellen Modelles wurde die Context Map aufgestellt. Für den Ausschnitt "SensingDevice wurde konkret auf die Modellierung der Fachlichkeit eingegangen und mit Constraints präzisiert. In einem weiteren Schritt wurde darauf eingegangen, wie der abgeleitete Domänen-Microservice in der Anwendungsentwicklung eingesetzt werden kann. Durch das Vorgehen wurde gezeigt, wie sich nach dem Domänenentwicklungsprozess auch Querschnittsdomänen erfassen lassen. Dies erlaubt die Wiederverwendbarkeit solcher Querschnittsbereiche, da diese bei vielen Anwendungen benötigt werden. Anhand der ConnectedCar-Anwendung CCSApp wurden die extrahierten Domäneninhalte bei der Anwendungsentwicklung erfolgreich eingesetzt. Das Beispiel der Querschnittsdomäne IoT kann nicht nur im ConnectedCar-Umfeld verwendet werden, sondern die dort erzielten Ergebnisse ließen sich auch bei Anwendungen des Gesundheitswesens einsetzten.

B3 - Testkonzept für Microservice-basierte Anwendungen

Ein weiterer Beitrag in Bezug auf das Gesamtkonzept ist das in Kapitel 7 vorgestellte Testkonzept. Dieses Testkonzept berücksichtigt hierbei die verschiedenen Arten von Tests wie Unit-, Komponenten-, Integrations-, Vertrags- und Ende-zu-Ende-Tests und wird in den Gesamtprozess eingeordnet. Bei einem verteilten Software-System werden hierbei weitere Herausforderungen an das Testen betrachtet, da die benötigten Funktionalitäten auf verschiedene Services verteilt sind und über Schnittstellen (z. B. über REST-APIs) kommunizieren. Der Entwickler enthält eine Vorgehensbeschreibung, die es ihm ermöglicht die notwendigen Tests systematisch zu entwickeln. Um einen Microservice testen zu können, muss dieser bereitgestellt werden und zugänglich sein. Mittels einer Pipeline lassen sich die Tests vollautomatisiert auf Code-Änderungen ausführen und bei erfolgreichem Testdurchlauf direkt ausliefern. Hierfür wurden Möglichkeiten zur Integration in einer Pipeline vorgestellt.

Validierung

Die Validierung der Beiträge erfolgte in Kapitel 8 anhand von drei Validierungstypen und zugehörigen Experimenten. Die Ergebnisse der Experimente wurden auf Bedrohungen und Einschränkungen der Validität untersucht. Die Typ 0-Validierung (Machbarkeit) fand durch den Autor gemäß des Anforderungskatalogs statt. Die in Kapitel 3 untersuchte Literatur wurde hierbei mit den Beiträgen verglichen. Die Typ 1-Validierung (Eignung) wurde anhand der Erprobung des vorgestellten Ansatzes zur domänengetriebenen Entwicklung von fortgeschrittenen Web-Anwendungen durch die Entwicklung verschiedenerer Anwendungen und zweier fachlicher Domänen (ConnectedCar und

Healthcare) überprüft. Basierend auf der Auswertung wurden dann die Verbesserung der Entwicklungsarbeiten durch den Prozesses vorgenommen. Der Validierungstyp 2 (Anwendbarkeit) wurde in einem Industriekontext durchgeführt, indem der Ansatz bei einem Kooperationspartner eingesetzt und die Entwicklung eines CustomerPortals in einer weiteren fachlichen Domäne vorgenommen wurde. Die Ergebnisse und Auswertung werden durch die Validierung aufgegriffen. Die verschiedenen durchgeführten Experimente bestätigen, dass sich die Beiträge für den Einsatz eignen.

9.2 Ausblick

Die in der Arbeit behandelten Forschungsfragen erfüllen die aufgestellten Anforderungen, aus den Prämissen und der Validierung ergeben sich weitere Fragestellungen, die in diesem Themenbereich untersucht werden sollten. In diesem Abschnitt werden diese weiterführenden Fragestellungen kurz aufgegriffen.

Formalisierung der Artefakte

Bislang werden die Artefakte im Engineering-Ansatz durch ein pragmatisches Vorgehen aufgestellt. Das aktuelle Ziel ist eine Formalisierung als einfaches UML-Profil. Gerade hier stellt sich die Frage, in welche Richtung die Formalisierung weitergeführt werden sollte, um eine automatisierte Überführung der Artefakte gemäß der modellgetriebenen Architektur zu erreichen. Dadurch ließe sich automatisiert die API und der benötigte Quellcode generieren, wodurch weitere Zeiteinsparungen folgen können.

Betrachtung weiterer API-Stile

Die betrachtete Architektur und die darin vorgesehenen Microservices kommunizieren aktuell nur über REST-basierte Web-Schnittstellen. Abhängig von der Architektur und dem eingesetzten Stil können auch weitere Arten der Kommunikation zwischen den Microservices relevant werden. Bei einem funktionsorientierten Paradigma würde sich "Remote Procedure Call" (RPC) anbieten. Hier wäre dann insbesondere die Ableitung der API-Spezifikation aus dem API-Diagramm relevant. Die ereignisgetriebene Architektur (event-driven architecture) ist ein weiterer verbreiteter Architektur-Stil. Bei diesem Stil sind die Ableitung der Ereignisse und der Message Broker relevant. Abhängig von den Bedürfnissen einer Anwendung können sich diese Paradigmen besser eignen und könnten daher als weiterer Untersuchungsgegenstand in Bezug auf den systematischen Ansatz betrachtet werden.

Nicht-funktionale Tests

Das Implementierungs- und Testkonzept behandelt nur funktionale Tests. Gerade bei einer Microservice-Architektur ist die Netzwerkverbindung zwischen den einzeln verteilten Microservices relevant. Aus der Aufteilung der Domäne- und der Anwendungslogik auf verschiedene Microservice-Typen resultiert der Nachteil, dass eine höhere Netzwerklatenz entsteht. Eine weitere Forschungsfrage kann sich darauf beziehen, wie die Verteilung der Microservices durchgeführt werden muss, um diesen Nachteil möglichst minimal zu halten. Hierfür sind dann auch nicht-funktionale Tests durchzuführen, um die bei einer entsprechenden Belastung der Anwendung angemessene Antwortzeiten sicherzustellen. Dadurch könnte auch die entstehende Last auf die einzelnen Microservices innerhalb der Anwendung bestimmt werden.

Sicherheit

Das Thema Sicherheit ist bei verteilten Anwendungen wie der Microservice-Architektur ein wichtiges Thema. Sicherheitsrelevante Aspekte wurden in dieser Arbeit nicht betrachtet. Dies betrifft beispielsweise die Authentifizierung und Autorisierung beim Entwurf der Anwendung. Gemäß dem Konzept zur Extraktion der Querschnittsfunktionalität ist gerade in diesem Bereich zu überprüfen, wie sicherheitskritische Mechanismen bei der Anwendungsentwicklung (und den extrahieren Domänen-Microservices) etabliert werden können. Dies beinhaltet die Authentifizierung von Benutzern, wie auch Mechanismen bezüglich der Autorisierung.

10 Anhang

10.1 Ergänzungen

10.1.1 Bewertungsbogen

Karlsruher Institut für Technologie

Bewertungsbogen – Domänengetriebene Entwicklung von fortgeschrittenen Web-Anwendungen

Vorwort zum Bewertungsbogen

Sehr geehrte(r) Teilnehmer/in,

im Rahmen meiner Forschungsarbeit zur Erlangung der Promotion am Karlsruher Institut für Technologie (Fakultät für Informatik, Forschungsgruppe Cooperation & Management, Prof. Abeck) beschäftige ich mich mit der domänengetriebenen Entwicklung von fortgeschrittenen Web-Anwendungen. Der zentrale Beitrag dieser Arbeit ist ein systematischer Engineering-Ansatz zur Entwicklung solcher fortgeschrittenen Web-Anwendungen. Hierbei ist der Ansatz in zwei Teilprozesse, den Domänenentwicklungsprozess und den Anwendungsentwicklungsprozess aufgeteilt.

Zur industrienahen Anwendung des von mir erarbeiteten Ansatzes führe ich ein Experiment durch, dessen Daten zur Validierung genutzt werden. Das Experiment findet während eines Semesters **(Wintersemester 22/23)** statt. Die Beantwortung dieses Bewertungsbogens ist als Teil des Experiments vorgesehen.

Die Erhebung der Daten erfolgt **anonym** und wird in zwei Phasen des Experiments stattfinden. Im ersten Teil und vor Beginn der domänengetriebenen Entwicklung von fortgeschrittenen Web-Anwendungen werden allgemeine Informationen erhoben. Dieser Teil wird vor der Ausführung des Experiments ausgefüllt. Im zweiten Teil werden die Bewertungen zu den einzelnen Teilen des Ansatzes evaluiert. Dies geschieht nach der Durchführung des Experiments. Damit Sie bei der Ausfüllung des Bewertungsbogens möglichst präzise antworten können, empfehle ich Ihnen, diesen vor Beginn der Umsetzung einmalig zu betrachten. Insgesamt beträgt die Dauer der Bewertung circa 20 Minuten, wobei der erste Teil voraussichtlich 5 Minuten einnehmen wird.

Ich bedanke mich bei Ihnen für die Teilnahme an dem Experiment und die gewissenhafte Bearbeitung des Bewertungsbogens!

Mit freundlichen Grüßen
Michael Schneider

Erster Teil: Allgemeine Informationen

Welche Rolle haben Sie aktuell inne?

☐ Junior Softwareentwickler
☐ Senior Softwareentwickler
☐ Leitender Softwareentwickler
☐ Softwarearchitekt
☐ sonstige:

Bitte beschreiben Sie kurz (2 - 3 Sätze) Ihre Aufgabengebiete in dieser Rolle:

Wie viele Jahre arbeiten Sie bereits in der Softwareentwicklung?

☐ weniger als 1 Jahr
☐ seit 1-3 Jahren
☐ seit 3-5 Jahren
☐ mehr als 5 Jahre

In welchen der "klassischen" Softwareentwicklungsbereichen waren Sie bereits tätig?

☐ Frontend-Entwicklung
☐ Backend-Entwicklung
☐ Softwarearchitektur
☐ Betrieb
☐ Infrastrukturen
☐ IT-Security

Wie sehr sind Sie mit der Thematik Entwicklung von Web-Anwendungen vertraut?

☐ nicht vertraut
☐ wenig vertraut
☐ bedingt vertraut
☐ vertraut
☐ sehr vertraut

Wie sehr sind Sie mit dem Einsatz von Anforderungsfällen (Use Cases) für die Anforderungsergebung vertraut?

☐ nicht vertraut
☐ wenig vertraut
☐ bedingt vertraut
☐ vertraut
☐ sehr vertraut

Wie sehr sind Sie mit dem Softwareentwicklungsansatz des domänengetriebenen Entwurfs (engl. Domain-Driven Design, DDD) vertraut?

☐ nicht vertraut ☐ wenig vertraut ☐ bedingt vertraut
☐ vertraut ☐ sehr vertraut

Wie sehr sind Sie mit dem Entwurf von REST-basierten Web-APIs (bspw. mit Open API) vertraut?

☐ nicht vertraut ☐ wenig vertraut ☐ bedingt vertraut
☐ vertraut ☐ sehr vertraut

Wie sehr sind Sie mit dem Architekturstil der Microservices vertraut?

☐ nicht vertraut ☐ wenig vertraut ☐ bedingt vertraut
☐ vertraut ☐ sehr vertraut

Waren Sie bereits an der Entwicklung von Web-Anwendungen beteiligt?

☐ Ja ☐ Nein

Wenn ja, bitte beschreiben Sie kurz das Projekt und die darin gelebte Vorgehensweise:

Wie sehr sind Sie mit Modellierungssprachen wie der Unified Modeling Language (UML) vertraut?

☐ nicht vertraut ☐ wenig vertraut ☐ bedingt vertraut
☐ vertraut ☐ sehr vertraut

Zweiter Teil: Bewertung des Ansatzes

Allgemeine Fragen zum Entwicklungsansatz
Nachfolgend werden Ihnen Fragen zum Ablauf der Migrationsiteration gestellt.

Wie viele Personen waren bei der Entwicklung beteiligt (inklusive Stakeholder)?

☐ 3 – 4 Personen ☐ 5 – 7 Personen
☐ 7– 10 Personen ☐ über 15 Personen

Wie oft wurden Sie bei der Durchführung der Entwicklung durch andere Themen unterbrochen?

Anzahl an Unterbrechungen:

Konnten Sie den Zeitrahmen für die Entwicklung der fortgeschrittenen Web-Anwendung einhalten?

☐ Ja ☐ Nein

Falls nicht, bitte nennen Sie den Grund hierfür:

Fragen zur Analysephase
Die nachfolgenden Fragestellungen widmen sich der Analyse des Entwicklungsansatzes und thematisieren den Verlauf der Spezifikation der Anforderungen.

Konnte mithilfe der Systematik der Analysephase die Anforderungen spezifiziert werden?

☐ Ja ☐ Nein

Falls nicht, bitte begründen Sie:

Konnten alle Anwendungsfälle (Use Cases) zu den Capabilities zugeordnet werden?

☐ Ja ☐ Nein

Falls nicht, beschreiben Sie bitte die Gründe:

Hat die Anforderungsbeschreibung in diesem Maße ausgereicht?

☐ Ja ☐ Nein

Falls nicht, bitte begründen Sie:

Welcher Arten von Anforderungen wurden mit der Systematik erfasst?

☐ Funktionale Anforderung ☐ Nicht-funktionale Anforderung

Wie bewerten Sie das systematische Vorgehen entlang der Analysephase und den damit verbundenen Analyseartefakten?

	sehr niedrig	niedrig	moderat	hoch	sehr hoch
Verständlichkeit	☐	☐	☐	☐	☐
Nachvollziehbarkeit	☐	☐	☐	☐	☐
Praktische Anwendbarkeit	☐	☐	☐	☐	☐
Zeitlicher Aufwand	☐	☐	☐	☐	☐
Korrektheit der Ergebnisse	☐	☐	☐	☐	☐
Umfang der Dokumentation	☐	☐	☐	☐	☐

Welche Verbesserungsvorschläge sehen Sie für die Erfassung der Anforderungen?

Fragen zur Durchführung der Entwurfsphase
Nachfolgend werden Fragen zur Entwicklung der Domäne, der Software-Architektur und dem Entwurf allgemein gestellt.

Teil A: Domänenmodellierung

Konnte mithilfe des Prozesses zur Domänenmodellierung die Domäne strukturiert werden?

☐ Ja ☐ Nein

Falls nicht, bitte begründen Sie:

Wurde auf eine Einhaltung der ubiquitären Sprache geachtet?

☐ Ja ☐ Nein

Falls nicht, bitte begründen Sie:

Wie schätzen Sie ihr Verständnis bezüglich der Geschäftsdomäne nach der Durchführung des Prozesses zur Domänenmodellierung ein?

sehr niedrig	niedrig	moderat	hoch	sehr hoch
☐	☐	☐	☐	☐

Wie bewerten Sie das systematische Vorgehen bzgl. des Prozesses zur Domänenmodellierung?

	sehr niedrig	niedrig	moderat	hoch	sehr hoch
Verständlichkeit	☐	☐	☐	☐	☐
Systematik	☐	☐	☐	☐	☐
Praktische Anwendbarkeit	☐	☐	☐	☐	☐
Zeitlicher Aufwand	☐	☐	☐	☐	☐
Wiederverwend-barkeit	☐	☐	☐	☐	☐
Korrektheit der Ergebnisse	☐	☐	☐	☐	☐

Teil B: Entwurf der Anwendung

Wie bewerten Sie die Überführung der Analyseartefakte und der aufgestellten Domäneninhalte in die Software-Architektur?

nicht hilfreich	wenig hilfreich	moderat	hilfreich	sehr hilfreich
☐	☐	☐	☐	☐

Wie bewerten Sie den Umfang der Software-Architektur?

zu klein	passend	zu umfangreich
☐	☐	☐

Wurden Anforderungen während des Entwurfs überarbeitet?

☐ Ja ☐ Nein

Falls ja, bitte bewerten Sie wie gut sich die Artefakte für die Überarbeitung der Anforderungs-spezifikation eignen:

nicht hilfreich	wenig hilfreich	moderat	hilfreich	sehr hilfreich
☐	☐	☐	☐	☐

War die Systematik zur Ableitung des API-Diagramms aus den Anforderungsfällen (Use Cases) hilfreich?

nicht hilfreich	wenig hilfreich	moderat	hilfreich	sehr hilfreich
☐	☐	☐	☐	☐

Wie hilfreich war das eingeführte API-Diagramm, um die API zu spezifizieren?

nicht hilfreich	wenig hilfreich	moderat	hilfreich	sehr hilfreich
☐	☐	☐	☐	☐

Konnten alle Aspekte der API-Spezifikation mit dem Vorgehen abgeleitet werden?

☐ Ja ☐ Nein

Falls nicht, nennen Sie bitte, wo es Probleme gab:

Wie technologie-affin finden Sie den Ansatz?

sehr niedrig	niedrig	moderat	hoch	sehr hoch
☐	☐	☐	☐	☐

Wie beurteilen Sie das systematische Vorgehen für den Entwurf des Ansatzes zur Entwicklung von fortgeschrittenen Web-Anwendungen?

	sehr niedrig	niedrig	moderat	hoch	sehr hoch
Bezug zu den Anforderungen	☐	☐	☐	☐	☐
Verständlichkeit	☐	☐	☐	☐	☐
Systematik	☐	☐	☐	☐	☐
Wiederverwendbarkeit	☐	☐	☐	☐	☐
Struktur der Web-Schnittstellen	☐	☐	☐	☐	☐
Umfang der Dokumentation	☐	☐	☐	☐	☐
Praktische Anwendbarkeit	☐	☐	☐	☐	☐
Zeitlicher Aufwand	☐	☐	☐	☐	☐
Anzahl der Entwicklungszyklen	☐	☐	☐	☐	☐
Korrektheit der Ergebnisse	☐	☐	☐	☐	☐

Welche Verbesserungsvorschläge sehen Sie für das systematische Vorgehen?

Fragen zur Umsetzung der Implementierung und Tests
Nachfolgend sind Fragen zum Implementierungs- und dem Testkonzept formuliert.

Inwieweit hat Ihnen die Implementierungs-Mikroarchitektur geholfen, die Microservices auf der Implementierungsebene zu strukturieren?

nicht hilfreich	wenig hilfreich	moderat	hilfreich	sehr hilfreich
☐	☐	☐	☐	☐

Hat Ihnen das Testkonzept geholfen, die entwickelte Anwendung gezielter und systematischer zu Testen?

nicht hilfreich	wenig hilfreich	moderat	hilfreich	sehr hilfreich
☐	☐	☐	☐	☐

Hatten Sie davor schon eine Systematik für das Testen eingesetzt? Welche?

Wie beurteilen Sie den Ansatz im Bezuge auf die Implementierungs- und Testphase insgesamt?

	sehr niedrig	niedrig	moderat	hoch	sehr hoch
Verständlichkeit	☐	☐	☐	☐	☐
Nachvollziehbarkeit	☐	☐	☐	☐	☐
Praktische Anwendbarkeit	☐	☐	☐	☐	☐
Zeitlicher Aufwand	☐	☐	☐	☐	☐
Korrektheit der Ergebnisse	☐	☐	☐	☐	☐

Welche Verbesserungsvorschläge sehen Sie für die Implementierungs- und Testphase?

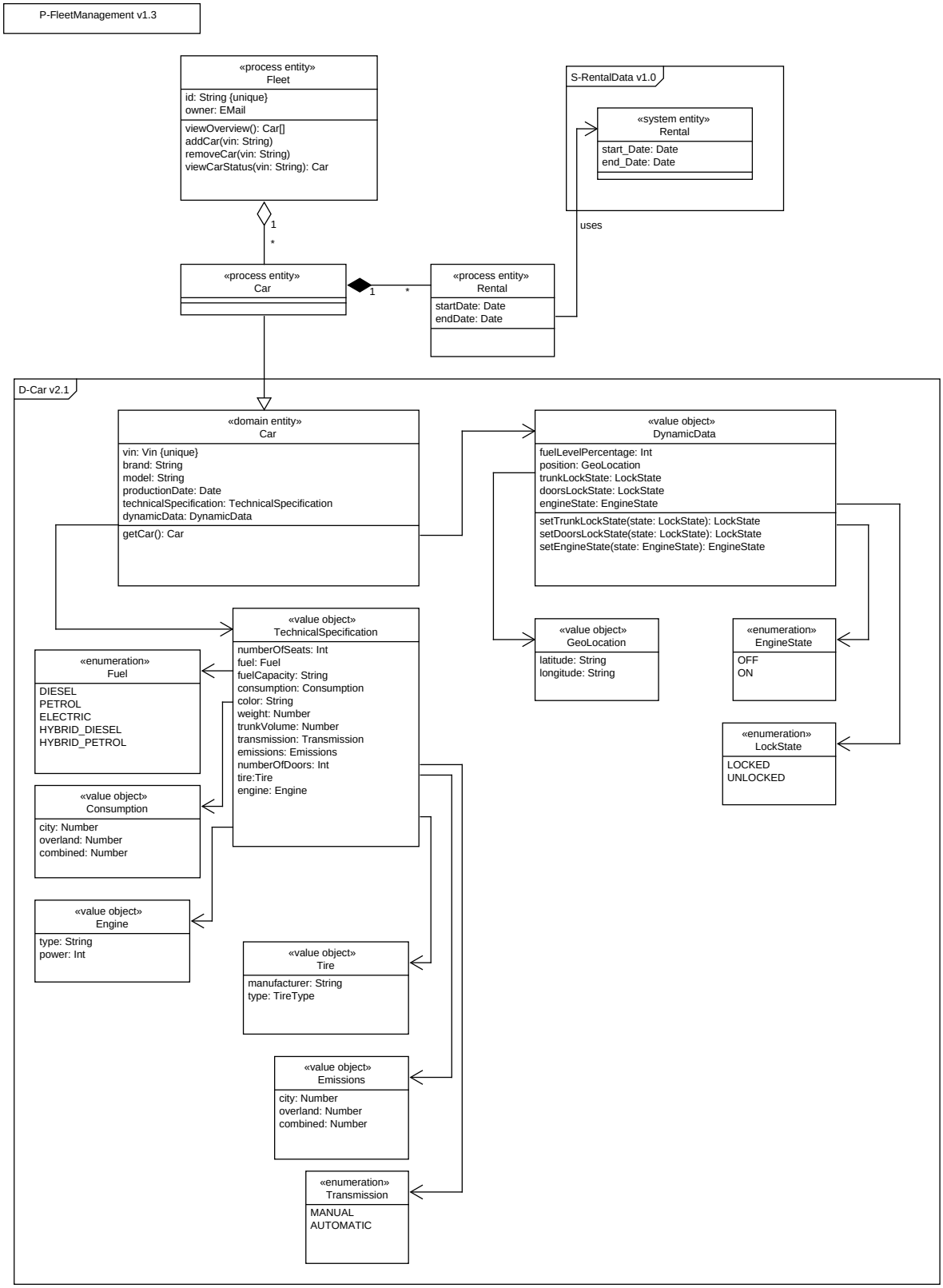
Bewertung des Gesamtansatzes
Zum Abschluss erfolgt eine allgemeine Bewertung für den Ansatz zur domänengetriebenen Entwicklung von fortgeschrittenen Web-Anwendungen.

Wie bewerten sie den Ansatz zur Entwicklung fortgeschrittener Web-Anwendungen in seiner Gesamtheit?

	sehr niedrig	niedrig	moderat	hoch	sehr hoch
Systematik	☐	☐	☐	☐	☐
Nachvollziehbarkeit	☐	☐	☐	☐	☐
Verständlichkeit	☐	☐	☐	☐	☐
Vollständigkeit	☐	☐	☐	☐	☐
Technologie-Agnostizität	☐	☐	☐	☐	☐
Entscheidungsunterstützung	☐	☐	☐	☐	☐
Praktische Anwendbarkeit	☐	☐	☐	☐	☐

Haben Sie noch weitere Verbesserungsvorschläge für den Gesamtansatz oder sonstige Hinweise?

10.1.2 API-Diagramm der CCSApp (A-FleetManagement)



10.2 Beispielhafte Ableitung eines CDC-Vertrags

Um die Aufstellung des Vertrags zu verdeutlichen, wird in Abbildung 10.1 die Aufstellung am konkreten Beispiel erläutert. Im Beispiel aus Abbildung 7.11 sind die beiden Domänen-Microservices die bereitstellenden Microservices und der Anwendungs-Microservice der konsumierende Microservice. Das Vertragsartefakt wird auf der Seite des konsumierenden Microservice als Ressource "contracts" platziert. Hierbei wird jede Interaktion kurz beschrieben. In diesem Fall soll die Interaktion dem Anwendungs-Microservice A-FleetManagement alle Informationen eines Fahrzeugs liefern, die vom Domänen-Microservice Car bereitgestellt werden.

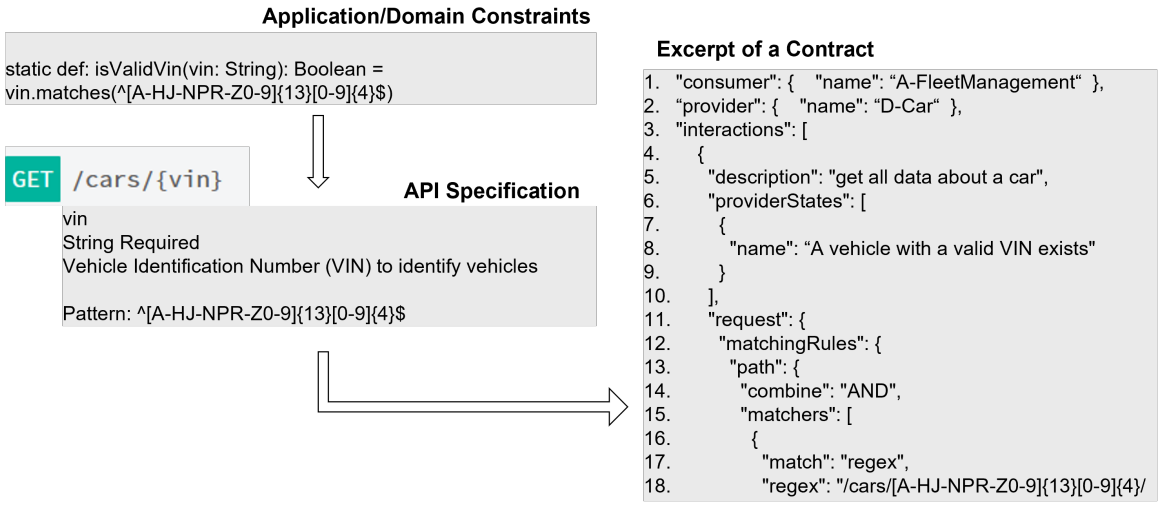


Abbildung 10.1: Ableitung des Vertrags und dessen Inhalte

Die für die Anfrage (request) benötigten Inhalte werden aus der API-Spezifikation des bereitstellenden Microservices abgeleitet. Im aufgezeigten Beispiel ist der Pfad (/cars) enthalten, der auf den API-Endpunkt der Anfrage zeigt, sowie der Parameter VIN (Vehicle Identification Number). Die VIN ist aus der API-Spezifikation des Domänen-Microservice abgeleitet. Durch ein Constraint wird zusätzlich durch einen regulären Ausdruck festgehalten, wie eine VIN aufgebaut ist. Die API-Spezifikation greift dieses Constraint auf und setzt dieses um. Im Vertrag werden solche Constraints durch zusätzliche Regeln festgehalten. Die im Vertrag vorhandenen "matchingRules" ermöglichen es, verschiedene Regeln zu definieren, wie bspw. ein Regex-Muster. Im konkreten Beispiel ist ein regulärer Ausdruck für die VIN angegeben. Diese Regeln werden durch die CDC-Tests implementiert, um eine Validierung der Anfragen und Antworten zu ermöglichen. Im laufenden Beispiel wird als Operator ein Vergleich eingesetzt. Die Regeln werden mit dem Operator AND kombiniert. Hierbei ist der Pfad mit "/car/[A-HJ-NPR-Z0-9]13[0-9]4/" angegeben, wobei die VIN im Pfad als regulärer Ausdruck beschrieben wird. Weitere mögliche Optionen sind ein Abgleich auf Typ, Format oder Gleichheit. Die Abgleichungsregeln in der Antwort sollten möglichst einfach gehalten werden, um komplexe Tests zu

vermeiden.

10.3 Umsetzung der Ende-zu-Ende-Tests am Beispiel

Die konkrete Umsetzung der Ende-zu-Ende-Tests findet im Test-Repository statt. Entsprechend der Vorgehens aus Abschnitt 7.4.1 wird die Ableitung der Ende-zu-Ende Tests am konkreten Beispiel der CCSApp durchgeführt. Das Beispiel wird anhand des Anwendungsfalls "Add Car to Fleet" demonstriert.

Abbildung 10.2 veranschaulicht die Beziehung zwischen den Beschreibungen der Anwendungsfälle und der Übersicht der Testschritte und deren Ergebnisse bei der Testausführung. Die Anwendungsfälle werden systematisch in die verschiedenen Testschritte überführt. Der Titel des Anwendungsfalls spiegelt den Namen des Tests wider. Die Vorbedingungen werden genutzt, um den gewünschten Systemzustand zu erreichen. Hierbei wird im konkreten Fall eine Flotte registriert. Bevor die Vorbedingung zum Tragen kommt, werden weitere zusätzlich benötigte Schritte behandelt. Im konkreten Fall muss die entsprechende Seite der Web-Anwendung geöffnet werden. Nach der Vorbereitung kann die Umsetzung des eigentlichen Flusses beginnen. Die Testschritte werden aus jedem Anwendungsfall eines Flusses abgeleitet, der den primären Akteur als aktives Subjekt enthält. Ablaufschritte, die die Aktionen des Systems oder die Interaktionen zwischen dem System und sekundären Akteuren beschreiben, sind nicht für die Ende-zu-Ende-Tests relevant. Jeder Testfall wird aus der Perspektive des Hauptakteurs abgeleitet und die Aktionen mit der Benutzerschnittstelle werden entsprechend simuliert. Ähnlich wie bei der Implementierung von Gherkin-Schritt-Definitionen werden die Testschritte als wiederverwendbare Funktionen implementiert [Cuc-BDD]. Der Vorteil ist hierbei, dass alternative Testfälle, die bereits implementierten Schritte wiederverwenden können. Die alternativen Abläufe enthalten in der Regel die gleichen Testschritte, ab der Abweichung werden die Schritte gemäß der Abweichung berücksichtigt. Die notwendigen Navigationsschritte in der Benutzerschnittstelle werden in den Testfällen ebenfalls benötigt und werden daher ebenfalls als wiederverwendbare Funktionen implementiert.

Neben dem Hauptfluss werden auch die Alternativflüsse 2a und 2b getestet. Die Testschritte der aus den alternativen Abläufen abgeleiteten Testfälle sind identisch bis zu dem Punkt, an dem der Benutzer (in diesem Fall der Flottenmanager) eine VIN eingibt, um ein Auto der Flotte hinzuzufügen. Die implementierten Funktionen dieser Schritte werden in den alternativen Implementierungen der Testfälle wiederverwendet.

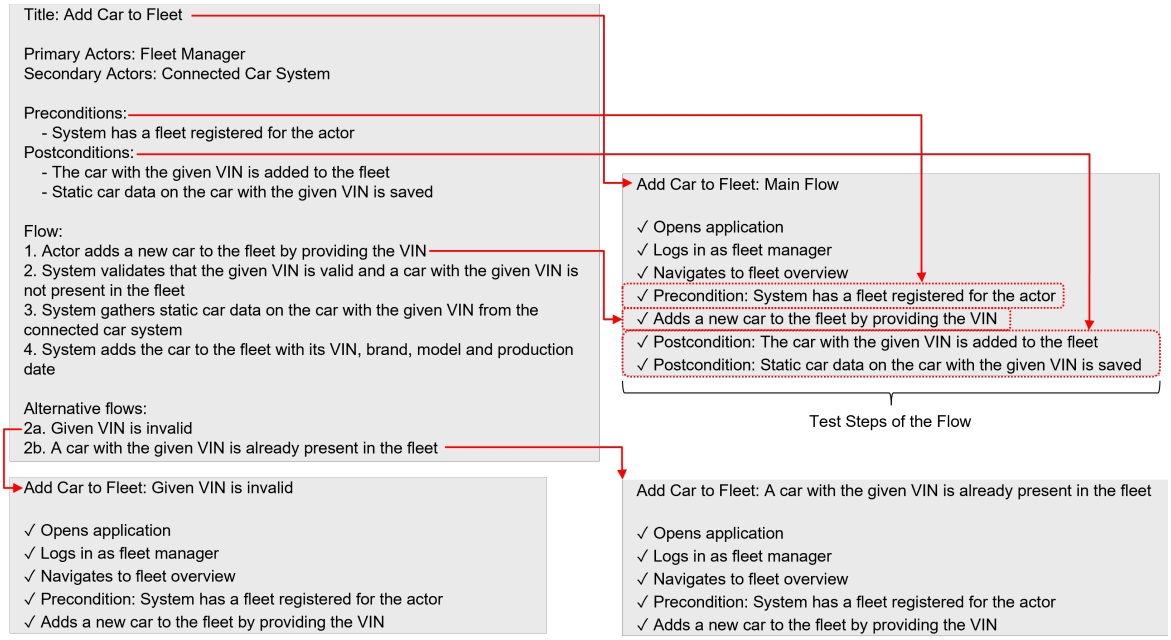


Abbildung 10.2: Beispielhafte Ableitung der Testschritte aus dem Anwendunngsfall " Add Car to Fleet"

Überführung der Ende-zu-Ende-Tests in die Implementierung

Anhand des Anwendungsfall "Add Car to Fleet" der CCSApp wird die Umsetzung des Ende-zu-Ende-Konzepts demonstriert. Als Werkzeug wird Cypress [Cyp-Jav], ein JavaScript-basiertes Testframeworks verwendet. Die Ende-zu-Ende-Tests werden wie vom Testkonzept vorgesehen von der Implementierung der Benutzerschnittstelle entkoppelt, indem ein eigenes Test-Repository verwendet wird. Dieser Ansatz sorgt dafür, dass das gesamte System als Blackbox betrachtet wird und ermöglicht die Auswahl von Implementierungswerkzeugen und Technologien unabhängig von der ausgewählten Technologie der Benutzerschnittstelle. In Listing 10.1 ist der Hauptfluss des Anwendungsfalls "Add Car to Fleet" umgesetzt.

Der Titel des Testfall wird als Parameter für die Cypress-Funktion "describe" in Zeile 1 angegeben. Jeder Testschritt enthält eine textuelle Beschreibung, die als Parameter an die Cypress-Funktion "it" übergeben wird. Cypress übernimmt diese Funktionen von der BDD-Syntax, die in der gebündelten Bibliothek Mocha [Cyp-Moc] [Moc-Moc] enthalten ist. Die Testinitialisierungsschritte sind in den Zeilen 2-5 angegeben. Die ersten drei Testschritte (Zeile 2-4) implementieren die Navigationsschritte der Benutzerschnittstelle. Diese Navigationsschritte sind von der Benutzeroberfläche abhängig und werden nicht durch den Anwendungsfall abgedeckt. Um die Wiederverwendbarkeit zu gewährleisten, werden die Funktionen in einem Paket "common" abgelegt. Dieses Paket beinhaltet implementierte parametrisierte Testschritte, die nicht zu den Anwendungsfällen gehören. Die Vorbedingung finalisiert

die Testinitialisierung (Zeile 5) und ruft die Funktion hasFleet() auf. Diese initialisiert für den Flottenmanager eine Flotte, zu der neue Fahrzeuge hinzugefügt werden können.

```
1 describe('Add Car to Fleet: Main Flow', () =>
2     it('Opens application', common.openApplication())
3     it('Logs in as fleet manager', common.logInAs(common.actors.
          FLEET_MANAGER))
4     it('Navigates to fleet overview', common.navigateTo(common.pages.
          FLEET_OVERVIEW))
5     it('Precondition: System has a fleet registered for the actor',
          flows.hasFleet())
6     it('Adds a new car to the fleet by providing the VIN', flows.
          addCarWithValidVIN())
7     it('Postcondition: The car with the given VIN is added to the fleet
          ', () =>
8         cy.contains('td', flows.validVIN).should('exist')
9     )
10    it('Postcondition: Static car data on the car with the given VIN is
           saved', () =>
11        cy.contains('td', flows.validVIN).should('exist')
12        ...
13    )
14 )
```

Listing 10.1: Umgesetzte Testfälle für den Anwendungsfall Add Car to Fleet

Wie bei der Systematik angegeben werden nachfolgend alle Schritte überführt, die den Akteur betreffen. Im betrachteten Anwendungsfall ist der Flottenmanager nur in Schritt 1 "Actor adds a new car to the fleet by providing the VIN" aktiv. Dieser Schritt des Anwendungsfalls wird in Zeile 6 durch die Funktion addCarWithValidVIN() implementiert. Im Anschluss an die Implementierung aller von dem Akteur beteiligten Schritten wird die Nachbedingung (Zeilen 7-13) validiert. Da die Nachbedingungen für jeden Anwendungsfall spezifisch sind, werden die Behauptungen (engl. assertions) direkt in den Testschritten implementiert, ohne die Logik in eine andere Funktion auszulagern.

Die Implementierung des Testschritts addCarWithValidVIN (siehe Zeile 6) wird in Listing 10.2 genauer dargestellt.

```
1 export function addCarWithValidVIN() {
2     return () => {
3         cy.contains('td', validVIN)
4             .siblings()
5             .contains('delete')
6             .click()
7         cy.contains('td', validVIN).should('not.exist')
8         cy.get('#vinInput').type(validVIN)
```

```
9           cy.get('#addCarToFleetButton').click()
10      }
11  }
```

Listing 10.2: Konkrete Implementierung der Funktion addCarWithValidVIN()

Die in Listing 10.1 definierte Schritte und deren Funktionen sind bei Cypress nur innerhalb einer "it"-Anweisung ausführbar. Die Funktion addCarWithValidVIN() gibt daher eine Funktion zurück (Zeile 2), welche an die aufrufende Funktion delegiert wird. In Zeile 3-9 werden die verschiedenen Aktionen des Akteurs anhand der Benutzeroberfläche simuliert. Da das Auto nicht existieren darf, wird zuerst sichergestellt, dass dass das Fahrzeug nicht existiert (Zeile 3-6), indem die vorhandenen Autos überprüft (und falls vorhanden) entfernt wird. Die anschließende Behauptung (Zeile 7) überprüft, dass kein Fahrzeug mit der gewünschten Fahrgestellnummer vorhanden ist. Anschließend wird die Eingabe des Akteurs simuliert und die Fahrgestellnummer eingegeben. Über eine Schaltfläche der Benutzeroberfläche wird die Eingabe letztendlich bestätigt. Die weiteren Testschritte aus Abschnitt 10.1 werden analog in die Implementierung überführt.

10.4 Abbildungsverzeichnis

10.5 Tabellenverzeichnis

10.6 Quelltextverzeichnis

10.7 Literaturverzeichnis

[AS+19] Sebastian Abeck, Michael Schneider, Jan-Philip Quirmbach, Heiko Klarl, Christof Urbaczek, Shkodran Zogaj: A Context Map as the Basis for a Microservice Architecture for the Connected Car Domain, INFORMATIK 2019: 50 Jahre Gesellschaft für Informatik–Informatik für Gesellschaft, 2019.

[BB+10] Imran Sarwar Bajwa, Behzad Bordbar, Mark G Lee: OCL Constraints Generation from Natural Language Specification, in 2010 14th IEEE International Enterprise Distributed Object Computing Conference, IEEE, pages 204–213, 2010.

[BW+15] Len Bass, Ingo Weber, Liming Zhu: DevOps: A software architect's perspective, Addison-Wesley Professional, 2015.

[BN+19] Christoph Binder, Christian Neureiter, Goran Lastro: Towards a Model-Driven Architecture Process for Developing Industry 4.0 Applications, International Journal of Modeling and Optimization, vol. 9, no. 1, pages 1–6, 2019.

[BH+04] Marc Born, Eckhardt Holz, Olaf Kath: Softwareentwicklung mit UML 2: die"neuenËntwurfstechniken UML 2, MOF 2 und MDA, Pearson Deutschland GmbH, 2004.

[BK15] Hans Brandt-Pook, Rainer Kollmeier: Softwareentwicklung kompakt und verständlich, Springer, 2015.

[BD09] Bernd Bruegge, Allen H Dutoit: Object-Oriented Software Engineering. UIsing UML, Patterns, and Java, Learning, vol. 5, no. 6, page 7, 2009.

[CT+05] David C Chou, Hima Bindu Tripuramallu, Amy Y Chou: BI and ERP integration, Information management & computer security, vol. 13, no. 5, pages 340–349, 2005.

[CN+12] Lawrence Chung, Brian A Nixon, Eric Yu, John Mylopoulos: Non-Functional Requirements in Software Engineering, vol. 5, Springer Science & Business Media, 2012.

[Cl14] Toby Clemson: Testing Strategies in a Microservice Architecture, November 2014, vol. 30, 2014.

[CMU-SA] CMU: Software Architecture Getting Started Glossary Modern Software Architecture Definitions, URL `https://www.sei.cmu.edu/architecture/start/glossary/moderndefs.cfm`, 2010.

[Co01] Alistair Cockburn: Writing Effective Use Cases, Boston: Addison-Wesley, 2001.

[Co05] Alistair Cockburn: Hexagonal Architecture, alistair. cockburn. us, 2005.

[Co10] Mike Cohn: Succeeding with agile: software development using Scrum, Pearson Education, 2010.

[Cuc-BDD] Cucumber: BDD Testing and Collaboration Tools for Teams, https://cucumber.io/.

[Cyp-Moc] Cypress: Bundled Libraries: Mocha, https://docs.cypress.io/guides/references/bundled-libraries#Mocha.

[Cyp-Jav] Cypress: JavaScript End to End Testing Framework, https://www.cypress.io/.

[DK+18] Soumya Kanti Datta, Mohammad Irfan Khan, Lara Codeca, Benoît Denis, Jérôme Härri, Christian Bonnet: IoT and Microservices Based Testbed for Connected Car Services, in 2018 IEEE 19th International Symposium onÄ World of Wireless, Mobile and Multimedia Networks"(WoWMoM), IEEE, pages 14–19, 2018.

[Da03] Andrew Davies: Integrated Solutions: The Changing Business of Systems Integration, The business of systems integration, pages 333–368, 2003.

[De17] Brajesh De: API Management: An Architect's Guide to Developing and Managing APIs for Your Organization, Apress, ISBN 978-1-4842-1306-3 978-1-4842-1305-6, URL http://link.springer.com/10.1007/978-1-4842-1305-6, 2017.

[MK+17] Arthur M Del Esposte, Fabio Kon, Fabio M Costa, Nelson Lago: InterSCity: A Scalable Microservice-based Open Source Platform for Smart Cities., in SMARTGREENS, pages 35–46, 2017.

[Git-Doc] Gitlab Documentation: Multi-project pipeline, URL https://docs.gitlab.com/ee/ci/pipelines/downstream_pipelines.html#multi-project-pipelines.

[Do15] Wolfgang Dorst: Umsetzungsstrategie Industrie 4.0: Ergebnisbericht der Plattform Industrie 4.0, Bitkom Research GmbH, 2015.

[DG+17] Nicola Dragoni, Saverio Giallorenzo, Alberto Lluch Lafuente, Manuel Mazzara, Fabrizio Montesi, Ruslan Mustafin, Larisa Safina: Microservices: Yesterday, Today, and Tomorrow, Present and ulterior software engineering, pages 195–216, 2017.

[Du16] Zoya Durdik: Architectural Design Decision Documentation through Reuse of Design Patterns, vol. 14, KIT Scientific Publishing, 2016.

[DG+13] Schahram Dustdar, Harald Gall, Manfred Hauswirth: Software-Architekturen für verteilte Systeme: Prinzipien, Bausteine und Standardarchitekturen für moderne Software, Springer-Verlag, 2013.

[Eb19] Christof Ebert: Systematisches Requirements Engineering: Anforderungen ermitteln, dokumentieren, analysieren und verwalten, dpunkt. verlag, 2019.

[EW+17] Nico Ebert, Kristin Weber, Stefan Koruna: Integration Platform as a Service, Business & Information Systems Engineering, vol. 59, no. 5, pages 375–379, 2017.

[EB+11] Brian Elvesæter, Arne-Jørgen Berre, Andrey Sadovykh: Specifying Services using the Service Oriented Architecture Modeling Language (SoaML) - A Baseline for Specification of Cloud-based Services, in CLOSER, pages 276–285, 2011.

[OAP-Ope] Jamie Tanna et al.: OpenAPI Client and Server Code Generator, URL `https://github.com/deepmap/oapi-codegen`.

[Ev04] Eric Evans: Domain-Driven Design: Tackling Complexity in the Heart of Software, Addison-Wesley Professional, 2004.

[Ev14] Eric Evans: Domain-Driven Design Reference: Definitions and Pattern Summaries, Dog Ear Publishing, 2014.

[FS19] David Faragó, Dehla Sokenou: Keynote: Microservices Testen Erfahrungsbericht und Umfrage, Test, Analyse und Verifikation von Software (TAV) der Gesellschaft für Informatik (GI), Stuttgart, 2019.

[Fi00] Roy Thomas Fielding: REST: Architectural Styles and the Design of Network-based Software Architectures, Ph.D. thesis, University of California, 2000.

[Fo12] Martin Fowler: Test Pyramid, Accessed on November 2017 https://martinfowler.com/bliki, TestPyramid. html, 2012.

[FM+17] Paolo Di Francesco, Ivano Malavolta, Patricia Lago: Research on Architecting Microservices: Trends, Focus, and Potential for Industrial Adoption, in 2017 IEEE International Conference on Software Architecture (ICSA), pages 21–30, 2017.

[Fr15b] Klaus Franz: Begriffe zum Testen, Springer Berlin Heidelberg, Berlin, Heidelberg, ISBN 978-3-662-44028-5, pages 31–36, URL `https://doi.org/10.1007/978-3-662-44028-5_3`, 2015.

[Fr15] Paul Fremantle: A Reference Architecture for the Internet of Things, WSO2 White paper, URL `https://docs.huihoo.com/wso2/wso2-whitepaper-a-reference-architecture-for-the-internet-of-things.pdf`, 2015.

[GW+19] Israr Ghani, Wan MN Wan-Kadir, Ahmad Mustafa, Muhammad Imran Babir: Microservice Testing Approaches: A Systematic Literature Review, International Journal of Integrated Engineering, vol. 11, no. 8, pages 65–80, 2019.

[Gi18] Pascal Giessler: Domain-Driven Design of Resource-oriented Microservices (in German), Ph.D. thesis, Karlsruhe Institute of Technology (KIT), Cooperation & Management (C&M), 2018.

[GH+18] Pascal Giessler, Benjamin Hippchen, Michael Schneider, Sebastian Abeck: Wiederverwendbare Microservices durch Domain-Driven Design, in Objektspektrum, 2018.

[Gl13] Martin Glinz: A Glossary of Requirements Engineering Terminology, Standard Glossary of the Certified Professional for Requirements Engineering (CPRE) Studies and Exam, Version, vol. 1, page 56, 2011.

[GF94] O.C.Z. Gotel, C.W. Finkelstein: An analysis of the requirements traceability problem, in Proceedings of IEEE International Conference on Requirements Engineering, pages 94–101, 1994.

[OMG-OCL] Object Management Group: Object Constraint Language, URL `https://www.omg.org/spec/OCL/2.4`.

[W3C-WoT] W3C Working Group: Web of Things, URL `https://www.w3.org/WoT/documentation/`.

[GR+17] Javier J Gutiérrez, Isabel Ramos, Manuel Mejías, Carlos Arévalo, Juan M Sánchez-Begines, David Lizcano: Modelling Gherkin Scenarios Using UML, 2017.

[HP18] Olaf Hartig, Jorge Pérez: Semantics and Complexity of GraphQL, in Proceedings of the 2018 World Wide Web Conference, pages 1155–1164, 2018.

[HS17] Wilhelm Hasselbring, Guido Steinacker: Microservice Architectures for Scalability, Agility and Reliability in E-Commerce, in 2017 IEEE International Conference on Software Architecture Workshops (ICSAW), IEEE, pages 243–246, 2017.

[HH+19] Philipp Hertweck, Tobias Hellmund, Hylke van der Schaaf, Jürgen Moßgraber, Jan-Wilhelm Blume: Management of Sensor Data with Open Standards., in ISCRAM, 2019.

[Hi21] Benjamin Hippchen: Iterative Migration monolithischer Softwaresysteme in eine domänengetriebene Microservice-Architektur, Ph.D. thesis, Karlsruher Institut für Technologie (KIT), 2021.

[HG+17] Benjamin Hippchen, Pascal Giessler, Roland Steinegger, Michael Schneider, Sebastian Abeck: Designing Microservice-based Applications by Using a Domain-Driven Design Approach, International Journal on Advances in Software, vol. 10, no. 3&4, pages 432–445, 2017.

[HS+19] Benjamin Hippchen, Michael Schneider, Pascal Giessler, Sebastian Abeck: Systematic Application of Domain-Driven Design for a Business-Driven Microservice Architecture, International Journal on Advances in Software Volume 12, Number 3 & 4, 2019, 2019.

[IK20] Kasun Indrasiri, Danesh Kuruppu: gRPC: Up & Running - Building Cloud Native Applications with Go and Java for Docker and Kubernetes, O'Reilly Media, ISBN 978-1-4920-5833-5, URL `http://gen.lib.rus.ec/book/index.php?md5=A7098600FFF8A5FAA7A647C913167AB9`, 2020.

[OAI-Wha] Open API Initiative: FAQ - What is OpenAPI, URL `https://www.openapis.org/faq`.

[ITU-T12] International Telecommunication Union (ITU): Overview of the Internet of Things, Standard, URL `https://www.itu.int/rec/T-REC-Y.2060-201206-I`, 2012.

[Ja17] Nic Jackson: Building Microservices with Go, Packt Publishing Ltd, 2017.

[JC+92] Ivar Jacobson, Magnus Christerson, Patrik Jonsson, Gunnar Overgaard: Object-Oriented Software Engineering: A Use Case Driven Approach, Addison Wesley Longman Publishing Co., Inc., USA, ISBN 0201403471, 2004.

[Ju06] Matjaz B Juric: A Hands-on Introduction to BPEL, Oracle (white paper), page 21, https://www.oracle.com/technical-resources/articles/matjaz-bpel.html, 2006.

[Ka03] Doug Kaye: Loosely coupled: the missing pieces of Web services, RDS Strategies LLC, 2003.

[KM05] Martin Kempa, Zoltán Adám Mann: Model driven architecture, Informatik-Spektrum, vol. 28, pages 298–302, 2005.

[Kh20] V Khorikov: Unit Testing Principles, Practices, and Patterns, 2020.

[He08] Heiko Koziolek: Parameter Dependencies for Reusable Performance Specifications of Software Components, Ph.D. thesis, Universität Oldenburg, 2008.

[KJ+15] Alexandr Krylovskiy, Marco Jahn, Edoardo Patti: Designing a Smart City Internet of Things Platform with Microservice Architecture, in 2015 3rd International Conference on Future Internet of Things and Cloud, IEEE, pages 25–30, 2015.

[La04] Craig Larman: Applying UML and Patterns: An Introduction to Object-Oriented Analysis and Design and Iterative Development, Prentice Hall PTR, 2004.

[LM+19] Jyri Lehvä, Niko Mäkitalo, Tommi Mikkonen: Consumer-Driven Contract Tests for Microservices: A Case Study, in International Conference on Product-Focused Software Process Improvement, 2019.

[FL14] James Lewis, Martin Fowler: Microservices: A Definition of This new Architectural Term, URL: https://martinfowler.com/articles/microservices.html, vol. 25, pages 14–26, 2014.

[LF14] James Lewis, Martin Fowler: Microservices: a definition of this new architectural term, MartinFowler. com, vol. 25, pages 14–26, 2014.

[MT+20] Niko Mäkitalo, Antero Taivalsaari, Arto Kiviluoto, Tommi Mikkonen, Rafael Capilla: On Opportunistic Software Reuse, Computing, vol. 102, pages 2385–2408, 2020.

[MB+01] Ruth Malan, Dana Bredemeyer, et al.: Functional Requirements and Use Cases, Bredemeyer Consulting, 2001.

[MP+10] Maria Maleshkova, Carlos Pedrinaci, John Domingue: Investigating Web APIs on the World Wide Web, in 2010 Eighth IEEE European Conference on Web Services, pages 107–114, 2010.

[Ma09] Robert C Martin: Clean Code: A Handbook of Agile Software Craftsmanship, Pearson Education, 2009.

[Ma12] Robert C Martin: Clean Architecture, URL `https://blog.cleancoder.com/uncle-bob/2012/08/13/the-clean-architecture.html`, 2012.

[MS+02] Stephen J Mellor, Kendall Scott, Axel Uhl, Dirk Weise: Model-Driven Architecture, in Advances in Object-Oriented Information Systems: OOIS 2002 Workshops Montpellier, France, September 2, 2002 Proceedings, Springer, pages 290–297, 2002.

[Moc-Moc] Mocha: Mocha: The fun, simple, flexible JavaScript test framework, `https://mochajs.org/`.

[Mul-API-led] MuleSoft: API-led connectivity - The next step in the evolution of SOA, `https://www.mulesoft.com/lp/whitepaper/api/api-led-connectivity`, 2022.

[MA+17] Lauren Murphy, Tosin Alliyu, Andrew Macvean, Mary Beth Kery, Brad A Myers: Preliminary Analysis of REST API Style Guidelines, Ann Arbor, vol. 1001, page 48109, 2017.

[BH+13] Youssef Hassoun Mustafa Bozkurt, Mark Harman: Testing and verification in service-oriented architecture: a survey, Software Testing, Verification and Reliability, vol. 23, no. 4, pages 261–313, 2013.

[NM+16] Irakli Nadareishvili, Ronnie Mitra, Matt McLarty, Mike Amundsen: Microservice Architecture: Aligning Principles, Practices, and Culture, Ö'Reilly Media, Inc.", 2016.

[Ne15] Sam Newman: Building Microservices: Designing fine-grained Systems, Ö'Reilly Media, Inc.", 2015.

[No06] Dan North, et al.: Introducing BDD, Better Software Magazine, vol. 12, 2006.

[OGC-STA-Sen] Open Geospatial Consortium (OGC): OGC SensorThings API Part 1: Sensing, Standard, URL `http://docs.opengeospatial.org/is/15-078r6/15-078r6.html`, 2016.

[OGC-STA-Tas] Open Geospatial Consortium (OGC): OGC SensorThings API Part 2: Tasking Core, Standard, URL `http://docs.opengeospatial.org/is/17-079r1/17-079r1.html`, 2016.

[Ol03] Michael Olan: Unit Testing: Test Early, Test Often, Journal of Computing Sciences in Colleges, vol. 19, no. 2, pages 319–328, 2003.

[Re19] Gerard O'Regan: Concise guide to software testing, Springer, 2019.

[Pac-Doc] Pact Foundation: Pact Documentation, `https://docs.pact.io/`, 2022.

[Po10] Klaus Pohl: Requirements Engineering: Fundamentals, Principles, and Techniques, Springer Publishing Company, Incorporated, 2010.

[Sem-Sem] Tom Preston-Werner, et al.: Semantic Versioning 2.0. 0, `https://semver.org/`, 2013.

[RS+17] Florian Rademacher, Sabine Sachweh, Albert Zündorf: Towards a UML Profile for Domain-Driven Design of Microservice Architectures, in International Conference on Software Engineering and Formal Methods, Springer, pages 230–245, 2017.

[RS+18] Florian Rademacher, Jonas Sorgalla, Sabine Sachweh: Challenges of Domain-Driven Microservice Design: A Model-Driven Perspective, IEEE Software, vol. 35, no. 3, pages 36–43, 2018.

[RG15] Mazedur Rahman, Jerry Gao: A reusable automated acceptance testing architecture for microservices in behavior-driven development, in 2015 IEEE Symposium on Service-Oriented System Engineering, IEEE, pages 321–325, 2015.

[RJ20] Casey Rosenthal, Nora Jones: Chaos Engineering: System Resiliency in Practice, O'Reilly Media, 2020.

[RQ+12] Chris Rupp, Stefan Queins, et al.: UML 2 glasklar: Praxiswissen für die UML-Modellierung, Carl Hanser Verlag GmbH Co KG, 2012.

[SR+18] Dmitrii Savchenko, Gleb Radchenko, Timo Hynninen, Ossi Taipale: Microservice Test Process: Design and Implementation, in International Journal on Information Technologies and Security, 2018.

[Sc02] Walt Scacchi: Process Models in Software Engineering, Encyclopedia of Software Engineering, 2002.

[SA23] Michael Schneider, Sebastian Abeck: Engineering Microservice-Based Applications Using an Integration Platform as a Service, in 2023 IEEE International Conference on Service-Oriented System Engineering (SOSE), pages 124–129, 2023.

[SA+23] Michael Schneider, Sebastian Abeck: Engineering Microservice-Based Applications Using an Integration Platform as a Service, International Congress on Intelligent and Service-Oriented Systems Engineering, 2023.

[SH+18] Michael Schneider, Benjamin Hippchen, Sebastian Abeck, Michael Jacoby, Reinhard Herzog: Enabling IoT Platform Interoperability Using a Systematic Development Approach by Example, in 3rd International Workshop on Interoperability and Open-Source Solutions for the Internet of Things (InterOSS-IoT), 2018.

[SH+19] Michael Schneider, Benjamin Hippchen, Pascal Giessler, Chris Irrgang, Sebastian Abeck: Microservice Development Based on Tool-Supported Domain Modeling, in Conference on Advances and Trends in Software Engineering (SOFTENG), 2019.

[SS+19] Boris Scholl, Trent Swanson, Peter Jausovec: Cloud Native: Using Containers, Functions, and Data to Build Next-Generation Applications, Ö'Reilly Media, Inc.", 2019.

[SS09] Yashwant Singh, Manu Sood: Model driven architecture: A perspective, in 2009 IEEE International Advance Computing Conference, IEEE, pages 1644–1652, 2009.

[Sm14] John Ferguson Smart: BDD in Action, Manning Publications, 2014.

[BH+14] IC Society, P Bourque, RE Fairley: Guide to the software engineering body of knowledge (SWEBOK (r)): version 3.0, 2014.

[ANS829] IEEE Computer Society: IEEE Standard for Software and System Test Documentation, IEEE Std 829-2008, pages 1–150, 2008.

[St73] Herbert Stachowiak: Allgemeine Modelltheorie, Springer, 1973.

[SL+17] Long Sun, Yan Li, Raheel Ahmed Memon: An open IoT framework based on microservices architecture, China Communications, vol. 14, no. 2, pages 154–162, 2017.

[Ti00] Walter F Tichy: Hints for Reviewing Empirical Work in Software Engineering, Empirical Software Engineering, vol. 5, no. 4, pages 309–312, 2000.

[TS05] Nikolai Tillmann, Wolfram Schulte: Parameterized Unit Tests, ACM SIGSOFT Software Engineering Notes, vol. 30, no. 5, pages 253–262, 2005.

[Tu19] Nick Tune: EventStorming Modelling Tips to Facilitate Microservice Design, URL `https://medium.com/nick-tune-tech-strategy-blog/eventstorming-modelling-tips-to-facilitate-microservice-design-1b1b0b838efc`.

[Fra-Fro] Hylke van der Schaaf, Michael Jacoby: FROST-Server, `https://github.com/FraunhoferIOSB/FROST-Server`, 2022.

[Ve13] Vaughn Vernon (editor): Implementing Domain-Driven Design, Addison-Wesley, 2013.

[WL+20] Muhammad Waseem, Peng Liang, Gastón Márquez, Amleto Di Salle: Testing Microservices Architecture-Based Applications: A Systematic Mapping Study, in 2020 27th Asia-Pacific Software Engineering Conference (APSEC), pages 119–128, 2020.

[Wh00] J.A. Whittaker: What is software testing? And why is it so hard?, IEEE Software, vol. 17, no. 1, pages 70–79, 2000.

[Wi16] Frank Witte: Testumgebung, in Testmanagement und Softwaretest, Springer, pages 119–122, 2016.

[WR+12] Claes Wohlin, Per Runeson, Martin Höst, Magnus C Ohlsson, Björn Regnell, Anders Wesslén: Experimentation in Software Engineering, Springer Science & Business Media, 2012.

[WH12] Matt Wynne, Aslak Hellesøy: The Cucumber Book: Behaviour-Driven Development for Testers and Developers, Pragmatic Bookshelf, ISBN 9781934356807, 2012.

[Ya20] Naren Yellavula: Hands-On RESTful Web Services with Go: Develop Elegant RESTful APIs with Golang for Microservices and the Cloud, Packt Publishing Ltd, 2020.

[Zi17] Olaf Zimmermann: Microservices Tenets: Agile Approach to Service Development and Deployment, Computer Science-Research and Development, vol. 32, pages 301–310, 2017.

10.8 Publikationsliste

[HG+17] Benjamin Hippchen, Pascal Giessler, Roland H. Steinegger, Michael Schneider, Sebastian Abeck: Designing Microservice-Based Applications by Using a Domain-Driven Design Approach. International Journal on Advances in Software, vol. 10 (432 – 445), 2017.

[SH+18] Michael Schneider, Benjamin Hippchen, Sebastian Abeck, Michael Jacoby, Reinhard Herzog: Ënabling IoT Platform Interoperability Using a Systematic Development Approach by Example", Global Internet of Things Summit (GIoTS), 2018.

[HS+19] Benjamin Hippchen, Michael Schneider, Pascal Giessler, Iris Landerer: Methodology for Splitting Business Capabilities into a Microservice Architecture: Design and Maintenance Using a Domain-Driven Approach, The Fifth International Conference on Advances and Trends in Software Engineering (SOFTENG), 2019.

[SH+19] Michael Schneider, Benjamin Hippchen, Pascal Giessler, Chris Irrgang: Microservice Development Based on Tool-Supported Domain Modeling, The Fifth International Conference on Advances and Trends in Software Engineering (SOFTENG), 2019.

[SZ+21] Michael Schneider, Stephanie Zieschinski, Hristo Klechorov, Lukas Brosch, Patrick Schorsten, Sebastian Abeck, Christof Urbaczek: A Test Concept for the Development of Microservice-based Applications, The Sixteenth International Conference on Software Engineering Advances (ICSEA 2021), Barcelona, 2021.

[SA23] Michael Schneider, Sebastian Abeck: Engineering Microservice-Based Applications Using an Integration Platform as a Service, IEEE International Conference on Service-Oriented System Engineering (SOSE), Athens, 2023.