

GESELLSCHAFT
FÜR INFORMATIK



Raphael Sack, Lukas Epple (Hrsg.)

Seminar Software-Engineering HOR-TINF20

**13.12.2022
Horb am Neckar, Deutschland**

Duale Hochschule Baden-Württemberg Stuttgart, Campus Horb

Vorwort

Im Rahmen des Seminars "Advanced Software Engineering" des Kurses TINF2020 an der Dualen Hochschule Baden-Württemberg Stuttgart, Campus Horb wird ein Kongress zur Präsentation und Diskussion abgehalten.

Es werden mehrere Arbeiten im Zusammenhang mit dem Themenkomplex "Software Engineering" vorgetragen. Dabei werden unterschiedlichste Aspekte, wie beispielsweise Reusability, Auswirkungen von architektonischen, infrastrukturellen Entscheidungen und Werkzeugauswahl, sowie Usability, Resilienz und Nachhaltigkeit beleuchtet.

Der vorliegende Tagungsband enthält 10 wissenschaftliche Beiträge, die von unterschiedlichen Gruppen, bestehend aus Teilnehmern des Kurses TINF2020 erarbeitet wurden.

Der Kongress soll eine Plattform bieten, die erarbeiteten Beiträge zu diskutieren und Wissen auszutauschen. Unser Dank gilt allen, die sich aktiv an der Vorbereitung und Durchführung des Kongresses beteiligt haben.

Horb am Neckar, Dezember 2022

Epple, Lukas

Inhaltsverzeichnis

Hauptvorträge

Kongresstag 1

Ingmar Bauckhage, Vsevolod Pypenko	
<i>Zusammenarbeit in der Software-Entwicklung</i>	11
Adrian Liehner, Jülf Freudenberger	
<i>Software Reuse</i>	37
Fabian Klimpel, Lukas Epple, Raphael Sack	
<i>Auswirkungen von SOA</i>	61
Gabriel Sperling, Reinhold Jooß	
<i>Infrastructure as Code</i>	91

Kongresstag 2

Jonathan Schwab, Felix Wochele, Jonas Weis	
<i>Intelligente Werkzeuge SWE</i>	117
Jannik Dürr, Dominic Joas, Ruben Kalmbach	
<i>Hybride Applikationen</i>	151
Alicia Dietrich, Nick Dürr, Jan Perthel	
<i>Untersuchung von Web-Usability Prinzipien und Evaluation zweier Webshops</i>	185
Niklas Arnold, Luca Negron-Martinez	
<i>Everything-as-a-Service</i>	229
Robin Epple, Timo Vollert	
<i>Resilienz in Systemarchitekturen</i>	249

Fabian Heinl, Philipp Kremling

Green IT

267

Autorenverzeichnis

Hauptvorträge

Kongressstag 1

Varianten und Herausforderungen der Zusammenarbeit in der Software-Entwicklung - Ein Vergleich verschiedener Herangehensweisen und Werkzeuge

Ingmar Bauckhage¹, Vsevolod Pypenko²

Abstract: Zusammenarbeit ist bei der Entwicklung von Software-Projekten von großer Bedeutung. Dieser Beitrag vergleicht vier Vorgehensmodelle in Bezug auf die Zusammenarbeit: Wasserfall-Modell, V-Modell, Rational Unified Process und Scrum. Es werden Unterschiede herausgearbeitet sowie Vor- und Nachteile benannt. Die Zusammenarbeit während der Implementierung und des Qualitätsmanagements wird näher betrachtet.

Herausforderungen sind z. B. die gemeinsame Arbeit am Quellcode. Hier existieren verschiedene Workflows für die Werkzeuge Git und Github, die veranschaulicht und bewertet werden.

Im Qualitätsmanagement ist vor allem die Zusammenarbeit von verschiedenen Teams notwendig. Dies kann über regelmäßigen Austausch mithilfe von formalen Dokumenten und Meetings sichergestellt werden. Darüber hinaus wird beschrieben, welche verschiedenen Möglichkeiten der Verifikation und des Testens es gibt und für welche Projekte diese notwendig sind.

Keywords: Collaboration; Zusammenarbeit; Qualitätsmanagement; Quellcode-Verwaltung; Wasserfall-Modell; V-Modell; RUP; Scrum; Git; Branching

1 Einführung

Die Entwicklung von großen Software-Projekten kann nur durch Zusammenarbeit gelingen. Dies ist vor allem durch die steigende Komplexität moderner Software bedingt [Wi22]. Um die steigende Komplexität zu beherrschen haben sich eine Vielzahl an Vorgehensmodellen, Strategien und Werkzeugen entwickelt. Welche Unterschiede bestehen dabei zwischen den einzelnen Modellen in Bezug auf die Zusammenarbeit und welche Empfehlungen können für die Wahl ausgesprochen werden?

1.1 Motivation

Wie soll der Begriff *groß* im Rahmen dieser Arbeit verstanden werden? Ein *großes* Software-Projekt soll hier insbesondere durch die Anzahl an Entwicklerinnen und Entwickler gekennzeichnet sein. Ein konkreter Vorschlag wäre, ein Software-Projekt *groß* zu

¹ DHBW Stuttgart Campus Horb, TINF2020, Florianstraße 15, 72160 Horb am Neckar, Deutschland i20003@hb.dhbw-stuttgart.de

² DHBW Stuttgart Campus Horb, TINF2020, Florianstraße 15, 72160 Horb am Neckar, Deutschland i20028@hb.dhbw-stuttgart.de

nennen, wenn mehr als drei Entwicklerinnen und Entwickler zusammenarbeiten. Dies ist eine recht willkürliche Festlegung, entscheidend soll aber insbesondere die Abgrenzung zur Einzel-Entwicklung sein.

Auch in kleineren und mittleren Unternehmen steigen die Anforderungen an Software, während die Anzahl der Entwicklerinnen und Entwickler meist nicht so schnell gesteigert werden kann. Dadurch fehlen dedizierte Mitarbeiterinnen oder Mitarbeiter für das Management der Zusammenarbeit, während in größeren Unternehmen oft komplett Abteilungen alleine hierfür zuständig sind. In der Folge kommt es zu Problemen in der Zusammenarbeit und daraus resultierend einer geringeren Produktivität oder Fehler in der Software.

Die Zusammenarbeit im Team während eines Projekts ist entscheidend für den Ausgang des Projektes. Dabei spielt die Kommunikation zwischen den Teammitgliedern eine enorme Rolle, vor allem je größer das Projekt und somit die Anzahl der Beteiligten ist [Ve00]. Um diese Herausforderung zu bewältigen haben sich verschiedene Modelle entwickelt, auf die im Folgenden kurz eingegangen wird. Außerdem kommt es bei der gemeinsamen Bearbeitung von Quellcode oft zu Konflikten, die aufwändig gelöst werden müssen. Hierfür sollen in dieser Arbeit Strategien vorgestellt werden, um diese Konflikte zu minimieren.

1.2 Vorgehensmodelle in der Software-Entwicklung

Vorgehensmodelle bieten die Möglichkeit ein Rahmenwerk mit festen Strukturen und Prozessen festzulegen, um häufig wiederkehrende Aufgaben bei der Software-Entwicklung zu erleichtern und zu standardisieren. Sie stellen im Grunde einen Plan dar, welcher den Entwicklungsprozess in überschaubare, zeitlich und inhaltlich begrenzte Phasen aufteilt und festlegt, wie die Übergänge zwischen den Phasen zu gestalten sind und welche Werkzeuge oder Werkzeug-Prototypen verwendet werden sollen. Auch die Arten der Zusammenarbeit sind hier mehr oder weniger festgelegt.

Dabei haben sich eine Vielzahl an konkurrierenden Modellen entwickelt, die von ihren jeweiligen Schöpfern angepriesen werden. Wenn eine grobe Aufteilung vorgenommen werden soll, kann unterschieden werden in die *klassischen* Vorgehensmodelle, die oft aus akademischer oder institutioneller Umgebung stammen und agile Methoden, die sich aufgrund von Problemen mit den bereits vorhandenen Vorgehensmodellen entwickelt haben. Die Klassifizierung von Vorgehensmodellen lässt sich noch weiter verfeinern. Für diese Arbeit reicht aber die übergeordnete Unterteilung und gegebenenfalls wird bei Bedarf direkt darauf eingegangen.

1.3 Zielsetzung

Diese Arbeit soll einen Überblick über die Zusammenarbeit in verschiedenen Vorgehensmodellen der Software-Entwicklung geben und Unterschiede herausarbeiten. Dabei sollen Vor- und Nachteile für verschiedene Einsatzzwecke aufgezeigt werden. Für die Phasen der Implementierung und der Qualitätssicherung soll außerdem eine nähere Betrachtung

erfolgen und Werkzeuge und Methoden vorgestellt und bewertet werden, die bei der Lösung der genannten Probleme helfen können.

2 Zusammenarbeit in ausgewählten Vorgehensmodellen

Im Folgenden werden die vier Vorgehensmodelle *Wasserfallmodell*, *V-Modell*, *Rational Unified Process* und *Scrum* in Bezug auf die Zusammenarbeit kurz vorgestellt und miteinander verglichen.

2.1 Wasserfall-Modell

Das Wasserfallmodell wurde im Jahr 1970 von Dr. Winston W.Royce in einem Paper vorgestellt. Somit ist dieses Modell auch einer der ältesten und bekannten Vorgehensmodellen in der Softwareentwicklung. Trotz des Alters findet das Modell immer noch Einsatz in vielen großen Unternehmen [AA15c][AA15b].

Das Vorgehensmodell stellt die Projektphasen in einem „Wasserfall“ dar. Dabei werden die einzelnen Phasen top-down nacheinander abgearbeitet und sind in sich abgeschlossen. Das heißt, dass bevor eine Phase komplett abgearbeitet wird, darf die Nächste nicht angefangen werden. Darüber hinaus lässt das reine Wasserfall-Modell keine Rückführung zu den vorherigen, abgeschlossenen Phasen, zu [Go11, S. 84].

Als Grundlage für den gesamten Entwicklungsprozess dient die Spezifikation. So werden auch die Anforderungen für Anwender von vornherein ermittelt und festgeschrieben. Die Ergebnisse einer Phase fallen in die nächste, um dort weiterverarbeitet zu werden. Zur Kommunikation zwischen den Entwicklern und Anwendern werden umfangreiche Dokumente über das Softwaresystem erstellt [Go11, S. 83]. Die wichtigsten Phasen des Modells lassen sich folgendermaßen definieren[AA15c, S. 85][Go11]:

1. Anforderungsdefinition: in Zusammenarbeit mit dem Kunden werden die Spezifikationen, Ziele und Anforderungen an den Softwareprodukt definiert. Anschließend wird eine umfassende Dokumentation erstellt, die die Grundlage der weiteren Arbeit bildet.
2. Software- und System-Entwurf: die Anforderungen werden bearbeitet und es werden abstrakte Softwarekomponenten sowie die grundlegende Softwarearchitektur modelliert
3. Implementierung: in dieser Phase werden die modellierten Software-Komponenten programmiert.
4. Integration und Testen: die einzelnen Elemente des Software-Systems werden zusammengeführt und es werden Tests durchgeführt.
5. Betrieb und Wartung: das erstellte Produkt wird zum Gebrauch freigegeben.

Das Wasserfallmodell, bereits in der einfachen Ausführung, bietet eine gute Möglichkeit, die Anwenderteams bei kleineren Projekten wie z.B. Prototypen, zu organisieren. Darüber hinaus lässt sich das Modell durch weitere Funktionalitäten erweitern. Zum Beispiel, um die Qualität der Software zu verbessern und eine höhere Flexibilität in die Projekte zu schaffen, wurde das sogenannte „Wasserfallmodell mit Rückführschleifen“ entwickelt. Dabei finden Validierungen und Verifikationen der vorherigen Phase beim Übergang in die nächste Phase statt und es lässt sich ein „Rückschritt“ über mehrere Phasen implementieren [Go11, S. 86, 87]. Die graphische Abbildung des oben beschriebenen Modells wird in Abbildung 1 dargestellt. Dabei wird das einfache Wasserfallmodell mit den roten Pfeilen und das erweiterte mit blauen dargestellt.

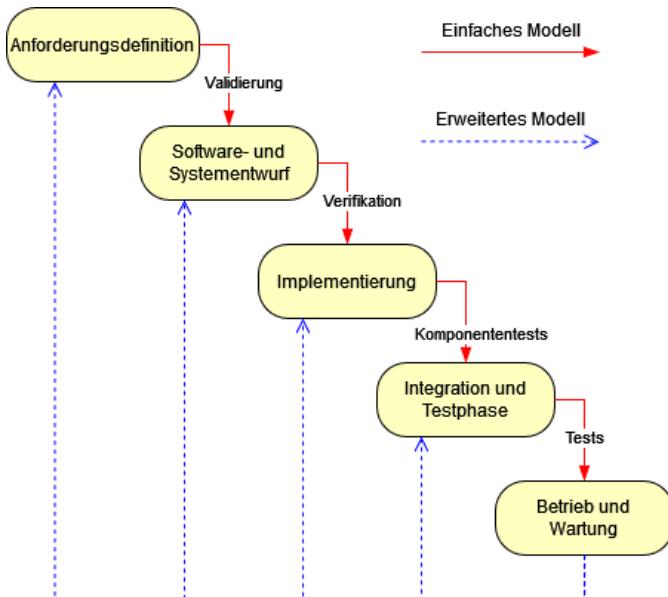


Abb. 1: Das einfache Wasserfallmodell

Die Vorteile des Wasserfallmodells liegen in der klaren, einfachen Struktur des Modells. Die Abläufe werden klar und verständlich in Phasen aufgeteilt wodurch sich die Ergebnisse kontrollieren lassen. Dadurch können die Aufgaben während einer Phase klar auf die Entwicklerteams aufgeteilt und verwaltet werden und jede Phase kann fehlerarm ablaufen. Darüber hinaus lässt sich das Modell mithilfe verschiedener Ansätze erweitern wodurch verschiedene Aspekte der Verwaltung und Softwareentwicklung, z.B. Qualitätssicherung verbessert werden [Go11, S. 84–87] [Ky19, S. 14].

Die Nachteile des Modells basieren zum einen auf der Starrheit und zum anderen auf dem viel zu großem Anteil der Dokumentation in dem Modell. Aufgrund der Starrheit lässt sich nicht so schnell auf die Veränderungen der Anforderungen und Spezifikation reagieren und dadurch können erhebliche Zeit- und Geldkosten anfallen. Dieses Problem

wird zwar teilweise durch das erweiterte Wasserfall Modell gelöst, jedoch genügt es nicht ganz, wenn die Anforderungen oft verändert werden müssen. Die Dokumentation, die umfänglich während des Projektablaufs geführt wird, kann auch unter Umständen einen viel zu großen Stellenwert im Projekt übernehmen, wodurch die eigentlichen Ziele nicht vollständig erreicht werden [Ky19, S. 14] [AA15c].

Bezogen auf die Zusammenarbeit, verfügt das Modell über zwei Ansätze: es findet ein Austausch zwischen Entwicklern innerhalb eines Teams während einer Phase und eine Kommunikation mithilfe von Dokumenten zwischen verschiedenen Phasen. Durch diese Ansätze kann zwar eine verständliche, fest vorgegebene Kommunikation geführt und eine Kontrolle angesetzt werden, aber bei größeren Projekten, kann dies zu einer langen Projektlaufzeit führen. Die Starrheit des Modells wird dabei nicht aufgelöst, was zu großen Problemen bei Anforderungsänderungen und anderen ungeplanten Geschehen führen kann.

2.2 V-Modell

Ein weiteres sequenzielles Modell, welches heutzutage häufig in deutschen Behörden und großen Unternehmen verwendet wird, ist das V-Modell [Hö08]. Wie der Name bereits bezeichnet, werden die einzelnen Projektpasen im Modell in Form eines „V“ dargestellt. Das V-Modell wurde als Weiterentwicklung des Wasserfallmodells entwickelt und liegt dabei den Fokus vor allem auf der Qualitätssicherung. Diese wird durch Einführung der Verifikation und Validierung mithilfe von Tests nach jeder Projektphase ermöglicht [AA15a][Va14].

Das V-Modell entstand bereits in 80er Jahren und stellte als Modell-Prototyp den Grundgerüst der modernen Vorgehensmodelle für die Bundeswehr in Deutschland her [DW99, S. 1]. Dieser Prototyp wurde im Laufe der Jahre bearbeitet und im Jahr 1992 für alle Bundesbehörden in Deutschland in einer verbindlichen Fassung als Rahmenregelung empfohlen [DW99, S. 3]. Diese Fassung beschrieb den Arbeitsprozess „statisch, als Abfolge einzelner Arbeitsschritte“ [DW99](S 3) wodurch dynamische Aspekte des Entwicklungsprozesses wie z.B. Kombinierung der verschiedenen Phasen miteinander, unbeschrieben wurden. Dieses und andere Probleme des Modells führten dazu, dass das Modell überarbeitet und weiterentwickelt werden musste. Dies führte zur Standardisierung des V-Modells 97 im Jahre 1997 [DW99, S. 3, 4]. Die neuen Projekte und Entwicklungen in Methodik und Technologie wurden in diesem Modell dennoch unzureichend beschrieben und konnten somit nicht dem Stand der Technik entsprechen. Als Antwort auf diese Probleme wurde im Jahre 2004 das V-Modell XT vorgestellt. XT steht dabei für „extreme Tailoring“ („extremes Maßschneidern“) und drückt die angestrebte Flexibilität des Modells aus [Hö08, S. 3]. Das V-Modell XT ist die neueste Version des V-Modells und wird fortlaufend aktualisiert. Diese Version wird im Folgenden näher beschrieben.

Das V-Modell XT beschreibt die Erstellung eines IT-Systems als eine Folge von Aktivitäten, bei denen Produkte erstellt werden [DW99, S. 4]. Nach dem Erstellen eines Produktes

wird dieser gegen die gestellten Anforderungen geprüft (**Verifikation**) [Fr09, S. 107]. Zum anderen, ist auch zu prüfen, ob das erstellte Produkt für die vorgesehene Aufgabe geeignet ist, also ob z.B. das richtige System in Auftrag gegeben wurde. (**Validierung**) In der Software-Entwicklung erfolgt die **Verifikation** durch die ausgebildeten Prüfer, welche bei der Übergabe der Produkte, die in Form von formalen Dokumenten bestehen, mitwirken [Hö08, S. 36, 37] [Fr09, S. 107]. Die **Validierung** erfolgt dabei oft mithilfe des „Test-first“ Ansatzes. Dabei werden, angefangen bereits bei der Anforderungsdefinition bis hin zur Implementierungsphase, Tests und Abnahmekriterien für die Projektbausteine definiert [Va14, S. 2].

Insgesamt lässt sich das Modell wie in Abbildung 2 beschreiben. Dabei findet in dem Modell von Oben nach Unten eine schrittweise Verfeinerung des Gesamtsystems. Am Anfang findet eine Anforderungsdefinition, die dann bis zum Modulentwurf und deren Implementierung zerlegt wird. Nachdem man dann im Modell unten „angekommen ist“, wird das System wieder rekursiv aufgebaut und dabei die vorher geschriebenen Tests bis zur Abnahme durchgeführt [Go11, S. 93][LL07]. Das V-Modell XT ist insgesamt in 4 Submodelle gegliedert. Das im

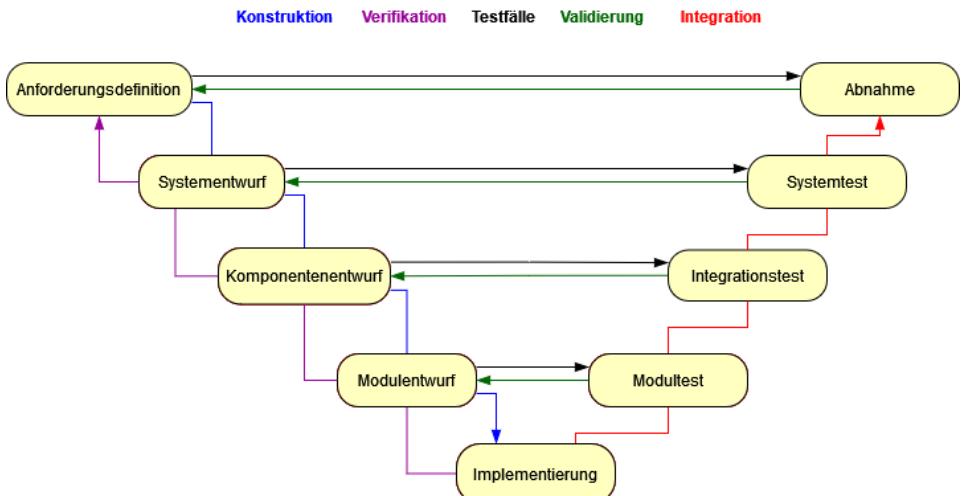


Abb. 2: Vereinfachte Darstellung des V-Modells XT

Bild 2 dargestelltes Modell beschreibt dabei das Submodell „Softwareentwicklung“. Zu den weiteren Tätigkeiten gehören: Qualitätssicherung, Konfigurationsmanagement und das Projektmanagement. Die Vier Modelle sind miteinander verbunden, und sorgen zusammen für einen kontrollierten, sicheren Projektablauf [Fr09, S. 6] [DW99, S. 4].

Der Kernunterschied des V-Modells XT zu ihrem Vorgänger, dem V-Modell XT liegt in der Anpassbarkeit und Flexibilität. Dabei wurde das Modell so entwickelt, dass es für verschiedenen Projekten angewendet werden kann. Zu den Maßnahmen gehören zum Beispiel, die anpassbare Größe der Dokumentation, damit „so viel ... wie nötig, aber so wenig wie möglich“ erzeugt wird [Fr09, S. 3]. Darüber hinaus lässt sich in dem Modell die Anzahl von Phasen oder Vorgehensbausteinen und die Zuordnung von Mitarbeiter-Rollen

jeweils spezifisch für das Projekt zuordnen. Diese Methode wird als „Tailoring“ bezeichnet. [Fr09, S. 6]

Insgesamt lässt sich sagen, dass das Modell gut für verschiedene Projekttypen und -Größen einsetzbar ist [Fr09, S. 3]. Wie bereits oben beschrieben, findet das V-Modell insbesondere in der Industrie und bei IT-Projekten der Bundesbehörden Einsatz. Darüber hinaus, lassen sich verschiedene Projektdurchführungsstrategien innerhalb des V-Modells auswählen, z.B. iterativ oder evolutionär, welche für unterschiedliche Arten von Projekten anwendbar sind [Hö08, S. 7]. Insbesondere kann das Modell in Projekten verwendet werden, die feste, verständliche und wenig veränderbare Anforderungen haben [Va14]. Die Größe dieser Projekte sollte auch nicht zu gering sein, weil ein großer Aufwand in der Planung eingesetzt werden muss.

Die Vorteile des Modells liegen in der Qualitätssicherung und strikten Kontrolle innerhalb des Modells. Die häufige Verwendung des „Test First“ Ansatzes und die Maßnahmen zur Verifikation und Validierung ermöglichen einen fehlerarmen Ablauf des Projekts und die Erstellung eines qualitativ hohen Endprodukts. Durch das neu eingeführte „Tailoring“ im V-Modell XT ist das Modell auch flexibler geworden und kann somit für verschiedene Projekte benutzt werden.

Die Nachteile des Modells leiten sich vor allem aus der Starrheit des Modells heraus. Da das Modell linear ist, können die einzelnen Projektphasen nach dem das Projekt angefangen ist, nicht einfach umstrukturiert werden. Darüber hinaus müssen bei dem Modell die Anforderungen strikt formuliert werden, da die Änderungen dieser schwer in der Mitte des Projekts umgesetzt werden. [AA15a] [BK08]

Wie bereits oben beschrieben, ist das V-Modell in 4 Submodelle gegliedert. Dadurch lassen sich die Rollen der Qualitätsmanager, der Softwarearchitekten und anderen Mitarbeiter voneinander trennen und jeder Mitarbeiter befindet sich in seinem Zuständigkeitsgebiet. Die Kommunikation wird dabei wie bei dem Wasserfallmodell mithilfe von Dokumenten geregelt. Darüber hinaus lassen sich unterschiedliche Arten von Austausch in Unternehmen regeln.

2.3 Rational Unified Process

Der *Rational Unified Process (RUP)* ist zum einen ein Vorgehensmodell, zum anderen auch die zugehörigen Entwicklungswerzeuge, die von IBM vertrieben werden. Entstanden ist RUP als Kombination der Prozesse der Firmen *Rational* und *Objectory*. Kombiniert werden dabei ein iterativer, inkrementeller Softwareprozess mit architekturzentrierter Vorgehensweise und die objektorientierte Darstellung, meist in Form von Use-Case-Diagrammen. [Hu15, S. 49, 50]

Im Gegensatz zu z. B. dem Wasserfallmodell überschneiden sich die einzelnen Phasen bei RUP und unterscheiden sich vor allem in ihrer Intensität voneinander in zeitlicher Hinsicht. Abbildung 3 veranschaulicht hierbei die zwei Dimensionen von RUP. Die vier Phasen sind dabei *Konzept*, *Entwurf*, *Implementierung* und *Produktübergabe*. Zu erkennen ist, dass die Implementierung bereits während der Konzeptphase beginnt und die Analyse und Design

auch während der Implementierung noch aktiv ist. Dadurch kann auf Anforderungsänderungen im Projektverlauf eingegangen werden. Zudem findet eine deutlich ausgeprägtere Kommunikation zwischen den verschiedenen Teams statt [Ve00, S. 6, 7].

Im Unterschied zu z. B. Scrum ist RUP ein kommerzielles Produkt. Es kann ohne Anpassungen direkt verwendet werden, bietet aber auch die Möglichkeit der Erweiterung und Anpassung [Kr99, S. 31].

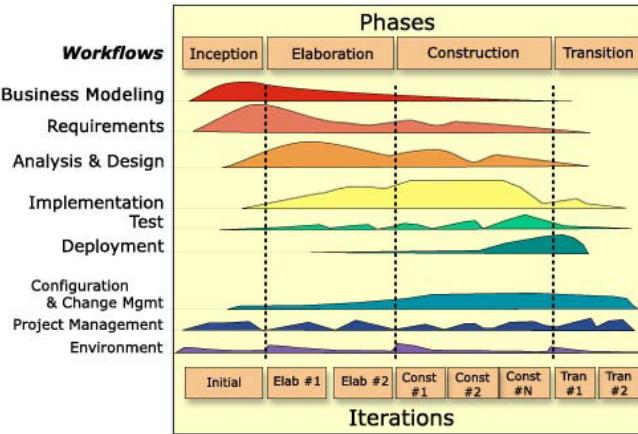


Abb. 3: Dimensionen von RUP. Übernommen von [Kr00]

2.4 Scrum

Scrum ist ein, vor allem von *Ken Schwaber* und *Jeff Sutherland* entwickeltes Vorgehensmodell, das der agilen Software-Entwicklung zugerechnet werden kann. Es wird seit den 1990er Jahren entwickelt und findet seitdem eine steigende Verbreitung [SS20, S. 1] [re18]. Einer der Kerngedanken von Scrum ist die selbstständige Arbeit der Mitarbeiterinnen und Mitarbeitern und das regelmäßige Reflektieren und Anpassen von Methoden und Vorgehensweisen um möglichst immer optimale Resultate zu erzielen. Um das zu erreichen teilt Scrum komplexe Probleme in kleinere Probleme auf, die für sich erarbeitet und bewertet werden können [SS20, S. 3].

Dabei gibt Scrum keine detaillierten Anweisungen vor, wie etwas zu tun ist, sondern lediglich einen Rahmen, der dafür essentiell ist. Innerhalb dieser Freiheit soll die Expertise der Mitarbeiterinnen und Mitarbeitern dafür sorgen, dass die nötigen Dinge getan werden. [SS20]

Die enge Zusammenarbeit, sowohl innerhalb des Teams als auch mit Kundinnen und Kunden ist hier stark ausgeprägt. Während sich z. B. das V-Modell stärker auf die Übergabe von Dokumenten verlässt, steht hier ein häufiger Kontakt im Fokus [Sh19]. Im wesentlichen gibt es nur drei Artefakte bei Scrum, *Product Backlog*, *Sprint Backlog* und *Increment* [SS20, S. 10–12].

2.5 Vergleich der vier Vorgehensmodelle

Bei einem Vergleich der vier Modelle lässt sich eine Einteilung in zwei Dimensionen vornehmen. Wie Abbildung 4 zeigt kann ein Modell einerseits als formell oder informell eingeordnet werden, andererseits kann in sequentielle und evolutionäre Modelle unterschieden werden. Sowohl das Wasserfallmodell als auch das darauf aufbauende V-Modell sind stark formell aufgebaut und stützen sich auf eine extensive Dokumentation, die bei der sequentiellen Abarbeitung des Entstehungsprozesses den Rahmen bildet. Die Zusammenarbeit findet hier primär über diese Artefakte statt, z. B. auch beim Übergang von Designphase zu Implementierungsphase.

Bei Scrum stehen deutlich weniger formelle Artefakte im Vordergrund, sondern die direkte Zusammenarbeit verschiedener Spezialistinnen und Spezialisten. Auch bei der Zusammenarbeit mit Kundinnen oder Kunden ist diese stärker ausgeprägt und es werden im Sinne einer evolutionären Entwicklung auch unvollständige Produktstände zum Testen an diese weitergegeben. RUP stellt einen Mittelweg dar, da zwar ebenfalls evolutionär vorgegangen wird, aber trotzdem noch ein stärkerer formeller Fokus als bei Scrum im Vordergrund steht.

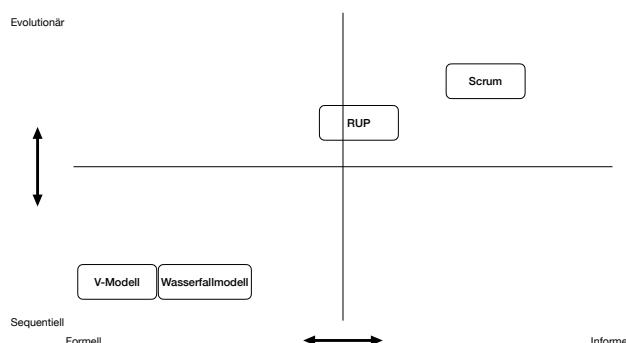


Abb. 4: Einordnung von Vorgehensmodellen. Angepasst nach [Sh19]

Ein weiterer wichtiger Unterschied zwischen den vier Vorgehensmodellen besteht darin, in welchen Bereichen das jeweilige Modell am geeignetsten ist. Das V-Modell und Wasserfallmodell eignen sich durch ihren Schwerpunkt auf eine präzise, gut dokumentierte Vorgehensweise z. B. für Anwendungsbereiche, in denen ebenfalls ein präzises Vorgehen unabdingbar ist. Shiklo [Sh19] nennt an dieser Stelle Softwareunternehmen im Bereich der Medizin oder dem Luftverkehr sowie staatliche Projekte, da hier vorhersehbare Projektpläne und Budgets mit einer strengen Kontrolle erforderlich sind.

Evolutionäre Vorgehensmodelle, wie etwa RUP können optimal für größere Projekte eingesetzt werden, bieten im Vergleich zu den sequentiellen Modellen aber bessere Möglichkeiten bei Anforderungsänderungen im Projektverlauf. Dabei ist RUP erst ab einer gewissen Projektgröße sinnvoll einzusetzen, da eine Vielzahl an Rollen erfüllt werden müssen. Scholz [Sc05, S. 150] nennt eine Teamgröße mit mehr als zehn Personen, mehr als 30 Rollen und über 100 verschiedene Artefakttypen.

Im Gegensatz dazu ist Scrum, als rein agiles Modell, vor allem für kleine Teams geeignet. Dies wird unterstützt durch eine Untersuchung, dass kleinere Teams bessere Leistung erbringen [HV70]. Das kann z. B. bei Start-Up-Unternehmen der Fall sein, bei denen die Kundeneinbeziehung auch stärker im Vordergrund steht. Aber auch größere Projekte können mithilfe von Scrum bewältigt werden, wenn diese in kleinere Teilprojekte aufgeteilt werden können. Hier hat sich der Begriff des *Scrum-of-Scrums* entwickelt, wobei die einzelnen Scrum-Teams einen Vertreter wählen, der wiederum Teil der übergeordneten Scrum-Teams ist [Da19, S. 227].

Allen genannten Modellen gemein ist, dass sie in Bezug auf die konkrete Zusammenarbeit relativ abstrakt bleiben. Welche Strategien und Werkzeuge können z. B. in der Phase der Implementierung helfen, Probleme zu vermeiden? Darauf geht das folgende Kapitel näher ein.

3 Zusammenarbeit während der Implementierung

Nach der Betrachtung der Zusammenarbeit in verschiedenen Vorgehensmodellen soll vertieft auf die gemeinsame Arbeit während der Implementierung eingegangen werden. Die Implementierung ist hier die Phase, in der auf Grundlage des Designs die nutzbare Software entsteht. Diese Phase kann streng zwischen der Designphase und der Testphase liegen, oder wie in agilen Vorgehensmodellen iterativ eingebettet sein. Unabhängig davon können im wesentlichen drei Bereiche der Zusammenarbeit genannt werden. Als erstes ist das die Übergabe von der Designphase zur Implementierung. Hauptlich ist dann die gemeinsame Arbeit an einer Code-Basis während der Implementierung betroffen und abschließend muss eine Übergabe an die Qualitätssicherung erfolgen. Die Ausgestaltung der gemeinsamen Arbeit am Quellcode wird näher betrachtet. Dabei geht es darum Strategien kennenzulernen und den Einsatz von Werkzeugen zu bewerten.

Während der Bearbeitung von Quellcode treten zwei Herausforderungen auf. Zum einen besteht der Wunsch auf ältere Versionen zurückfallen zu können, falls ein Experiment nicht funktioniert. Zum anderen soll es möglich sein, dass mehrere Entwicklerinnen und Entwickler parallel an der gleichen Datei arbeiten können. Naive Lösungen für diese Probleme sind leicht verständlich aber aufwändig. Vor einem Experiment wird eine Sicherheitskopie angelegt. Falls das Experiment scheitert, wird die aktuelle Datei verworfen und wieder durch die Sicherheitskopie ersetzt. Für die gleichzeitige Arbeit an Dateien kann eine Kennzeichnung der Änderungen im Code erfolgen, anschließend ein Austausch der Dateien. Eine der Entwicklerinnen oder einer der Entwickler hat nun die Aufgabe, die Dateien zu vergleichen und die Änderungen in einer resultierenden Datei zusammenzufassen. Dass dieses Vorgehen zwar funktioniert, aber nicht sehr effizient ist, ist offensichtlich. Um diese Arbeitsschritte zu automatisieren haben sich Systeme für die Quellcode-Verwaltung entwickelt, die beide Herausforderungen erleichtern. Im Folgenden wird ein kurzer Überblick über diese Systeme gegeben bevor Strategien für die Zusammenarbeit damit vorgestellt werden.

3.1 Quellcode-Verwaltungs-Systeme

Die Entwicklung von Software für die Quellcode-Verwaltung findet bereits seit den Siebziger Jahren statt und hatte mit *IEUPDAT* von IBM [Co22; IB22] ein erstes rudimentäres Werkzeug. Weitere Systeme waren das *Source Code Control System (SCCS)* [Mc21] oder das *Revision Control System (RCS)* [GN22] und mit *Concurrent Versions System (CVS)* auch ein System, bei dem mehrere Entwicklerinnen und Entwickler gleichzeitig an einer Datei arbeiten können [Mc21]. Aktuellere Systeme sind z. B. *Git* [CL22], *Subversion* [Th22], *Mercurial* [Me22] oder *Team Foundation Version Control (TFVC)* [Mi22a]. Diese Systeme können nach ihrer grundlegenden Funktionsweise unterschieden werden:

- Lokale Versionskontrolle: Lokale Versionierung, oft von einer einzelnen Datei. Z. B. SCCS, RCS
- Zentrale Versionskontrolle: Hier ist auch die Kommunikation über das Netzwerk möglich. Wichtig ist, dass der aktuelle Stand des Quellcodes immer auf dem zentralen Server ist. Bei der Bestätigung einer Änderung wird diese auf das zentrale Repository übertragen. Z. B. CVS, Subversion, TFVC
- Verteilte Versionskontrolle: Auch hier ist die Netzwerkkommunikation möglich. Oft existiert auch hier ein zentrales Repository, dieses dient aber nur dem gemeinsamen Austausch. Jede Entwicklerin und jeder Entwickler hat ein voll funktionsfähiges lokales Repository. Z. B. Git, Mercurial

Die Plattform *Openhub* [Sy22] veröffentlicht die Verteilung auf Quellcode-Verwaltungs-Systeme von bei ihr gehosteten Projekten. Im November 2022 hatte *Git* einen Anteil von 73 Prozent, gefolgt von *Subversion* mit 22 Prozent sowie *Mercurial* und *CVS* mit jeweils einem Prozent. Andere Systeme sind hier nicht relevant. Anzumerken ist, dass hier ausschließlich Open-Source-Projekte in die Statistik fallen, weshalb davon auszugehen ist, dass auch weitere Systeme im Unternehmenskontext relevant sind. Deutlich sichtbar ist aber die Bedeutung von *Git*, weshalb sich diese Arbeit in der Folge auf dieses System konzentriert.

3.2 Quellcode-Hosting-Lösungen

Es existieren verschiedene Plattformen, die die gemeinsame Bearbeitung von Quellcode und das Hosting von Projekten ermöglichen. Plattformen, die auf *Git* aufbauen sind z. B. *Github*, *BitBucket* (das ursprünglich rein auf *Mercurial* fokussiert war) und *Gitlab*. Im Gegensatz zur reinen Versionsverwaltung, auf der diese Plattformen aufbauen steht die Zusammenarbeit und das Präsentieren des Projekts im Vordergrund.

3.3 Strategien zur Quellcode-Verwaltung

Die genannten Quellcode-Verwaltungs-Systeme helfen bei der gemeinsamen Bearbeitung des Quellcodes, die als stetes Aufspalten und Zusammenführen von Quellcode gesehen werden kann. Im Folgenden werden verschiedene Strategien und Workflows vorgestellt, um diesen Vorgang zu gestalten. Diese Strategien lassen sich grundlegend auch mit anderen Quellcode-Verwaltungs-Systemen anwenden, für die Beispiele wird jedoch Git verwendet. Es wird herausgestellt, welche Vor- und Nachteile die Systeme haben und welche Herausforderungen sie haben. Einige Begriffe sollten eingeführt werden, um Unschärfen zu vermeiden. Das sind die folgenden:

- **Repository:** Der Ort, in den Regel ein Verzeichnis, an dem die Quellcode-Dateien sowie Metadaten und Informationen für die Quellcode-Verwaltung gespeichert sind. Oft gibt es ein *zentrales Repository*, das über das Netzwerk erreichbar ist und zum Austausch im Team dient.
- **Klonen:** Beim Klonen eines, meist zentralen, Repositorys wird eine lokale Kopie erstellt, auf der gearbeitet werden kann. Die Bearbeitungen können später wieder mit dem zentralen Repository vereinigt werden.
- **Commit:** Die *Bestätigung*, dass die Arbeit an Quellcode-Dateien gesichert werden soll. Mit einem Commit wird der aktuelle Stand in die Quellcode-Verwaltung aufgenommen und kann später wieder abgerufen werden oder an ein *zentrales Repository* geschickt werden.
- **Branch:** Eine Folge von *Commits*. Für einen Branch sind mehrere Betrachtungsweisen möglich. Als Beispiel ist ein zentrales Repository gegeben, welches genau einen Branch enthält. Beim Klonen des Repositorys existiert dieser Branch dann zwei mal. Bei einem Commit auf dem lokalen Branch ändert sich der Branch auf dem zentralen Repository erstmal nicht. Dadurch ergeben sich zwei unterschiedliche Branches, die jedoch den gleichen Namen besitzen können, wobei sich der Speicherort unterscheidet. Das wäre eine Betrachtung von Branches. Im Fall von Git wird auch das dedizierte Erstellen von Branches angeboten, die neben dem *Hauptbranch* existieren und beim Klonen mit berücksichtigt werden. Diese können sowohl im zentralen als auch im lokalen Repository angelegt werden. Das wäre die zweite Betrachtung von Branches.
- **Mergen:** Wenn mehrere Branches existieren besteht in der Regel die Anforderung, diese zu einem späteren Zeitpunkt wieder zu vereinigen. Dieser Vorgang wird als Mergen bezeichnet. Es bedeutet, dass die Commits aus einem Branch in einen anderen Branch übernommen werden. Dabei kann es zu Konflikten kommen, wenn eine Datei in beiden Branches bearbeitet wurde.
- **Integration:** Das Mergen von Änderungen in den *Hauptbranch* des zentralen Repositorys.

Die Betrachtung der folgenden Strategien beruht auf der Annahme, dass jede Version der Software auf der vorangegangenen basiert. Das bedeutet nicht, dass nicht mehrere Versionen gleichzeitig unterstützt und sogar noch mit Hotfixes versehen werden können. Aber wenn an einer Software kundenspezifische Anpassungen vorgenommen wurden, dann dürfen die jeweiligen Anpassungen nur in den Versionen des jeweiligen Kunden oder der Kundin auftauchen. In diesem Fall ist eine komplexere Strategie und mehrere parallele Entwicklungszweige meist notwendig.

Wie Fowler [Fo20] schreibt, ist nicht das Branchen ein Problem, sondern das spätere Mergen. Deshalb soll zuerst auf die Probleme hierbei eingegangen werden, bevor Strategien zum Branchen vorgestellt werden.

3.3.1 Merging

Im optimalen Fall ist das Design einer Software so modular und leicht erweiterbar, dass eine Quellcode-Datei nie von mehreren gleichzeitig bearbeitet wird. Dann würde es beim Mergen keine Probleme geben. In der Realität ist es aber oft der Fall, bzw. teilweise notwendig, dass eine Datei parallel bearbeitet wird. In diesem Fall ist eine gute Strategie für das Mergen notwendig, um die auftretenden Konflikte zu minimieren bzw. eine Auflösung zu erleichtern. Im Falle, dass es parallele Bearbeitungen einer Datei gibt, lassen sich im wesentlichen zwei Konflikttypen unterscheiden, *textuelle* und *semantische* Konflikte [Fo20]. Ein textueller Konflikt tritt dabei auf, wenn z. B. der Name einer Klasse von zwei Entwicklerinnen oder Entwicklern jeweils unterschiedlich geändert wird. Ein semantischer Konflikt tritt auf, wenn Entwicklerin A den Name der Klasse ändert, Entwickler B parallel diese Klasse mit ihrem alten Namen verwendet. Während ein textueller Konflikt vom Quellcode-Verwaltungssystem leicht erkannt wird und von den Entwicklerinnen und Entwicklern gelöst werden kann, wird ein semantischer Konflikt nicht erkannt und fällt erst zur Kompilierzeit oder im schlimmsten Fall zur Laufzeit auf. Neben dem technischen Mergen des Quellcodes sind also auch hier eine Strategie und Werkzeuge nötig, um semantische Konflikte zu erkennen, bevor eine Integration stattfindet. Eine Möglichkeit sind z. B. *Unit Tests*, die nach dem Mergen der Änderungen auf dem lokalen Repository durchgeführt werden bevor zum zentralen Repository integriert wird. Dadurch wird dafür gesorgt, dass alle Branches auf dem zentralen Repository immer in einem stabilen Zustand sind.

Weitere Herausforderungen ergeben sich aus der Häufigkeit des Mergens, bzw. des Integrierens. Abbildung 5 zeigt zwei unterschiedliche Möglichkeiten der Visualisierung von Branches. Dabei wird veranschaulicht, dass mit fortschreitender Zeit Branches immer weiter divergieren, anstatt parallel nebeneinander zu laufen. Durch häufiges Integrieren, z. B. mit Hilfe der Continuous Integration, lässt sich diese Divergenz begrenzen, wodurch das Mergen weniger Konfliktpotential besitzt. Nicht in allen Teams ist das allerdings möglich, da hierfür eine gute Testumgebung notwendig ist [Fo20], insbesondere die Open-Source-Community unterscheidet sich hier deutlich vom Unternehmenskontext.

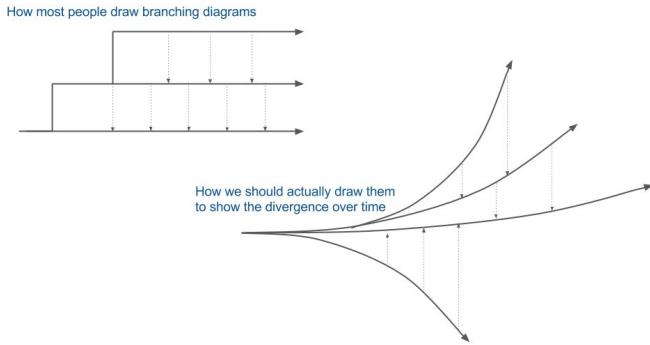


Abb. 5: Möglichkeiten für die Visualisierung von Branching-Diagrammen. Übernommen von [Le17]

3.3.2 Ein-Branch-Strategie

Der einfachste Workflow besteht darin, dass nur ein Branch genutzt wird. In Git wird dieser inzwischen standardmäßig *main* genannt. Wenn Entwicklerinnen oder Entwickler an einem Feature arbeiten möchten, klonen sie das zentrale Repository und arbeiten auf ihrem lokalen Repository auf dem *main*-Branch. Hier ist also die erste, oben genannte Betrachtungsweise, eines Branches relevant. Wenn das Feature fertig ist, können die Änderungen zurück zum zentralen Repository gepusht werden. Falls dort bereits andere Änderungen committed wurden, muss zuvor ein Merge durchgeführt werden.

Dieser Workflow ist lediglich für wenig komplexe Projekte geeignet, wie z. B. die Erstellung dieser Arbeit, bei der wenige Entwickler beteiligt sind. Ein Nachteil ist, dass ein Commit, der an das zentrale Repository gepusht wird, potentiell für einen instabilen Zustand sorgt. Insbesondere, wenn an einem Feature gearbeitet wird, das länger dauert und aus mehreren Commits besteht, kann es notwendig sein, dass ein Austausch über die bisherigen Commits stattfindet. Der einzige Weg, der hier möglich ist, ist der *main*-Branch auf dem zentralen Repository. Wenn das unfertige Feature nicht sauber vom Bestandscode getrennt ist, kann es zu Problemen kommen. Der Vorteil ist, dass der Workflow sehr leicht zu verstehen ist. Bei Git wird diese Strategie zum Teil auch als *Basic Workflow* bezeichnet [An21]. Die genannten Probleme könnten vermieden werden, wenn dieser Workflow in Kombination mit einer Forking-Strategie (siehe auch Abschnitt 3.3.4) genutzt wird.

Alternativ ist ein Vorgehen nach dem Continuous Integration möglich. Zentraler Punkt hierfür ist das Integrieren sobald ein stabiler Zustand erreicht ist. Dabei muss auch bei länger dauernden Features gewährleistet werden, dass diese abgekoppelt sind, indem z. B. das Interface als letztes implementiert wird [Fo20] und tatsächlich stabile Zustände erreicht werden. Es findet hier also eine Verlagerung der Arbeit statt. Anstelle aufwändige und fehleranfällige große Merge-Vorgänge durchzuführen, wird der Fokus darauf gelegt, mehr Aufwand in die Erreichung von stabilen Zuständen zu stecken und dafür kleinere, einfache Merge-Vorgänge zu haben.

3.3.3 Multi-Branch-Strategie

Eine weitere Möglichkeit, um die Nachteile der Ein-Branch-Strategie zu vermeiden, ist die Nutzung von weiteren Branches neben dem Hauptbranch. Durch die Nutzung von eigenen Branches, im Sinne der zweiten, oben genannten Betrachtungsweise, für die Entwicklung von Features, wird der *main*-Branch in einem stabilen Zustand gehalten und trotzdem können auf dem Feature-Branch Commits vorgenommen und auch ausgetauscht werden. Die Kernidee besteht darin, dass einzelne Features getrennt voneinander entwickelt werden, und zwar bis sie komplett fertig sind. Da die Feature-Branches aber auch auf dem zentralen Repository gespeichert werden können, ist trotzdem ein Austausch der Änderungen mit anderen Entwicklerinnen und Entwicklern möglich ohne den *main*-Branch in einen instabilen Zustand zu bringen. Nicht zu unterschätzen ist trotzdem das Problem, die einzelnen Branches wieder in den *main*-Branch zu mergen. Denn auch wenn die einzelnen Features inhaltlich unabhängig sind, kann es vorkommen, dass an den gleichen Stellen Änderungen im Bestandscode notwendig sind.

Ein konkretes Beispiel für diese auch *Feature-Branching* genannte Strategie ist *git-flow* [Dr10]. Folgende Branches existieren in diesem Workflow:

- Main (oder Master): Dieser spiegelt zu jeder Zeit einen Stand wider, der ausgeliefert werden kann. Zusammen mit Develop bilden sie die Haupt-Banches.
- Develop: Dieser spiegelt den aktuellen Stand der Entwicklung wider. Von diesem Branch gehen Feature- und Release-Branches ab und hierhin wird integriert.
- Feature Branches: Werden von Develop abgespalten und dorthin integriert. Sind für die dedizierte Entwicklung von Features vorgesehen.
- Release Branches: Werden von Develop abgespalten, um unabhängig von weiteren Integrationen nach Develop letzte Fehler zu korrigieren und einen stabilen Zustand herzustellen, der nach Main integriert werden kann. Wird auch wieder zurück nach Develop integriert.
- Hotfix Branches: Werden von Main abgespalten, um einen schlimmen Fehler einer veröffentlichten Version zu beheben. Wird wieder nach Main und Develop integriert.

3.3.4 Weitere Strategien

Neben den ausführlich beschriebenen Strategien, die zu den am häufigsten beschriebenen gehören, gibt es weitere, die je nach Anwendungsumfeld eine Alternative darstellen können. Dabei entstehen beständig neue Vorschläge oder Varianten, um sich an die neuen Entwicklungsumgebungen und Anforderungen anzupassen. Dazu gehören z. B. der *Forking Workflow* [An21], *OneFlow-Workflow* [Ru17] oder *Github Flow* [Ch11].

Der *Forking-Workflow* hat sich insbesondere in der Open-Source-Community und mit der Verbreitung von Plattformen wie GitHub oder GitLab durchgesetzt, wo zum Teil viele

Entwicklerinnen und Entwickler mit unterschiedlichen Arbeitsweisen zusammenarbeiten. Anstelle eines einzigen zentralen Repositorys, zu dem alle ihre Änderungen pushen besitzt jeder ein eigenes zentrales Repository (ein sogenannter *Fork*) auf dem gearbeitet wird. Dadurch können lokale Änderungen veröffentlicht und geteilt werden, um beispielsweise Feedback zu erhalten, ohne das originale Repository in einen instabilen Zustand zu bringen. Der Vorteil besteht darin, dass die Besitzerin oder der Besitzer des originalen Repositorys keinen weitreichenden Zugriff auf das Repository an viele Entwicklerinnen und Entwickler gewähren muss. Über Pull-Requests kann jedoch eine Anfrage an das originale Repository gestellt werden, die Änderungen auch dort zu integrieren. Dieser Workflow kann je zentralem Repository mit anderen Workflows kombiniert werden, sodass an einem Fork auch mehrere arbeiten können.

OneFlow ist ebenso wie *Github Flow* eine Reaktion auf den *GitFlow-Workflow*, der für moderne Web-Entwicklung nicht mehr optimal war und als zu komplex und fehleranfällig betrachtet wurde [Ru15][Ch11]. Die Kernidee ist hierbei, möglichst wenige langlebige Branches zu haben, um die Versionsgeschichte möglichst klar zu lassen. Ein weiterer Vorteil weniger Branches ist, dass es weniger Verwirrung über die richtige Verwendung der Branches und weniger Probleme beim Mergen gibt. Dies wird erreicht, indem sehr häufig integriert wird und der main-Branch immer in einem stabilen Zustand gehalten wird, z. B. mithilfe von selbsttestendem Code. Insbesondere bei der Webentwicklung und Continuous Delivery/Deployment vereinfacht das die Arbeit unter der Voraussetzung.

3.4 Grenzen von Git

Bei der Wahl einer Strategie sollte auch berücksichtigt werden, welche Artefakte während der Implementierung anfallen. Insbesondere Binärdateien können dabei problematisch sein. Binärdateien können zum Beispiel 3D-Modell-Daten sein [Bl22]. Auch hier fallen oft lediglich kleine Anpassungen an, sodass es unnötig wäre, während des Zusammenführens die komplette Datei zu kopieren. Dies ist aber notwendig, da Git bei Binärdateien nicht nur die Änderungen speichern kann. Dadurch erhöht sich der Speicherbedarf des Repositorys stark und auch die Performance kann darunter leiden, wenn jedes Mal viele Daten übertragen werden müssen. Hier muss eine Einschätzung getroffen werden, wie häufig Binärdateien geändert werden und wie groß diese sind. Wenn nur wenige, kleine Binärdateien im Repository vorhanden sind, die zudem selten geändert werden, zum Beispiel finale Dokumentations-PDFs, kann Git ohne Bedenken genutzt werden. Im Falle vieler, potentiell großer Binärdateien, wie zum Beispiel bei der Entwicklung von 3D-Computerspielen stellt dies einen Einsatz von Git in Frage. Zur Lösung dieses Problems gibt es Ansätze als Erweiterungen von Git [Ke16]. Der Kerngedanke ist dabei, dass nicht die Binärdateien selbst im Repository gespeichert sind, sondern Zeiger auf diese Dateien [Ge19]. Dadurch kann der Speicherbedarf und die Performance von Git optimiert werden. Die Versionierung und parallele Bearbeitung der Binärdateien muss dann aber durch das entsprechende Bearbeitungsprogramm sichergestellt werden können.

3.5 Zwischenfazit

Die vorgestellten Strategien bieten eine Grundlage für die Entscheidungsfindung bei der Wahl von Richtlinien und Werkzeugen während der Zusammenarbeit bei der Implementierung. Nicht vergessen werden sollte, dass diese Strategien nicht für sich stehen und auch keinen Anspruch auf Gültigkeit erheben. Sie müssen immer im Kontext der gesamten Entwicklungsumgebung stehen und natürlich gegebenenfalls an die konkrete Situation im Projektteam angepasst werden. In diesem Zusammenhang ist auch eine Abstimmung mit der Strategie des Qualitätsmanagements notwendig, auf das im Folgenden näher eingegangen wird. Eine Untersuchung zur Softwarequalität in Abhängigkeit der Branching-Strategie kam dabei zu dem Ergebnis, dass zu viele Branches einen negativen Einfluss haben und die Branching-Strategie mit der Software-Architektur und der Team-Struktur abgestimmt sein sollte, um negative Konsequenzen zu vermeiden [SBZ12].

4 Qualitätsmanagement und Qualitätssicherung

Die Software ist ein großer Teil unseres Lebens geworden und ist Bestandteil vieler Produkte, die wir nutzen: von den Handys bis zu den Autos. Wir erwarten, dass die Programme korrekt und fehlerlos funktionieren, ohne dabei nachzudenken, welcher Verwaltung- und Sicherungsaufwand dahintersteckt. Wenn diese Anforderungen erfüllt werden, spricht man von „guter Qualität“. [Sc12, S. 1, 2] Was bedeutet überhaupt „gute Qualität“ und wie lässt sich diese messen? Welche Maßnahmen sind dabei notwendig, um die hohe Qualität zu leisten? Welche Ansätze zur Qualitätssicherung sind für welche Projektgrößen und Vorgehensmodellen anwendbar und besonders vorteilhaft? In diesem Kapitel wird versucht, diese Fragen zu beantworten.

4.1 Definition und Eingrenzung

Es gibt unterschiedliche Wege, die Softwarequalität zu bewerten. In der Softwareentwicklung wird häufig der Standard ISO/IEC 9126 zur Bewertung der Softwarequalität verwendet. Diese Norm definiert den Begriff Software-Qualität als:

„Gesamtheit der Merkmale und Merkmalswerte eines Software-Produkts, die sich auf dessen Eignung beziehen, festgelegte Erfordernisse zu erfüllen.“ [Ho08, S. 6]

Konkret bedeutet das, dass die Software-Qualität sich aus mehreren verschiedenen Kriterien zusammensetzt. Der Standard definiert dabei 6 Qualitätsmerkmale: Funktionalität, Zuverlässigkeit, Benutzbarkeit, Effizienz, Wartbarkeit und Portierbarkeit. Dabei beeinflussen sich die Merkmale gegenseitig: positiv und negativ. Zum Beispiel, eine Verbesserung der Wartbarkeit

ergibt eine Verbesserung der Funktionalität, aber eine Verschlechterung der Effizienz. Somit lässt sich sagen, dass eine Verbesserung aller Qualitätsmerkmale gleichzeitig nicht möglich und ein Kompromiss zu suchen ist [Ho08, S. 11][Go11, S. 132]. Um die Software-Qualität optimal zu verbessern, werden Maßnahmen zur Qualitätssicherung gebraucht. Diese lassen sich in die Produktqualität- und Prozessqualitätsmaßnahmen unterteilen.

Bei der Produktqualität steht das Softwareprodukt im Vordergrund. Die Maßnahmen dieser Gruppe werden in konstruktive Maßnahmen (Fehlervermeidung während der Entwicklung) und analytische Maßnahmen (Fehlerfindung nach der Entwicklung) unterteilt.

Die Prozessqualität umfasst die in Kapitel 2 beschriebenen Vorgehensmodellen und andere Verwaltungstechnologien. Für diese Arbeit wird der Bereich der Prozessqualität. Es werden lediglich die Auswirkungen und Vorteile auf die Projekte in unterschiedlichen Vorgehensmodellen beim Einsatz der verschiedenen Produktqualitätsmaßnahmen beschrieben.

4.2 Konstruktive Qualitätssicherung

Zu dem Gebiet der konstruktiven Qualitätssicherung fallen insbesondere Maßnahmen, die eine frühzeitige Erkennung und Vermeidung der Fehler während des Entwicklungsprozesses ermöglichen. In diesem Kapitel werden 3 wichtigsten davon: **Softwarerichtlinien**, **Vertragsbasierte** Programmierung und **Dokumentation** vorgestellt.

Softwarerichtlinien bezeichnen alle Mittel, die den Programmierprozess über die syntaktischen und semantischen Regeln der jeweiligen Programmiersprache hinweg regelt [Ho08, S. 65]. Die wichtigsten Ziele dieser Maßnahmen sind *Vereinheitlichung* und *Fehlerreduktion*. Um diese Ziele zu erreichen, werden *Notations-* und *Sprachkonventionen* eingesetzt.

Die meisten Software-Ingenieure entwickeln im Laufe der Jahre ihren eigenen Programmier-Stil, worauf sich die Person gewohnt. In Teams kann diese Tatsache aber dazu führen, dass mehrere Mitarbeiter den Code von anderen nicht richtig verstehen oder mehr Zeit dafür brauchen. Um dieses Problem zu lösen, werden **Notationskonventionen** verwendet. Diese umfassen die Namensgebung der verwendeten Strukturen und Variablen, die Verwendung von Leerzeilen und Charakteristik und die Dokumentation des geschriebenen Codes. Oft gibt es unternehmensspezifische Konventionen, dennoch haben sich folgende Notationsvorschriften in der Software-entwicklung durchgesetzt:

- **Pascal Case:** die Variablennamen beginnen mit dem Großbuchstaben und jedes weitere Wort wird ohne Trennzeichen mit großen Buchstaben eingefügt (z.B. `ForegroundColor`).
- **Camel Case:** gleich wie Pascal Case, aber das erste Wort wird mit Kleinbuchstaben angefangen (`foregroundColor`).
- **Uppercase/Lowercase:** der gesamte Name wird entweder in Groß- oder Kleinbuchstaben geschrieben (`FOREGROUNDCOLOR`).

Oft werden verschiedene Notationsstile miteinander kombiniert, damit verschiedene Arten von Bezeichner dargestellt werden. Dies sorgt für ein besser lesbares und verständliches Code.

Um ein weiteres Ziel der Softwarerichtlinien, die Fehlerreduktion, zu ermöglichen, werden die **Sprachkonventionen** verwendet. Im Gegensatz zu den Notationskonventionen, werden hier die Entwickler darin eingeschränkt, bestimmte Sprach-Konstrukte unter bestimmten Voraussetzungen zu verwenden. Dazu gehören zum Beispiel, mit welchen VariablenTypen auf die einzelnen Bits zugegriffen werden darf (unsigned int in MISRA C Standard) [Ho08, S. 76] [Mi22b].

Einen weiteren Bestandteil der konstruktiven Qualitätssicherung bildet die **Vertragsbasierte** Programmierung. Das Konzept wurde von Bertrand Meyer unter dem Namen Design by contract eingeführt [Me92]. Kurz beschrieben, basiert die vertragsbasierte Programmierung auf den folgenden Prinzipien:

- **Vor- und Nachbedingungen:** die Objekte müssen bestimmte Voraussetzungen erfüllen, um Routinen betreten und verlassen zu können. Dafür werden bestimmte „require“ Blöcke in das Programm eingefügt.
- **Invarianten:** im Programm werden globale Beschränkungen der Wertebereiche bestimmter Variablen und Strukturen gesetzt.
- **Zusicherungen:** bezeichnen die Überprüfungen, die im Gegensatz zu Invarianten nur an der Stelle des Auftretens ausgewertet werden.

All diese Prinzipien werden auch heute bei der Anforderungsspezifikation und für die Definition von Tests eingesetzt [Ho08, S. 96] [Me92]

Verwaltung der **Projektdokumentation** ist auch eine wichtige Maßnahme zur Herstellung der Qualitätssicherheit. Dabei lässt sich diese in die externen und internen Dokumente aufteilen. Die externe Dokumentation dient dem Austausch mit dem Kunden. Diese hat oft, insbesondere in großen Projekten klare Vorgaben und Muster wie sie zu erstellen ist (z.B. für Pflichtenheft) [Ho08, S. 141]. Bei der internen Dokumentation handelt es sich um andere Dokumente, die nur unternehmensintern zur Verfügung stehen. Diese Dokumente umfassen zum einen die oben beschriebene Software-Richtlinien und auch die Programmdokumentation und andere Spezifikationen.

Einer der wichtigsten Dokumente in vielen Vorgehensmodellen ist die Spezifikation. In dem Dokument werden die Anforderungen an ein Software-System genauer beschrieben. Diese Spezifikation kommt daher oft als externes Dokument und wird bei vielen Projekten formal gehalten. Die Anwender tauschen dann während der Entwicklung die sogenannten Implementierungsdokumente aus, wo anhand von Kommentaren im Code die implementierten Funktionen beschrieben werden. Die Formulierung dieser Dokumente kann großen

Aufwand, vor allem in dokumenten-basierten Vorgehensmodellen beanspruchen, weswegen auch viele dieser Modelle, wie z.B. das V-Modell kritisiert wird. [AA15c]

4.3 Analytische Qualitätssicherung

Bei der analytischen Qualitätssicherung wird, im Gegensatz zur konstruktiven, geht es nicht um Fehlervermeidung, sondern um Fehlerfindung und Beseitigung nach dem Entwicklungsprozess [Ho08, S. 20] [Go11, S. 132]. Dies wird mithilfe verschiedener Test- und Verifikationsverfahren nach und während der Entwicklung ermöglicht. Diese werden im Folgenden vorgestellt.

4.3.1 Software-Tests

Bereits im Jahr 1979 war in der Software-Entwicklung bekannt, dass ungefähr die Hälfte der Zeit- und Geldkosten in das Testen investiert werden muss [My12]. Diese Tatsache hat sich bis heute nicht wesentlich verändert, wenn überhaupt, nimmt die Implementierung und Durchführung der Tests in manchen Projekten einen wesentlich höheren Anteil an Aufwand. Diese Tests lassen sich bezüglich verschiedener Merkmale in verschiedene Klassen einteilen [Ho08, S. 158]:

- **Prüfebene:** In welcher Projektphase wird der Test durchgeführt?
- **Prüfkriterium:** Welche inhaltlichen Aspekte werden getestet?
- **Prüfmethodik:** Wie werden die Tests konstruiert?

In dieser Arbeit werden verschiedene Testarten anhand ihres Prüfkriteriums vorgestellt. Diese Tests werden anhand ihres Inhaltes unterschieden und können dabei in 3 Kategorien unterteilt werden: Funktionale, Operationale und Temporale Tests. Die wichtigsten Arten von **funktionalen** Tests sind [Ho08, S. 170] [Me18, S. 242]:

- **Funktionstests:** sind die am häufigsten verwendete Tests, bei diesen Tests wird überprüft, ob ein richtiges Ergebnis geliefert wurde, also ob die Applikation korrekt läuft.
- **Crashtests:** es wird versucht, ein System zum Abstürzen zu kriegen. Es findet eine gezielte Suche nach Schwachstellen statt, durch die Bekämpfung dieser, können sicherheitskritische Systeme geschützt werden.
- **Zufallstests:** es wird nicht mit vorgegebenen, gezielt erzeugten Eingabedaten getestet werden, sondern mit zufälligen. Da diese Methode aber schwer systematisch anzuwenden ist, kann sie nur als Ergänzung verwendet werden.

Die **operationalen** Tests beinhalten im Großen und Ganzen die Installation, Sicherheit und Bedienbarkeit von Systemen. Diese Tests sind insbesondere in der Web-Branche und anderen modernen Applikation wichtig, da die Bedienbarkeit und User-Experience häufig im Vordergrund stehen [Ho08, S. 170–174] [Me18, S. 242]. In den **temporalen** Tests werden vor allem die Effizienz und Schnelligkeit von Software getestet. Zum Beispiel wird es bei *Last- und Stresstesten* geprüft, wie sich ein System an den oder über den definierten Grenzen verhält. Die Geschwindigkeit von Software wird in den *Komplexitäts- und Laufzeitests* ermittelt, wodurch auch entschieden werden kann, ob die implementierten Algorithmen den Anforderungen entsprechen.

4.3.2 Statische Analyse und Verifikation

Durch die **statische Code-Analyse** umfasst alle Maßnahmen, bei denen der Quellcode ohne Programmausführung untersucht wird. Die untersuchenden Bestandteile des Codes werden mithilfe von Software-Metriken beschrieben. Durch diese Analyse wird die Zuverlässigkeit und die Funktionalität des Codes geprüft. Zu den verbreiteten Software-Metriken gehören die LOC (Lines of Code) und NCSS (Non-Commented Source Statements). Bei diesen Software-Metriken wird die Anzahl der geschriebenen Programmzeilen, bzw. ausführbaren Zeilen des Codes gezählt und dadurch wird eine grobe Aussage über die Programmkomplexität gemacht [Ho08, S. 249, 250]. Jedoch sagt dieser Ansatz in der Realität nicht viel über das Programm aus, weil unterschiedliche Programmiersprachen unterschiedliche Anzahlen von Befehlen und dadurch Zeilen haben können und auch den Aspekt der optimalen Nutzung der Werkzeuge betrachtet werden muss. Deswegen werden heutzutage viele andere Metriken benutzt, zum Beispiel wird es untersucht, wie oft eine Funktion überschrieben wird (in objektorientierten Sprachen) oder wie oft die Funktion im Programm aufgerufen/wieder verwendet wird. Bei diesem Ansatz wird schnell über die Rahmen der statischen Analyse gegangen und es wird in einer **Software-Verifikation** automatisch geprüft. Dabei wird der Code ausgeführt und mithilfe von verschiedenen mathematischen Analysen untersucht [Ho08, S. 334, 335].

4.4 Produktqualitätsmaßnahmen in Vorgehensmodellen

Wie bereits oben beschrieben, gibt es viele verschiedene Ansätze und Maßnahmen zur Qualitätssicherung in der Softwareentwicklung. Es gibt dennoch keine Vorgaben, welche dieser Methoden für welche Arten von Projekten angewendet werden können. Es lässt sich aber sowohl aus den Projektgrößen als auch der verwendeten Vorgehensmodellen die passenden Qualitätsmaßnahmen herausleiten. Dabei gibt es allgemeine Maßnahmen, durch die alle Projekte profitieren können und die auch oft eingesetzt werden. Dazu gehören, zum Beispiel, **Softwarerichtlinien** (4.2), weil man mithilfe der Sprach- und Notationskonventionen viele Fehler vermeiden kann, ohne dabei einen großen Managementaufwand einzusetzen.

Die anderen Methoden werden im Folgenden tabellarisch auf verschiedene Vorgehensmodelle und Projektgrößen angewendet (siehe Abbildung 6): Wie man sieht, werden viele

Qualitätsmaßnahme	Passendes Vorgehensmodell	Projektgröße
Vertragsbasierte Programmierung	V-Model, Wasserfallmodell	Mittel bis hoch
Projektdokumentation	Alle, aber insbesondere V-Modell und Wasserfallmodell	Je größer das Projekt, desto mehr Dokumentation
Funktionale Software-Tests	Allgemein einsetzbar, aber sehr wichtig in Scrum und RUP	Für alle Projektgrößen
Operationale Tests	Besonders wichtig für Scrum	Für alle Projektgrößen
Temporale Tests	V-Modell und RUP	Große Projekte
Statische Code-Analyse und -Verifikation	V-Modell	Große Projekte

Abb. 6: Anwendung verschiedener Qualitätsmaßnahmen

Maßnahmen im V-Modell und nicht in Scrum verwendet, weil im V-Modell der Fokus auf die Qualitätssicherung und Dokumentation gelegt wird. Im Gegensatz dazu wird bei Scrum schneller auf die Anforderungsänderungen reagiert und es wird schneller ein Ergebnis an den Kunden geliefert.

5 Zusammenfassung und Ausblick

Im Rahmen dieser Arbeit wurde die Bedeutung einer gelungenen Zusammenarbeit bei der Erarbeitung großer Software-Projekte hervorgehoben, wobei bereits gerade auch sehr kleine Teams große Probleme bei der Zusammenarbeit haben können. Es wurden vier Vorgehensmodelle in Bezug auf die Zusammenarbeit vorgestellt und verglichen. Als Ergebnis lässt sich hier festhalten, dass es auf die gesamte Entwicklungsumgebung und die Anforderungen ankommt, welches Modell geeignet ist. Aufgrund ihrer Komplexität sind Modelle wie das V-Modell oder RUP eher für größere Teams geeignet, wohingegen Scrum bereits von kleinen Teams leicht umgesetzt werden kann. Insgesamt beschäftigt sich die Forschung noch zu wenig mit konkreten Umsetzungen der Zusammenarbeit in verschiedenen Vorgehensmodellen, gerade auch in Bezug auf Werkzeuge.

Die vorgestellten Modelle und Strategien sind dabei lediglich Vorschläge, die in der Regel angepasst werden sollten. Auch wenn diverse Anbieter, gerade auch von Werkzeugen, versprechen, dass diese Out-of-the-Box funktionieren, ist dies selten der Fall. Adam Ruka [Ru22], Entwickler bei großen Unternehmen wie Amazon und Apple schreibt beispielsweise, dass die großen Tech-Unternehmen oft kein agiles Vorgehensmodell verwenden, das aber auch nicht heißt, dass sie ein Wasserfall-Modell verwenden. Stattdessen haben sie eigene Methoden und Modelle entwickelt, die oft Vorteile aus beiden Welten übernehmen und spezifisch auf die Unternehmenssituation angepasst sind. Für Unternehmen aus dem Mittelstand ist die Entwicklung eines komplett eigenen Vorgehensmodells oder Strategien für die Zusammenarbeit während der Implementierung aufgrund der Personal-Ressourcen meist nicht so leicht möglich. Hier ist dann die Orientierung an den bekannten Modellen und

einer kleineren Anpassung vermutlich der beste Ansatz. Dabei gibt es keine allgemein gültige Lösung. Das richtige Vorgehensmodell sowie die Strategie während der Implementierung hängt von den Anforderungen, der Entwicklungsumgebung und dem Team ab.

Literatur

- [AA15a] Adel, A.; Abdullah, B.: What is V-model- advantages, disadvantages and when to use it?, 2015, URL: <http://tryqa.com/what-is-v-model-advantages-disadvantages-and-when-to-use-it/>, Stand: 18. 11. 2022.
- [AA15b] Adel, A.; Abdullah, B.: What is Waterfall model- Examples, advantages, disadvantages & when to use it?, 2015, URL: <http://tryqa.com/what-is-waterfall-model-advantages-disadvantages-and-when-to-use-it/>, Stand: 18. 11. 2022.
- [AA15c] Adel Alshamrani; Abdullah Bahattab: A Comparison Between Three SDLC MODELS, 1. Jan. 2015.
- [An21] Antony, A.: 5 Different Git Workflows, 2021, URL: <https://medium.com/javarevisited/5-different-git-workflows-50f75d8783a7>, Stand: 02. 12. 2022.
- [BK08] Bunse, C.; von Knethen, A.: Vorgehensmodelle kompakt. Spektrum Akad. Verl., Heidelberg, 2008, ISBN: 3827419506.
- [Bl22] Blender Community: Blender 3.5 Nutzerhandbuch - Assets, Files & Data System, 2022, URL: <https://docs.blender.org/manual/de/dev/files/introduction.html>, Stand: 02. 12. 2022.
- [Ch11] Chacon, S.: GitHub Flow, 2011, URL: <http://scottchacon.com/2011/08/31/github-flow.html>, Stand: 02. 12. 2022.
- [CL22] Chacon, S.; Long, J.: git –local-branching-on-the-cheap, 2022, URL: <https://git-scm.com>.
- [Co22] Computer History Museum: The IEBUPDAT Program, 2022, URL: <https://www.computerhistory.org/collections/catalog/102713291>.
- [Da19] Dalton, J.: Great Big Agile, An OS for Agile Leaders. Springer Science+Business Media, New York, 2019.
- [Dr10] Driessen, V.: A successful Git branching model, 2010, URL: <https://nvie.com/posts/a-successful-git-branching-model/>.
- [DW99] Dröschel, W.; Wiemers, M.: Das V-Modell 97, Der Standard für die Entwicklung von IT-Systemen mit Anleitung für den Praxiseinsatz. Dröschel, Wolfgang (Edited by) Wiemers, Manuela (Edited by), De Gruyter Oldenbourg, München, 1999, ISBN: 978-3-486-25086-2.

- [Fo20] Fowler, M.: Patterns for Managing Source Code Branches, 2020, URL: <https://martinfowler.com/articles/branching-patterns.html>.
- [Fr09] Friedrich, J.; Hammerschall, U.; Kuhrmann, M.; Sihling, M.: Das V-Modell® XT, Für Projektleiter und QS-Verantwortliche kompakt und übersichtlich. Springer Berlin Heidelberg, Berlin, Heidelberg, 2009, ISBN: 9783642014888.
- [Ge19] Gehman, C.: How Git LFS Works: Overview of Git Large File Storage, 2019, URL: <https://www.perforce.com/blog/vcs/how-git-lfs-works>, Stand: 02. 12. 2022.
- [GN22] GNU: GNU RCS, 2022, URL: <https://www.gnu.org/software/rcc/>.
- [Go11] Goll, J.: Methoden und Architekturen der Softwaretechnik. Vieweg + Teubner, Wiesbaden, 2011, ISBN: 978-3-8348-1578-1.
- [Ho08] Hoffmann, D. W.: Software-Qualität. Hoffmann, Dirk W. (VerfasserIn), Springer Berlin Heidelberg, Berlin, Heidelberg, 2008, ISBN: 9783540763239.
- [Hö08] Höhn, R.; Rausch, A.; Broy, M.; Höppner, S.; Petrasch, R.; Biffl, S.; Wagner, R.; Hesse, W.; Bergner, K.: Das V-Modell XT, Grundlagen, Methodik und Anwendungen. Springer Berlin Heidelberg, Berlin, Heidelberg, 2008, ISBN: 9783540302506.
- [Hu15] Hughes, R.: Agile Data Warehousing for the Enterprise, A Guide for Solutions Architects and Project Leaders. Morgan Kaufmann, Amsterdam, Boston, Heidelberg, London, New York, 2015.
- [HV70] Hackman, J. R.; Vidmar, N.: Effects of Size and Task Type on Group Performance and Member Reactions. Sociometry 33/1, S. 37–54, 1970, ISSN: 00380431, URL: <http://www.jstor.org/stable/2786271>, Stand: 04. 12. 2022.
- [IB22] IBM: Quick References for IBM Mainframe Programming, 2022, URL: <https://ibmmainframes.com/references/a20.html>, Stand: 26. 11. 2022.
- [Ke16] Kenlon, S.: Hot to manage binary blobs with Git, 2016, URL: <https://opensource.com/article/16/8/how-manage-binary-blobs-git-part-7>, Stand: 02. 12. 2022.
- [Kr00] Kruchten, P.: The Rational Unified Process—An Introduction. In: the Rational edge - e-zine for the rational community. https://www.researchgate.net/publication/220018149_The_Rational_Unified_Process--An_Introduction, 2000.
- [Kr99] Kruchten, P.: Der Rational Unified Process. Eine Einführung. Addison Wesley, München, 1999.
- [Ky19] Kyeremeh, K.: Overview of System Development Life Cycle Models. SSRN Electronic Journal/, 2019.
- [Le17] LeRoy, J.: How to draw a branching diagram, 2017, URL: <https://twitter.com/jahnnie/status/937917022247120898>, Stand: 04. 12. 2022.

- [LL07] Ludewig, J.; Licher, H.: Software Engineering, Grundlagen, Menschen, Prozesse, Techniken. dpunkt-Verl., Heidelberg, 2007, ISBN: 9783898642682.
- [Mc21] McMillan, T.: A History of Version Control, 2021, URL: <https://blog.tarynmcmillan.com/a-history-of-version-control>.
- [Me18] Meyer, A.: Softwareentwicklung: Ein Kompass für die Praxis. Mode of access: Internet via World Wide Web Text (nur für elektronische Ressourcen), De Gruyter Oldenbourg, Berlin, München und Boston, 2018, ISBN: 3110575809.
- [Me22] Mercurial Community: Mercurial - Work easier - Work faster, 2022, URL: <https://www.mercurial-scm.org>.
- [Me92] Meyer, B.: Applying 'design by contract'. Computer 25/10, S. 40–51, 1992, ISSN: 0018-9162.
- [Mi22a] Microsoft: What is Team Foundation Version Control, 2022, URL: <https://learn.microsoft.com/en-us/azure/devops/repos/tfvc/what-is-tfvc?view=azure-devops>.
- [Mi22b] Misra Association: MISRA, 2/12/2022, URL: <https://www.misra.org.uk/>, Stand: 02. 12. 2022.
- [My12] Myers, G. J.: The art of software testing. J. Wiley & Sons, Hoboken, N.J., 2012, ISBN: 9781118133156.
- [re18] bitkom research: Scrum - König unter den agilen Methoden, 2018, URL: <https://www.bitkom-research.de/de/pressemitteilung/scrum-koenig-unter-den-agilen-methoden>.
- [Ru15] Ruka, A.: GitFlow considered harmful, 2015, URL: <https://www.endoflineblog.com/gitflow-considered-harmful>.
- [Ru17] Ruka, A.: OneFlow - a Git branching model and workflow, 2017, URL: <https://www.endoflineblog.com/oneflow-a-git-branching-model-and-workflow>, Stand: 26. 11. 2022.
- [Ru22] Ruka, A.: Big Tech uses neither Agile nor Waterfall, 2022, URL: <https://www.endoflineblog.com/big-tech-uses-neither-agile-nor-waterfall>, Stand: 02. 12. 2022.
- [SBZ12] Shihab, E.; Bird, C.; Zimmermann, T.: The Effect of Branching Strategies on Software Quality. In: Proceedings of the ACM-IEEE International Symposium on Empirical Software Engineering and Measurement. ESEM '12, Association for Computing Machinery, Lund, Sweden, S. 301–310, 2012, ISBN: 9781450310567, URL: <https://doi.org/10.1145/2372251.2372305>.
- [Sc05] Scholz, P.: Softwareentwicklung eingebetteter Systeme, Grundlagen, Modellierung, Qualitätssicherung. Springer-Verlag, Berlin, Heidelberg, 2005.
- [Sc12] Schneider, K.: Abenteuer Softwarequalität: Grundlagen und Verfahren für Qualitätssicherung und Qualitätsmanagement. 2012.

- [Sh19] Shiklo, B.: 8 Vorgehensmodelle der Softwareentwicklung: mit Grafiken erklärt, 2019, URL: <https://www.scnsoft.de/blog/vorgehensmodelle-der-softwareentwicklung>.
- [SS20] Schwaber, K.; Sutherland, J.: The Scrum Guide: The Definitive Guide to Scrum: The Rules of the Game, 2020, URL: <https://scrumguides.org>.
- [Sy22] Synopsys Inc.: Compare Repositories, 2022, URL: <https://www.openhub.net/repositories/compare>, Stand: 15.11.2022.
- [Th22] The Apache Software Foundation: Apache Subversion, 2022, URL: <https://subversion.apache.org>.
- [Va14] Vanshika Rastogi: Software Development Life Cycle Models-Comparison , Consequences., 2014, URL: <https://www.semanticscholar.org/paper/Software-Development-Life-Cycle-Models-Comparison-%2C-Rastogi/577cfae86ee8bd01d64783c1c6d240523bea3b03#extracted>.
- [Ve00] Versteegen, G.: Projektmanagement mit dem Rational Unified Process. Springer-Verlag, Berlin, Heidelberg, 2000.
- [Wi22] Wickner, A.: Wie ist die Komplexität von Software-Projekten jetzt und in Zukunft zu bewältigen. Informatik Aktuell/, Aug. 2022, URL: <https://www.informatik-aktuell.de/entwicklung/methoden/wie-ist-die-komplexitaet-von-software-projekten-jetzt-und-in-zukunft-zu-bewaeltigen.html>.

Software Reuse: Überblick und Anwendung

Adrian Liehner,¹ Jülf Freudenberger²

Abstract: Methoden und Werkzeuge, welche in der Entwicklung von Software Anwendung finden, unterliegen einem stetigen Wandel. Stets denken Entwickler darüber nach, wie sich der Prozess der Softwareentwicklung noch effizienter gestalten lässt. Ein in diesem Kontext häufig genannter Begriff ist **Software Reuse**.

Diese Arbeit zeigt auf, was Software Reuse ausmacht, welche Aspekte bei der Softwareentwicklung Wiederverwendet werden können und worauf bei der Wiederverwendung von Software zu achten ist. Es wird auf Software Reuse und Code Reuse eingegangen und erläutert, wie Software Reuse mit der Qualität der Software zusammen hängt. Außerdem wird auf Metriken eingegangen, die bei Software Reuse relevant sind. Des Weiteren werden einige Software Architekturmuster erläutert und deren Stärken und Schwächen bezüglich Software Reuse aufgezeigt.

1 Was ist Software Reuse?

Bei Software Reuse geht es um die Entwicklung von Software unter der Verwendung bereits bestehender Software-Komponenten. Als Komponenten können hier sowohl Quellcode, wie auch Software-Design, Schnittstellen, Anleitungen, Dokumentation oder auch Anforderungsspezifikationen verstanden werden [T422].

Software Reuse wird auch als Code Reuse oder als wiederverwendungsorientiertes Software Engineering bezeichnet.

Bei der Entwicklung von Software suchen Unternehmen Möglichkeiten, den Entwicklungsprozess zu beschleunigen und Kosten zu sparen. Deshalb ist es wichtig zu verstehen, worum es bei Software Reuse geht und wie Wiederverwendung von Software erfolgreich durchgeführt werden kann.

Historischer Kontext

Die Idee von der Wiederverwendung von Programmcode gibt es schon seit es Computer gibt, so beschrieb bereits Charles Babbage, wie eine Bibliothek an Lochkarten seiner „Analytical Engine“ Programme enthalten könnten, die wiederverwendet werden. [Wi22]

¹ DHBW Stuttgart Campus Horb, Informatik, Florianstraße 15, 72160 Horb am Neckar, Deutschland i20023@hb.dhbw-stuttgart.de

² DHBW Stuttgart Campus Horb, Informatik, Florianstraße 15, 72160 Horb am Neckar, Deutschland i20013@hb.dhbw-stuttgart.de

Allerdings geht es bei Software Reuse weniger um die Wiederverwendung von Programmen selbst, sondern mehr um die Wiederverwendung verschiedener Elemente der Softwareentwicklung wie Quellcode beim erstellen einer neuen Software.

Was heute unter Software Reuse verstanden wird, wurde erstmals von McIlroy [68] im Jahr 1968 in seinem Paper "Mass produced software components" beschrieben, welches er auf der NATO Software Engineering Conference im Jahr 1968 veröffentlichte. Darin beschreibt er die Idee von massenproduzierten Softwarekomponenten und zieht Vergleiche zu der Standardisierung von Schrauben und elektronischen Widerständen. Er zeigt auf, dass beim Erstellungsprozess neuer Software meist die Frage „Welche Mechanismen sollen wir bauen?“ statt „Welche Mechanismen sollen wir verwenden?“ zur Diskussion steht. McIlroy ist der Meinung, dass es an der Zeit ist, Software wiederzuverwenden. [68]

Mit seiner Aussage hat McIlroy völlig recht, da er im Gegensatz zu früheren Ideen der Wiederverwendung von Software durch die Technologie seiner Zeit verschiedene Möglichkeiten hat, Software öfters zu verwenden. So bot beispielsweise die in den 60-er Jahren entwickelte Objektorientierte Programmiersprache Simula [ND82] die Möglichkeit, Klassen in Bibliotheksdateien auszulagern und diese zum Zeitpunkt der Kompilierung zu verwenden.

Die Gedanken von McIlroy, Software wiederzuverwenden, werden im Laufe der Jahre in verschiedenen Formen in die Softwareentwicklung Einzug erhalten. Dazu tragen beispielsweise eine Vielzahl an Programmiersprachen bei, welche konzeptionell auf die Mehrfachverwendung von Quellcode ausgelegt sind - zum Beispiel wie bereits erwähnt über Objektorientierte Klassen oder Auslagerung in Bibliotheksdateien.

Einfluss des Internets auf Software Reuse

Die Erfindung des Internets hatte auf die Menschheit einen unberechenbaren Einfluss. Sehr viele Dinge, die zuvor nur offline möglich waren, konnten nun im Internet erledigt werden, wie Einkaufen, mit anderen Menschen kommunizieren oder das Lesen von Nachrichten oder Büchern.

In den Anfängen der Softwareentwicklung war so ziemlich jede Software in den USA öffentlich zugänglich, da Software meist hoch spezialisiert und dadurch von geringem Wert war. Erst im Laufe der Zeit entwickelte sich eine Softwareindustrie, der es um den kommerziellen Verkauf von Softwareprodukten ging.

Es war gang und gebe, dass Nutzer Software verbessern oder Fehler selbst beheben und diese selbstständig anderen über IBMs SHARE oder DECTUS zur Verfügung stellten. Software wurde physikalisch auf Magnetband, Floppy Discs oder Compact Discs ausgeliefert und transportiert.

Die heutigen Softwarereprodukte unterschieden sich stark von den aus den 80er und 90er

Jahren. Das Internet ermöglichte den Softwareunternehmen, ihre Produkte elektronisch auszuliefern.

Neue Anwendungstypen und technologische Fortschritte sorgten auch für neue Werkzeuge, die zur Anwendungsentwicklung genutzt werden konnten [Wa11]. Während Entwicklerteams zuvor noch gemeinsam an einem Ort arbeiteten ermöglichte das Internet globale Kommunikation, was Remote-Entwicklung möglich machte. So erhielten auch erste Internet-angebundene Projekte Repositorys, welche Quellcode und Dokumentation beinhalteten, Einzug in die Softwareentwicklung. Die ersten Repository-Plattformen SourceForge, Google Code und GitHub begannen hunderttausende Open Source Projekte zu hosten. Durch diese Entwicklung wurde die Wiederverwendung von Software so Mainstream wie sie heute ist.

Gerade durch Werkzeuge wie GitHub [Gi22] und Stack Overflow [St22] hat sich der Prozess der Softwareentwicklung in den letzten Jahren stark verändert. Viele Funktionalitäten die Entwickler zuvor selbst programmieren mussten, können heute einfach aus dem Internet übernommen werden. Die meisten Probleme, die beim Programmieren auftreten, hatte jemand anders bereits erlebt und es ist möglich, dass eine einfache Google-Suche ein schnelles Ergebnis liefert.

Ein ausformuliertes Konzept, wie Softwareentwickler im Internet nach Quellcode suchen, haben viele Unternehmen nicht, es ist meist dem Entwickler selbst überlassen, wie er mit dem Werkzeug, welches ihm zur Verfügung steht umgeht, und wie er Onlineressourcen in seine Entwicklung einfließen lässt. Ein Grund dafür ist, dass Entwickler unterschiedliche Kenntnisstände haben. Ist einem Entwickler die Lösung für ein spezifisches Problem möglicherweise bereits auf früheren Entwicklungen bekannt, so wird er es mit der ihm bekannten Methode lösen. Weniger erfahrene Entwickler greifen hingegen schneller zu Online-Ressourcen um ihre Probleme zu lösen.

Aktuelle Entwicklungen im Bereich der Künstlichen Intelligenz lassen vermuten, wie Softwareentwicklung in Zukunft aussehen könnte. KI-Werkzeuge, wie das auf OpenAI's GPT-3 basierte GitHub Copilot, können Softwareentwickler bei ihrer Arbeit unterstützen. Das Neuronale Netzwerk durchsucht Milliarden Zeilen an Code, um dem Entwickler eine Lösung für sein Problem zu präsentieren. Diese moderne Form von KI-basierter Softwareentwicklung könnte die automatisierte Zukunft von Software Reuse darstellen.

2 Arten von Software Reuse

Wenn wir von Software Reuse sprechen, ist es erforderlich zwischen der Wiederverwendung von Quellcode und der Entwicklung von wiederverwendbarem Quellcode unterscheiden.

Bei der Wiederverwendung von Quellcode muss der Entwickler nach Bauer [Ba16] zunächst herausfinden, für welche Funktionalität er Quellcode wiederverwenden möchte. Er muss evaluieren ob es sich überhaupt lohnt für sein Problem Quellcode wiederzuverwenden, oder ob der Aufwand zu groß ist und er den Code selbst schreiben sollte. Anschließend muss

er Programmcode finden, der sich für sein Problem anwenden lässt. Umso genauer der verwendete Code für das Projekt passt, umso weniger Aufwand hat der Entwickler bei der Anpassung und Implementation des Quellcodes in sein Projekt.

Ganz anders muss ein Entwickler vorgehen, der Quellcode entwickeln möchte, welcher möglichst wiederverwendbar sein soll. Er muss abwägen, wie viel Zeit und Mehraufwand er investieren kann, um seinen Quellcode wiederverwendbarer zu gestalten. Dabei kann er sich daran orientieren, wie oft sein Quellcode an anderen Stellen oder in anderen Projekten wiederverwendet werden kann, wie viel Wartungsaufwand der Software durch seine Verbesserungen gespart werden kann, die Anzahl der Personen, die von den Verbesserungen profitieren, und die Zeitspanne über die der Quellcode verwendet werden kann Bauer [Ba16].

Art der Artefakte, die wiederverwendet werden

Software Entwickler haben verschiedene Möglichkeiten Software wiederzuverwenden. In [Ba16] wurden in einer kleinen Umfrage Entwickler eines Entwicklerteams bei Google befragt, wie sie Wiederverwendung durchführen. Die häufigsten Antworten waren:

Answer	#Answers	Percentage
Source code	37	97%
Code in binary form	12	32%
Style guides	11	29%
UI Designs	10	26%
Requirement docs. / Use cases	5	13%
Architecture documentation	5	13%
Prototypes	2	5%
Informal design models	2	5%
Own, domain specific design models	2	5%
Semiformal design models (UML)	0	0%
Formal design models	0	0%
Other	0	0%

Abb. 1: Wiederverwendung verschiedener Artefakte

Es ist deutlich zu sehen, dass Quellcode am häufigsten wiederverwendet wird, während Design Models eher selten wiederverwendet werden. Das könnte damit zusammenhängen, dass sich die Designspezifikationen zwischen verschiedenen Projekten unterscheiden, sodass eine Wiederverwendung oft nicht sehr sinnvoll erscheint. Es ist allerdings zu beachten, dass es sich bei dieser Umfrage nur um eine Stichprobe handelt, und die Grafik nicht repräsentativ für alle Unternehmen ist.

Wichtig ist also, für das eigene Entwicklerteam eine Analyse durchzuführen, um herauszufinden, in welchem Umfang Software Reuse stattfindet. So kann festgestellt werden, ob potentiell weitere Artefakte der Softwareentwicklung wiederverwendet werden sollten.

Auch gibt es verschiedene Vorgehensweisen, wie Quellcode wiederverwendet werden kann. Auch diese wurden in [Wi96] in einer Umfrage ausgewertet. Auch diese Umfrage ist nicht repräsentativ, zeigt allerdings Möglichkeiten zur Wiederverwendung auf.

Answer	#Answers	Percentage
Software libraries	32	89%
Software frameworks	19	53%
Design patterns	13	36%
Code scavenging (copy, paste, modify)	12	33%
Component-based development	8	22%
Architecture reuse	5	14%
Product lines	1	3%
Application generators	1	3%
None	0	0%
Other	0	0%

Abb. 2: Wiederverwendung bei der Softwareentwicklung

Zunächst ist interessant, dass keiner der befragten Entwickler angab, in keiner Weise Wiederverwendung durchzuführen. Das nutzen von Softwarebibliotheken liegt mit großem Abstand vorne, hingegen landet das Kopieren und Einfügen von Quellcode lediglich auf dem vierten Platz. Als eine grundlegende Erkenntnis lässt sich erkennen, dass das Wiederverwenden von Software in verschiedensten Formen Bestandteil der Vorgehensweise von Entwicklern und aus dem Alltag nicht mehr wegzudenken ist.

Für Entwickler ist es wichtig, selbst zu Analysieren, auf welche Arten Software wiederverwendet wird. So können Potentiale für Software Reuse erkannt werden.

Die verschiedenen Arten Software wiederzuverwenden sind auch mit unterschiedlich viel Aufwand verknüpft. Während einfaches Kopieren von Code in Sekundenschnelle durchgeführt werden kann, benötigt hingegen die Wiederverwendung einer Softwarearchitektur eine strukturierte Planung und viel Zeit. Je nachdem was wiederverwendet werden soll, müssen unterschiedliche Personen von Entwicklungsteams mit einbezogen werden. Mit steigender Komplexität der Wiederverwendung wächst die Schwierigkeit, die Wiederverwendung erfolgreich und effizient durchzuführen.

3 Voraussetzungen, Chancen und Risiken von Software Reuse

Vorbereitung für systematische Wiederverwendung

Nach Frakes and Fox [Wi96] gibt es bei der Wiederverwendung von Software verschiedene Aspekte zu beachten, darunter Management und Wirtschaftlichkeit.

Wenn Software systematisch wiederverwendet werden soll, muss das Management der Organisation, welche die Software entwickelt, Anreize dafür schaffen. Eine Möglichkeit Anreize zu schaffen wäre den monetären Wert aufzuzeigen, welchen eine konkrete Entwicklung eines wiederverwendbaren Softwareteils schafft.

Indem festgehalten wird, wie oft Software wiederverwendet wird, könnten Organisationen die Entwicklung wiederverwendbarer Softwarekomponenten wirtschaftlich begründen.

Technische Voraussetzungen für erfolgreiche Wiederverwendung von Software

Damit die Wiederverwendung von Software erfolgreich sein kann, müssen nach Bauer [Ba16] bestimmte Rahmenbedingungen gegeben sein. So sollte eine adäquate Infrastruktur vorliegen. Das Projekt an dem gearbeitet wird benötigt eine Art von Quellcodeverwaltung und Versionierung, die es ermöglicht, die einzelnen Entwicklungsschritte nachvollziehen zu können. Außerdem sollte die Software dokumentiert werden, sodass andere Personen, die ebenfalls an der Software arbeiten, leichter verstehen, welche Softwarekomponente wie arbeitet. Bei der Entwicklung sollten die Entwickler auf gute Softwarequalität achten, welche die Lesbarkeit verbessert und Veränderungen am Code, wie zum Beispiel das Refactoring, möglichst unkompliziert zulässt.

Chancen von Software Reuse

Wiederverwendung von Software findet in Organisationen nur dann statt, wenn daraus Vorteile gezogen werden können. Es gibt verschiedene Aspekte welche sich, je nach Strategie und Umsetzung, durch die Wiederverwendung von Software positiv verändern können. So kann die Entwicklungszeit durch Wiederverwendung verkürzt werden, was wiederum die Kosten für die Entwicklung senkt. Je nach Anwendungsszenario können Fehler im Code vermieden werden, indem gut getestete Softwareteile wiederverwendet werden. So kann eine höhere Softwarequalität gewährleistet werden. Bekannte Fehler sind bereits behoben und Softwaretests bereits durchgeführt, wodurch das Endprodukt robuster wird.

Risiken von Software Reuse

Wenn die systematische Wiederverwendung von Software nicht richtig durchgeführt wird, kann diese auch zum Nachteil für die Entwickler und die Organisation werden.

Aufgezwungene Wiederverwendung von Software kann dazu führen, dass veraltete Technologien zu lange verwendet werden. Dies kann passieren, wenn beispielsweise Aktualisierungen von Frameworks oder Bibliotheken Neuentwicklungen erfordern würden, für welche entwe-

der nicht das Budget oder das Entwickler-Know-how vorhanden ist. Das Potential, welches durch Neuentwicklung mittels aktuellerer Technologien kommt, sollte nicht außer Acht gelassen werden.

Je nach Anwendungsszenario kann Wiederverwendung von Software auch zu Performanceinbußen führen. Die Wiederverwendung generischer Komponenten kann zu höherer Speichernutzung führen, wie es eine spezifische Neuentwicklung tun würde.

Zudem können falsche Praktiken bei der Wiederverwendung von Software dazu führen, dass der Code komplizierter zu warten und für neue Entwickler schwerer zu verstehen ist.

Reuse Failure Modes Model

Das Reuse Failure Modes Model wurde in [Wi96] erstmals beschrieben und veröffentlicht. Das Modell zeigt verschiedene Faktoren auf, die für erfolgreiche Wiederverwendung gegeben sein müssen.

Um zum Erfolg zu gelangen, muss die in [Wi96] beschriebene siebenschrittige Reuse Success Chain abgearbeitet werden. Es gibt verschiedene Gründe, weshalb die Wiederverwendung in den einzelnen Schritten fehlschlagen kann.

1. Es muss ein Versuch stattfinden, Software wiederzuverwenden.

Es gibt eine Vielzahl an Gründen, weshalb Software Reuse von Punkt eins an nicht stattfindet. Die Wiederverwendung scheitert an diesem Punkt wenn die Ressourcen, welche für die Entwicklung verwendet werden dürfen oder können beschränkt sind, oder an erster Stelle kein Anreiz für die Wiederverwendung gegeben ist. Für die Wiederverwendung muss Zeit zur Verfügung stehen und es muss ein klarer Nutzen aus der Wiederverwendung hervorgehen. Die Entwickler benötigen ausreichende Kenntnisse und Bildung, um die Wiederverwendung durchzuführen. Die Wiederverwendung kann außerdem an unzureichender Kommunikation scheitern. Auch Rechtliche Probleme können zum Scheitern der Wiederverwendung führen. Das Management muss die Entwicklung unterstützen und die Organisation der Entwicklung muss die Wiederverwendung ermöglichen. Zudem kann es passieren, dass die Kunden des Projektes das wiederverwenden von Software nicht möchten. Zu den selteneren Problemen, weshalb kein Versuch Software wieder zu verwenden stattfindet, gehören das NIH-Syndrom (Unternehmen erlauben es nicht, Software von Externen Quellen einzusetzen), Ego-Probleme der Entwickler oder unausgereifte Wiederverwendungstechnologie.

2. Die Software, die wiederverwendet werden soll muss existieren

Sollte die Software nicht für Wiederverwendung ausgelegt sein, kein Wirtschaftlicher Anreiz für Wiederverwendung existieren oder es sich um eine komplett neuartige Technologie handeln, scheitert das Wiederverwenden von Software daran, dass keine Software existiert, welche Sinnvoll wiederverwendet werden könnte.

3. Die Software, die wiederverwendet werden soll muss verfügbar sein

Das wiederverwenden von Software kann daran scheitern, dass keine Software zur Wiederverwendung verfügbar ist, weil kein geeigneter Aufbewahrungsort für wieder verwendbare Software zur Verfügung steht. Das scheitern aufgrund von mangelnder Verfügbarkeit droht außerdem bei proprietärer oder klassifizierter Software oder wenn der Import der Software nicht möglich ist.

4. **Der Entwickler muss das Softwareteil finden, das er wiederverwenden möchte**
Damit ein Entwickler Software wiederverwenden kann, muss er diese zunächst finden. Damit ein Entwickler Software finden kann, muss diese Ausreichend Repräsentiert sein. Außerdem müssen einem Entwickler ausreichen Werkzeuge zur Verfügung stehen, mit welchen er Software zur Wiederverwendung finden kann.
5. **Der Entwickler muss das Softwareteil verstehen**
Ist die Dokumentation der Software, welche ein Entwickler wiederverwenden möchte, unzureichend oder die Software allgemein zu komplex, kann dies dazu führen, dass der Entwickler die Wiederverwendung nicht durchführt.
6. **Das zu integrierende Teil der Software muss valide sein und den Qualitätsansprüchen seiner Entwicklung entsprechen**
Der Entwickler hat sich nun für Eine Software zur Wiederverwendung entschieden und die Funktionsweise dieser verstanden.
Auch hier gibt es verschiedene Gründe, weshalb die Wiederverwendung von Software scheitern kann, zum Beispiel wenn Entwickler eine schlechte Auswahl der wiederverwendeten Software Treffen oder wenn Tests die Software als Invalide abschreiben. Die Wiederverwendung kann auch daran scheitern, dass der Hersteller der zu verwendenden Software einen zu schlechten Support bietet, dass die Wiederverwendung langfristig bestand halten kann. Auch bei Inadäquater Leistung der Software, dem Verfehlten von Standards oder unzureichender Information zur Funktionsweise der Software kann die Wiederverwendung abgebrochen werden.
7. **Das zu integrierende Teil der Software muss integriert werden können**
Die Integration von Wiederverwendeter Software kann an einer Inkompatiblen Entwicklungsumgebung oder an unpassender Form scheitern. Inkompatible Hardware kann auch für das scheitern der Wiederverwendung verantwortlich sein. Werden zu viele Anpassungen benötigt oder werden nicht-funktionale Spezifikationen nicht eingehalten, kann die Wiederverwendung ebenfalls scheitern.

Es gibt eine Vielzahl an Gründen, weshalb die Wiederverwendung von Software scheitern kann. Von den genannten Gründen mögen manche intuitiv sein, jedoch lassen sich viele Gefahren minimieren, wenn sich die Entwickler und das Management Zeit nehmen, um sich Gedanken darüber zu machen, wie sie Software wiederverwenden.

4 Metriken für Software Reuse

Die Wiederverwendung von Software kann eine effektive Möglichkeit sein, um zukünftig Zeit und Kosten zu sparen. Allerdings ist es von großer Wichtigkeit, einige Metriken zu berücksichtigen, um sicherzustellen, dass die Software effektiv wiederverwendet werden kann. Folgende Metriken könnten verwendet werden:

Funktionale Richtigkeit:

Bei der Funktionalen Richtigkeit geht es darum, ob die Software genau das tut, was sie tun sollte. Die Funktionalität einer Software wird in der Design-Spezifikation festgelegt. Wird Software zur Wiederverwendung entwickelt, muss der Entwickler darauf achten, dass die Anforderungen an seine Software exakt erfüllt werden. Wird Software wiederverwendet, muss beachtet werden, dass verwendete Softwareteile, wie Bibliotheken, möglichst genau die vorgesehenen Funktionalitäten ausführen.

Die Funktionale Richtigkeit von Software ist essentiell für die Wiederverwendung, da Entwickler, welche Software wiederverwenden, davon ausgehen müssen, dass die Software korrekt funktioniert. Die wiederverwendeten Entwickler haben meist weder Know-how, Zeit oder Buget, um Fehler in der wiederverwendeten Software zu finden und zu beheben - es ist schlicht nicht ihre Aufgabe.

Zuverlässigkeit und Robustheit:

Software ist zuverlässig, wenn eine korrekte Ausführung garantiert werden kann. Dazu gehört beispielsweise, dass sich der Nutzer auf eine korrekte Ausgabe verlassen kann. Um zuverlässige Software zu entwickeln, müssen potentielle Fehlerquellen behoben werden, sodass die Software keine Fehlverhalten oder Abstürze hervorruft.

Unter der Robustheit versteht sich die Fehlertoleranz der Software. Werden beispielsweise fehlerhafte Eingaben getätigt, bleibt eine robuste Software dennoch funktionsfähig.

Ist Software zuverlässig und robust, kann sie einfacher und besser wiederverwendet werden.

Benutzerfreundlichkeit:

Mit der Benutzerfreundlichkeit wird die Einfachheit und Intuitivität der Software gemessen. Die Benutzerfreundlichkeit ist ein wichtiger Aspekt bei der Auswahl von Software. Das Wiederverwenden von UI-Bibliotheken oder unternehmensspezifischen Designrichtlinien kann dazu beitragen, die Benutzerfreundlichkeit zu verbessern. Es gibt verschiedene Testverfahren, die verwendet werden können, um die Benutzerfreundlichkeit zu überprüfen, welche in allen Phasen des Software-Lebenszyklus durchgeführt werden sollten.

Die Benutzerfreundlichkeit trifft nicht auf jeden Fall von Software Reuse zu, jedoch sollten Entwickler von wiederverwendbarer Software stets darauf achten, dass ihre Software ohne große Mühen oder Veränderungen wiederverwendbar ist.

Flexibilität und Portierbarkeit:

Dieser Aspekt wird vor allem bei der Entwicklung von wiederverwendbarer Software relevant und ist ein Maß für die Fähigkeit der Software, sich an veränderte Umgebungen und verschiedene Plattformen anzupassen. Flexible und portierbare Software kann leichter in anderen Projekten wiederverwendet werden. Welche Schritte genau erforderlich sind, um eine Software portierbar zu gestalten, hängt von dem Kontext und der Art der Entwicklung ab, welche durchgeführt wird.

5 Einfluss der Software Architektur

Das folgende Kapitel befasst sich mit der Definition, den geforderten Eigenschaften und einigen Beispielen an Software Architekturen. Inhaltlich wurden zu diesem Thema die wichtigsten Punkte aus der Literatur von Appelrath [Ap12], Bauer [Ba08], Dowalil [Do20], Goll [Go11][Go14], Starke; Hruschka [SH11][St20] und Tremp [Tr21] gesammelt und zusammengefasst.

Am Beispiels der Objektorientierten Programmierung, besteht der Code einer Software in der Regel aus einer Vielzahl an Klassen. Diese Klassen bestehen wiederum aus Methoden und Feldern. Viele dieser Klassen interagieren miteinander, indem sie untereinander Objekte der Klassen erstellen und auf deren Methoden zugreifen. Anhand wie diese einzelnen Klassen miteinander agieren und voneinander abhängig sind, lässt sich die Architektur der Software beschreiben.

Aus umgekehrter Sicht umfasst die Architektur die statischen - welche die Verteilung der einzelnen Komponenten - sowie die dynamischen Eigenschaften - welche die Interaktionen aller Komponenten untereinander beschreibt - der Software. Diese Eigenschaften müssen das Ziel verfolgen die geforderten Funktionalitäten bereitzustellen. Für die Entwickler der entsprechenden Software kann die geplante Architektur auch als Orientierung bei der Implementierung dienen, indem sie die Funktionalitäten der Software in abstrakter Weise darstellt.

Eine geeignete Software Architektur muss in der Lage sein die Erfüllung aller geforderten funktionalen und nicht funktionalen Anforderungen sicherstellen zu können. Die Wahl der Software Architektur hat große Auswirkung auf die Wartbarkeit und auch auf die Implementierung an sich der zu entwickelnden Software.

In der Literatur existiert eine Vielzahl an nicht funktionalen Anforderungen auf die in dieser Arbeit nicht voll umfassend eingegangen wird. Stattdessen werden auf die Qualitätskriterien eingegangen, welche für das Thema Software Reuse hauptsächlich relevant sind. Sie zeigen große Ähnlichkeit zu den Metriken aus dem vorherigen Kapitel auf. Die in dieser Arbeit thematisierten Kriterien sind im folgenden aufgelistet:

Änderbarkeit und Erweiterbarkeit. Ist es bei einem bestehenden System einer Software Architektur ohne großen Aufwand möglich Änderungen durchzuführen oder Erweiterungen hinzuzufügen, so weist das System eine hohe Flexibilität auf. Systeme mit niedriger Flexibilität können im schlimmsten Fall nicht im Rahmen eines angemessenen Arbeitsaufwands weiterentwickelt werden, falls die internen Abhängigkeiten der Komponenten eine komplette Überarbeitung des gesamten System nach sich ziehen.

Testbarkeit. Das System einer Software Architektur sollte stets eine ausreichende Testbarkeit aufweisen, um die Erfüllung sämtlicher geforderten Funktionalitäten prüfen zu können. Bietet die Architektur einer Software keine Möglichkeit alle Funktionalitäten zu testen sollte sie überarbeitet oder im schlimmsten Fall verworfen werden.

Verständlichkeit. Eine Architektur, als Beschreibung der statischen und dynamischen Eigenschaften einer Software, sollte dessen Struktur in verständlicher Weise aufzeichnen. Dies ist vergleichbar mit dem später beschriebenen Einfluss der Lesbarkeit eines Codes.

Wartbarkeit. Dieses Kriterium ist stark mit den Kriterien der Verständlichkeit, der Einfachheit, der Korrektheit und der Erweiterbarkeit gekoppelt. Ist eines der genannten Kriterien nicht erfüllt, so ist es für die Entwickler schwierig mit dem Code der Software zukünftig zu arbeiten.

Wiederverwendbarkeit. Die Wiederverwendbarkeit ist wiederum stark mit den Kriterium der Änderbarkeit und Erweiterbarkeit verknüpft. Ist die Struktur einer Software unflexibel, kann sie im schlimmsten Fall an künftige Anforderungen nicht angepasst und wiederverwendet werden. Dieses Kriterium entspricht dem Kernthema dieser Arbeit.

Wie in vielen Fällen existiert auch für die Wahl der Software Architektur keine allumfassende Lösung und ist von der Anwendung der zu entwickelnden Software abhängig. Je nach Anwendung existieren verschiedene Muster an Architekturen, unter denen manche die Anforderungen besser erfüllen als andere. Welches und ob eines der existierenden Muster verwendet werden sollen ist situativ zu klären und gut abzuwägen.

Die Nutzung von einem Architekturmuster bietet die **Vorteile**, dass der Aufwand einer komplett eigens entwickelnden Architektur sich deutlich reduziert und sich das Entwicklungsteam auf die Realisierung der angestrebten Software Architektur konzentrieren kann. Außerdem bieten viele der Muster, anhand des Anwendungsfalls, die Möglichkeit einer guten Wiederverwendung der Software. Logischerweise ist der **Nachteil** der, dass die Verwendung eines Musters nicht immer einem zu entwickelnden System gerecht werden kann, da eine Architektur mit seinen Regeln auch eine Einschränkung darstellen kann.

Auch werden die sogenannten SOLID-Prinzipien häufig in Kombination mit den Anforderungen an eine gute Software Architektur erwähnt. Da diese Prinzipien auch sehr ins atomare der Codierung gehen, werden sie im späteren Kapitel des Einflusses der Lesbarkeit des Codes genauer erläutert. Abschließend kann zusammengefasst werden, dass die für den Anwendungsfall geeignete Architektur als Bindeglied zwischen den Anforderungen und der

implementierten Lösung fungiert. Folglich gilt es die einzelnen Architekturmuster genauer zu betrachten.

Zunächst können die vielen Architkurmuster im Groben, anhand ihres Aufbaus, eingeteilt werden. Innerhalb dieser Gruppen unterscheiden sich die einzelnen Muster anhand von Merkmalen, die sie für diverse Anwendungen spezialisieren. Gernot Starke hat in diesem Zusammenhang eine informative Zusammenstellung:

Datenflusssysteme. Sie beschreiben in einer Aneinanderreihung von Operationen die Datenverarbeitung und Datenflussrichtung.

Datenzentrische Systeme. Sie beschreiben die Entnahme und die Verwendung eines Datenbestand, der zentral für alle Aktionen zugänglich ist.

Hierarchische Systeme. Sie beschreiben die Aufteilung des Gesamtsystem in hierarchische Ebenen.

Verteilte Systeme. Sie beschreiben die Aufteilung in Bausteine für das Speichern und die Datenverarbeitung.

Ereignisbasierte Systeme. Sie beschreiben die Kommunikation zwischen einzelnen, voneinander unabhängigen Bausteinen.

Interaktionsorientierte Systeme. Sie beschreiben die Systeme von graphischen Oberflächen.

Heterogene Systeme. Sie beschreiben die Verwendung von mehreren verschiedenen Systemen.

Bezogen auf die Wiederverwendung von Software, werden die Vor- und Nachteile einiger Software Architekturen aufgezählt und erläutert.

Die **Batch-Sequentiell Architektur** gehört zu den Dateflusssystemen und verarbeitet streng sequentiell die Eingangsdaten der Aneinanderreihung von stark aufeinander abgestimmten Operationen. Dies bedeutet, dass der nachfolgende Operationsbaustein erst mit der Datenverarbeitung beginnt, wenn sein Vorgänger seine Operationen komplett abgeschlossen hat. Folglich ist diese Architektur in seiner Grundstruktur nicht für parallele Verarbeitungen und für Verzweigungen vorgesehen.

Aufgrund dieser Eigenschaften gestaltet sich diese Architektur als besonders einfach. Die Schnittstellen des Systems können hier ein stark deterministischem Verhalten zeigen. Jedoch ist es ratsam einen externen Baustein zu entwickeln, der zentral die Steuerung der einzelnen Operationen und die Fehlerbehandlung übernimmt.

Zusammenfassend kann gesagt werden, dass die Batch-Sequentiell Architektur, bezogen auf die Wiederverwendbarkeit, ein zu stark gekoppeltes Verhalten zwischen den einzelnen Bausteinen zeigt. Diese Abhängigkeiten könnten einen Austausch, Änderung oder eine Erweiterung von einzelnen Bausteinen erschweren. Sie mag zwar in der Implementierung

für einen genauen Anwendungsfall genau die richtige Wahl zu sein, jedoch überwiegen die Nachteile, wenn es darum geht eine wiederverwendbare Architektur zu verwenden. Folglich wird, anhand der gesammelten Beschreibungen, diese Architektur **nicht empfohlen**.

Die Pipes and Filter Architektur gehört, ebenso wie die der Batch-Sequentiell, zu den Datenflusssystemen. Im Vergleich zu zum Batch-Sequentiell werden bei der Pipes and Filter Architektur die einzelnen Bausteine (Filter) stärker voneinander unabhängig gekoppelt und die Datenverarbeitung der einzelnen Bausteine muss sich ebenfalls nicht streng sequentiell verhalten. Die Bezeichnung Pipe steht für die Datenverbindung zwischen den einzelnen Filtern.

Zusätzlich zu der Einfachheit der Struktur, wie sie auch bei der Batch-Sequentiell Architektur vorliegt, sorgt die unabhängiger Kopplung der einzelnen Bausteine für die Möglichkeit in ebenfalls einfacherer Weise einzelne Bausteine zu bearbeiten oder auszutauschen. Außerdem wird die streng sequentielle Datenverarbeitung der Batch Sequentiell Architektur aufgeweicht und es können auch Verzweigungen hinzugefügt werden, sodass mehr Möglichkeiten in Form von Schnittstellen möglich sind.

Durch diese gewonnenen Vorteile verlieren sich jedoch folglich auch einige Vorteile der Batch-Sequentiell. Ein Beispiel ist, dass das Verhalten der Schnittstellen und dementsprechend auch die Fehlerbehandlung deutlich komplexer werden kann. Benötigt der Benutzer eine interaktive Möglichkeit die Datenverarbeitung zu beeinflussen, eignen sich diese Systeme jedoch nicht, aufgrund der immer noch relativ strikten Art der Datenverarbeitung.

Zusammenfassend kann gesagt werden, dass die Pipes and Filter Architektur, **außerhalb von interaktiven Systemen**, sich durchaus als eine **bedingt wiederverwendbare Architektur** erweisen. Aufgrund der Austauschbarkeit und Änderbarkeit von einzelnen Filtern kann die Software in einfacher Weise für weitere Anwendung angepasst werden. Eine **Voraussetzung** für eine Wiederverwendbarkeit ist jedoch, dass eine ausreichende Fehlerbehandlung bereits realisiert wurde, um das Kriterium der Testbarkeit erfüllen zu können.

Die **Blackboard Architektur** gehört zu den Datenzentrischen Systemen und beruht auf der Idee, dass sich in der Mitte des Systems das Kontrollelement namens Blackboard befindet und mit vielen voneinander unabhängigen Bausteinen verbunden ist. Jeder dieser Bausteine hat eine spezielle Aufgabe, sodass eine komplexe Aufgabe vom Blackboard auf die verschiedenen Bausteine verteilt werden kann. Auf diese Weise wird jeder dieser Bausteine eine Lösung für das erhaltene Teilproblem liefern und dem Blackboard zurückgeben.

Durch die Unabhängigkeit der einzelnen Bausteine ist es einfach weitere Bausteine der Architektur hinzuzufügen und es ist eine parallele Datenverarbeitung möglich. Ist jedoch eine Synchronisierung zwischen den Bausteinen erforderlich, könnte sich dieses Vorhaben bei dieser Architektur als schwierig erweisen. Außerdem kann sich eine Fehlerlokalisierung ebenfalls als kompliziert erweisen, da es bei der Zusammenführung der einzelnen Teillösung im Blackboard unter Umständen nicht ersichtlich ist, wo der Fehler entstanden ist.

Zusammenfassend kann gesagt werden, dass die Blackboard Architektur durchaus das Potential hat als **bedingt wiederverwendbare** Architektur verwendet zu werden. Durch die einfache Erweiterbarkeit der Architektur lässt sie sich ohne großen Aufwand an neue Anforderungen anpassen. Jedoch gilt auch wie bei der Pipes and Filter Architektur die **Voraussetzung**, dass eine ausreichende Fehlerbehandlung implementiert ist, um die Testbarkeit sicherzustellen.

Die **Master-Slave Architektur** gehört zu den Hierarchischen Systemen und besteht wie die anderen Architekturen aus mehreren Blöcken. Ein Block, der sogenannte Master, ruft Funktionen der anderen voneinander unabhängigen Blöcke, Slaves genannt, auf. Anders als bei den zuvor genannten Architekturen, haben die Slave-Blöcke der Master-Slave Architektur alle die gleichen Funktionen und sind folglich redundant. Durch die Unabhängigkeit der einzelnen Slave-Blöcke können diese auch parallel arbeiten, weswegen diese Architektur bevorzugt in Systemen mit hoher Verfügbarkeitsanforderung eingesetzt wird.

Durch die hohe Redundanz gestaltet sich der Aufwand für eine potentiell benötigte Änderung als entsprechend hoch. Aus diesem Grund wird diese Art der Architektur bezüglich der Wiederverwendbarkeit **nicht empfohlen**.

Ebenso wie die Master-Slave Architektur gehört die **Schichten Architektur** zu den Hierarchischen Systemen. Über eine bestimmte Anzahl an übereinander stehenden Schichten werden Gruppen an Funktionalitäten definiert und festgelegt. Jede Schicht bietet der jeweils höher liegenden Schicht Dienste an und kann auf die Dienste der jeweils niedriger liegende Schicht zugreifen. Auf diese Weise abstrahiert jede Schicht die Sicht für die jeweils höhere Schicht.

Durch die Zusammenfassung der "ähnlichen" Komponenten innerhalb einer Schicht, wird die Struktur der Software für die Entwickler deutlich verständlicher und hilft die Abhängigkeiten über konkret definierte Schnittstellen zwischen den Schichten zu reduzieren. Außerdem wird die Austauschbarkeit deutlich erhöht, wenn an eine Schicht neue Anforderungen gestellt wird. Ein Nachteil ist jedoch, dass durch das erforderliche Durchlaufen von mehreren Schichten, eine Beeinträchtigung der Performance zu erwarten ist. Je nach Art der neuen Anforderung an das System, kann es erforderlich sein, dass mehrere oder sogar alle Schichten angepasst werden müssen.

Zusammenfassend kann gesagt werden, dass die Schichten Architektur ebenso zu den **bedingt wiederverwendbaren** Architekturen gezählt werden kann. Durch die Aufteilung in mehreren Schichten mit lediglich Abhängigkeiten in eine Richtung ist es bis zu einem gewissen Grad möglich Änderungen oder Erweiterungen innerhalb einer Schicht ohne großen Aufwand durchzuführen. **Voraussetzung** ist jedoch, dass sich der Änderungsaufwand auf eine überschaubare Anzahl an Schichten begrenzt.

Die **Ports-und-Adapter Architektur** ist ebenfalls Teil der Hierarchischen Systeme . Im Mittelpunkt der Architektur steht die sogenannte Fachdomäne, welche den festen Kern darstellt, sämtliche Basisfunktionalitäten beinhaltet und eine komplett unabhängige

Komponente darstellt. Über definierte Schnittstellen der Fachdomäne, genannt Ports, können externe Systeme über ihre Schnittstellen, genannt Adapter, auf die Fachdomäne zugreifen und die Basisfunktionalitäten für ihre anwendungsspezifischeren Funktionen nutzen. Die Fachdomäne sollte so entwickelt werden, dass zukünftig keine Änderungen mehr erforderlich sein sollten, da die externen Systeme von ihr abhängig sind.

Unter der Annahme, dass es möglich ist eine Fachdomäne zu realisieren, die sämtliche aktuellen und zukünftigen Anforderungen erfüllt, wäre sie eine sehr gute Basis für eine wiederverwendbare Architektur. Jedes einzelne externe System könnte ohne den Einfluss auf die anderen System und die Fachdomäne geändert oder erweitert werden. Jedoch ist anzunehmen, dass es als sehr unwahrscheinlich einzustufen ist, dass sämtliche zukünftigen Anforderungen an die Fachdomäne zum Zeitpunkt der Entwicklung erfüllt werden können. Kommt es zu dem Fall, dass eine Änderung an der Fachdomäne vorzunehmen ist, kann dies zusätzlich im schlimmsten Fall eine komplette Überarbeitung aller externen Systeme nach sich ziehen. Folglich wird die Architektur der Ports-und-Adapter als **kurz- bis mittelfristig wiederverwendbar** eingestuft in dem Zeitrahmen, wo die Anforderungen noch ermittelbar sind.

Die **Broker Architektur**, als Teil der Verteilten Systeme, beschreibt eine Sammlung von mehreren Clients und Servern, die über eine Anlaufstelle, genannt Broker, miteinander verbunden werden. Anfragen von Clients werden durch den Broker an den entsprechenden Server weitergeleitet. Auf diese Weise ist eine direkte Kommunikation zwischen den Clients und Servern nicht möglich und der Broker hat die volle Kontrolle über die Kommunikation. Durch die kontrollierte Verknüpfung von mehreren Clients und Servern kann die Rechenleistung auf mehrere Rechner verteilt werden.

Die **Vorteile** der Broker Architektur sind, dass durch die Verteilung des Systems auf mehrere Rechner das System selbst skalierbar ist und, unter der Voraussetzung, dass sich die Schnittstellen zum Broker nicht ändern, können die Server-Implementierungen frei geändert werden. Außerdem ist es für einen Client nicht erforderlich, über den physischen Ort des angefragten Dienstes in Kenntnis zu sein, da dies durch den Broker gewährleistet ist.

Die resultierenden **Nachteile** sind jedoch, dass bei einem Fehler innerhalb des Brokers alle verbundenen Clients und Server ebenfalls betroffen sind. Folglich muss der Broker über ein gewisses Maß an Fehlertoleranz verfügen. Außerdem stellt der Broker durch seine Funktionalität das Nadelöhr der Architektur dar und hat folglich gegenüber anderen Architekturen Performanceeinbußen.

Auch wenn die Broker Architektur seinen Einsatz bei der Client-Server-Verknüpfung hat, könnte diese Architektur auch für einzelne Software-Lösungen angewendet werden. Beispielsweise könnten eine Broker-Klasse die Schnittstelle für sämtliche möglichen Klassen und Komponenten darstellen unter der **Voraussetzung**, dass die Performance im Rahmen bleibt. Da auf diese Weise der Broker der Fachdomäne aus der Ports-und-

Adapter Architektur ähnelt, wäre die Broker Architektur jedoch ebenfalls nur als **kurz- bis mittelfristig wiederverwendbar** anzusehen.

Die **Model-View-Control Architektur** gehört zu den Interaktionsorientierten Systemen. Sie trennt das System in genau drei Bausteine, Model, View und Control, mit unterschiedlichen Verantwortlichkeiten. Der Model-Baustein beinhaltet die Dienste, welche dem Benutzer zur Verfügung stehen. Der View-Baustein stellt die sichtbare Benutzeroberfläche, anhand der im Model gesetzten Gegebenheiten, für den Benutzer dar. Der Control-Baustein leitet die Benutzereingaben und -anfragen an das Model weiter, welches die Anfragen bearbeitet und dem View die entsprechend angepassten Daten und angefragten Dienste zur Darstellung zur Verfügung stellt. Auf diese Weise werden die Verantwortlichkeiten getrennt und die Interaktionen der Bausteine über kontrolliert definierte Schnittstellen realisiert.

Der große **Vorteil** der Model-View-Control Architektur bestehen darin, dass die einzelnen Bausteine, unter Beibehaltung der Schnittstellen, unabhängig voneinander entworfen und geändert werden können. Eine Änderung innerhalb der Benutzeroberfläche muss nicht zwangsläufig eine Änderung der Daten im Model nach sich ziehen. **Nachteil** ist jedoch, dass durch die benötigte Weiterleitung der Benutzeranfragen im View über das Control bis hin zum Model die Performance leiden kann.

Bei Software Programmen mit Benutzeroberflächen bietet die Model-View-Control Architektur ähnliche Vorteile wie die Schichten-Architektur mit Ausnahme, dass die Abhängigkeiten nicht hierarchisch nur in eine Richtung gehen. Nichtsdestotrotz kann diese Architektur als **bedingt wiederverwendbar** betrachtet werden, aufgrund der Änderbarkeit der Komponenten der einzelnen Bausteine, unter der **Voraussetzung**, dass sich die Schnittstellen zwischen den Bausteinen nicht ändern.

6 Einfluss der Lesbarkeit des Codes einer Software

Doch selbst die Wahl der best passendsten Software Architektur reicht nicht aus, um Elemente einer Software wiederverwendbar zu gestalten.

Das folgende Kapitel befasst sich mit der Relevanz der Lesbarkeit von Software Code und den Möglichkeiten diese zu verbessern. Inhaltlich wurden zu diesem Thema die wichtigsten Punkte aus der Literatur von Kapil [Ka19], Martin [Ma09] und Roth [Ro21] gesammelt und zusammengefasst.

Um Änderungen von Software Code durchführen zu können, ist es erforderlich die innere Struktur und deren Verhalten zu kennen und zu verstehen. Durch das Verständnis über das Verhaltens ist der Entwickler in der Lage dieses Verhalten zu bewahren oder in kontrollierter Weise anzupassen. Eine hohe Lesbarkeit des Codes unterstützt den Entwickler den Code zu verstehen.

Ist die Lesbarkeit eines Codes hingegen kaum oder gar nicht gegeben, so benötigt der Entwickler mehr Zeit, um die Funktionsweise des Codes in Erfahrung zu bringen bevor er überhaupt seine eigentliche Änderung am Software Code durchführen kann. Oftmals wird zu Beginn der Entwicklung einer Software der Fokus auf die Funktionalität gesetzt und wenig Augenmerk auf die Lesbarkeit gesetzt. Da dies am Anfang einer Software, welche noch relativ wenig Zeilen aufweist, noch relativ wenig ins Gewicht fällt, ist das Problem der schlechten Lesbarkeit erst bei zukünftigen Änderungen spürbar. Mit der kontinuierlich wachsenden Anzahl an Zeilen im Code, steigt der Aufwand den Code in seiner Gänze zu kennen und zu verstehen.

Folglich ist eine gute Lesbarkeit eines Software Codes nicht nur nützlich, sondern für die Wiederverwendbarkeit einer Software höchst erforderlich. Es existiert eine Vielzahl an Möglichkeiten die Lesbarkeit eines Codes zu verbessern. Manche betreffen ganze Bereiche des Codes, andere gezielt einzelne Code-Zeilen.

Um eine Übersichtliche **Klasse** zu gestalten, sollten direkt **zu Beginn vor den Methoden sämtliche Felder** der Klasse aufgelistet werden. Existieren Felder mit unterschiedlichen Schutzgraden, so sollten die **öffentlichen Felder vor den privaten** platziert werden. Dies hat den Vorteil, dass die Entwickler die von außen erreichbaren Felder schnell auffinden können. Sind statische wie nicht statische Felder vorhanden, so sollten die statischen, unabhängig vom Schutzgrad, vor den nicht statischen Feldern stehen. Dies hat wiederum den Vorteil, dass sich die nicht statischen Felder näher an den Methoden befinden, welche die Felder i.d.R. häufiger nutzen.

Nach den Feldern kommen die Funktionen, oft auch Methoden genannt. Auch wenn es sich hier ebenfalls möglich ist die öffentlichen Methoden vor den privaten aufzulisten, bietet es sich im Falle der **Reihenfolge der Methoden** an, dass private Methoden direkt hinter den öffentlichen Methoden platziert werden, welche die privaten Methoden aufrufen. Dies sorgt dafür, dass die Methoden für den Entwickler leichter von oben nach unten lesbar sind und weniger die Leserichtung ändern muss.

Da sogenannte Single-Responsibility-Prinzip besagt, dass Klassen **genau eine Verantwortlichkeit** besitzen sollten, um sie so klein wie möglich und so groß wie nötig zu gestalten. Diese Verantwortlichkeit sollte sich in ihrem Namen widerspiegeln. Auf diese Weise erhält der Entwickler eine schnelle Orientierung, wofür die Klasse gedacht ist. Folglich sollte ein großes zu entwickelndes System über **viele kleinere Klassen** verfügen und nicht über vereinzelte große Klassen, wo jede mehrere Verantwortlichkeiten in sich vereint.

Wie es bei den Klassen mit den Verantwortlichkeiten gehandhabt wird, sollte es auch bei den **Methoden** gelten, dass sie nach Möglichkeit eine **klar abgegrenzten Aufgabe** erfüllt. Auf diese Weise wird die Größe einer Funktion automatisch begrenzt. Auch sollte eine Methode eine möglichst geringe Anzahl an Verschachtelungsebenen aufweisen, da diese die Leserlichkeit einer Methode stark beeinträchtigen. Durch **Auslagern von Blöcken**, die eine eigene Aufgabe erfüllen, kann die Verschachtelungstiefe einer Methode reduziert werden.

Bezüglich der **Anzahl an übergebenen Parameter einer Methode** ist zu sagen, dass diese **so gering wie möglich** gehalten werden sollte. Viele Übergabeparameter zwingt den Leser dazu diese bereits zu Beginn der Methode alle interpretieren zu müssen. Die Erreichung einer ausreichenden Testabdeckung der Funktionalität von Methoden erschwert sich zu der Anzahl an existierenden Parameter. Je mehr Parameter übergeben werden, desto schwerer wird es alle möglichen Kombinationen an übergebenen Werten zu testen. Existieren dagegen keine Übergabeparameter, ist dieses Problem bedeutungslos.

Jedoch kann die Anzahl an benötigter Parameter nicht immer gering gehalten werden. Eine Möglichkeit der Parameterreduzierung ist die **Verwendung eines Objekts**, welches die Parameter mit ähnlichem Konzept als Felder beinhaltet. Hier ist jedoch darauf zu achten, dass durch eine ausreichend aussagekräftige Namensgebung der Klasse des Objekts ersichtlich wird, auf welche Parameter zugegriffen werden können. Eine weitere Möglichkeit wäre die Parameter mit ähnlichem Konzept **innerhalb einer Liste** zu übergeben. Ein Beispiel wäre die Übergabe einer Liste, die Punktkoordinaten umfasst, anstelle die Punkte einzeln zu übergeben.

Auch das **Fehler-Handling** sollte bei der Lesbarkeit des Codes mitberücksichtigt werden. Die häufigste Maßnahme des Fehler-Handlings ist die Nutzung von Ausnahmen mithilfe von **Try-Catch-Finally-Anweisungen**. Frühere Programmiersprachen hatten dieses Feature zur Ausnahmebehandlungen nicht, weswegen sich die Fehlerbehandlung auf die Rückgabe von Fehlercode beschränkte, die der Nutzer oder Entwickler interpretieren musste. Die Nutzung von Try-Catch-Finally-Anweisungen bieten die Möglichkeit das Programm zu einem konsistenten Ablauf zu bringen und dem Leser diesen Ablauf im Code zu zeigen.

Kommt es zu einem Fehler, sollte die Rückmeldung durch die Try-Catch-Finally-Anweisung **ausreichend Informationen** liefern, sodass der Nutzer oder Entwickler die Art und den Ort des Fehlers identifizieren kann. Oftmals werden bei auftretenden Fehlern nicht die gewünschten Daten einer Methode zurückgegeben, sondern lediglich der Wert NULL. Dies ist jedoch ein Beispiel für schlechte Fehlerbehandlung, da keine Informationen geliefert werden und im schlimmsten Fall, aufgrund der Rückgabe, der Fehler in eine andere Methode verlagert werden könnte.

Neben dem eigentlichen Quellcode gilt auch für den Code von **Unit-Tests**, dass sie eine möglichst textbfgute Lesbarkeit aufweisen sollten. Die vorherrschende Annahme, dass schlecht lesbarer Testcode besser ist als gar keine Tests zu haben, ist durchaus kritisch zu sehen. Ist es erforderlich, dass der Quellcode angepasst wird, muss u.U. auch der Code des oder der zugehörigen Unit-Tests angepasst werden. Folglich könnte die durch gut lesbaren Quellcode gewonnene Zeit bei der Änderung der Unit-Tests wieder verloren gehen. Wird bei der Anpassung von Unit-Tests zu viel Zeit benötigt, können Entwicklungsteams Gefahr sie als mehr als Belastung denn als Nutzen zu sehen.

Sollte es zu dem Szenario kommen, dass die Tests einen Belastungsgrad erhalten, wo sie nicht weiter verfolgt (können) oder gar entfernt werden, verliert auch der Quellcode seine

Wiederverwendbarkeit. Grund ist, dass durch fehlende Tests der Quellcode nicht mehr auf seine Funktionalität hin geprüft wird und so Änderungen zu unkontrolliertem Verhalten oder zu Fehlern führen kann. Auf diese Weise verliert selbst der bestlesbare Quellcode seine Flexibilität und folglich seine Wiederverwendbarkeit.

Um die Tests mit einer guten Lesbarkeit zu gestalten, ist es zu vermeiden, dass ein einzelner Test mehrere Aspekte abdeckt. Wie bei den anderen Aspekte sollte auch hier das Prinzip der Single-Responsibility verfolgt werden, sodass **größere Tests in mehrere voneinander unabhängige Tests aufgeteilt** werden. Das Prüfen von mehreren Aspekten in einem Test hat obendrein noch die Schwäche, dass bei einem auftretenden Fehler nicht sofort ersichtlich ist, welcher Aspekt den Fehler verursacht hat.

Neben einigen sinnvollen **allgemeingültigen Formatierungsregeln** ist in erster Linie von großer Wichtigkeit, dass ein Entwicklungsteam sich auf einen **gemeinsamen Formatierungsstil einigt und diesen konsequent anwendet**. Auf diese Weise wird der Code für das Entwicklungsteam deutlich lesbarer. Wird ein Tool bei der Erstellung von Programmcode verwendet, welches automatisiert die Formatierungsregeln umsetzt, ist dies obendrein eine große Hilfe bei der Entwicklung.

Die zunächst wahrscheinlich offensichtlichste Tatsache sagt aus, dass eine gute Lesbarkeit leichter bei **kleineren Quellcodedateien** realisieren lässt als bei größeren. Aber unabhängig von der Größe sollte sich eine Quelldatei nach Möglichkeit wie ein Buch oder eine Zeitung lesen lassen. Dieser Umstand wurde bereits in Abschnitt über die Klassen angesprochen. Folglich liest sich eine Quelldatei am besten, wenn die **Leserichtung von oben nach unten** verläuft und die **Detailtiefe nach unten hin zunimmt**. Auf diese Weise kann der Leser direkt zu Beginn die allgemeinen Konzepte der Quelldatei erfassen und bei Bedarf die spezifischeren im weiteren Verlauf betrachten.

Einige weitere allgemeingültige Formatierungsregeln werden im Folgenden stichpunktartig beschrieben:

Felder. Wie bereits in Abschnitt über die Klassen angesprochen, sollten die Felder direkt zu Beginn instantiiert werden und nicht über die Quellcodedatei verteilt sein.

Kontrollvariablen. Auch wenn es möglich ist Kontrollvariablen von Schleifen auch außerhalb der Schleife zu deklariert, sollten dies vermieden werden und diese innerhalb der Schleifenanweisung geschehen. Dies vergrößert den Code nicht unnötig.

Methoden. Wie ebenfalls in Abschnitt über die Klassen angesprochen, sollten Methoden, wovon eine die andere aufruft direkt untereinander stehen.

Einrückung. Unabhängig davon, dass die meisten Entwicklungstools die Einrückung bei Codeblöcken von Methoden, Schleifen etc. automatisch durchführen, ist deren Wichtigkeit zu betonen. Anhand dieser Einrückungen sind die Blöcke leicht zu identifizieren und geben dem Leser einen schnellen Überblick.

Leerzeilen. Eine oder mehrere zusammengehörige Codezeilen, die einen bestimmten gewissen Zweck erfüllen, sollten oberhalb und unterhalb durch jeweils eine Leerzeile von den anderen Codezeilen getrennt werden. Dies heißt im Umkehrschluss, dass zusammengehörige Codezeilen nah beieinanerstehen sollten.

Leerzeichen. Aufgaben innerhalb einer Zeile, welche eine enge Bindung haben, sollten möglichst nah beieinander stehen, während Aufgaben mit geringerer Bindung mithilfe von Leerzeichen voneinander getrennt werden sollten. Einfache Beispiele wären das Setzen von Leerzeichen um den Zuweisungsoperator herum oder die Betonung einer mathematischen Operationsreihenfolge, anhand von Punkt-vor-Strich.

Die **Namensgebung** sollte das Ziel verfolgen, dass der Leser nicht erst durch das Lesen des Inhalts einer Klasse, einer Methode oder des Einsatzes einer Variable versteht, welchen Sinn sie verfolgt, sondern bereits beim Lesen des Namens. Ein Name hat den Zweck einer Klasse, Methode oder Variablen treffend zu beschreiben. Auf diese Weise bedarf es für den Leser keiner langen Einarbeitung in den Code.

Einige weitere allgemeingültige Regeln lassen sich innerhalb der Literatur finden:

Substantive für Klassen. Die Namen von Klassen und folglich deren Objekte sollten aus einem Substantiv bestehen.

Verben für Methoden. Die Namen von Methoden sollten aus Ausdrücken bestehen, die ein Verb beinhalten.

Angemessene Kontextmenge liefern. Worte für Namen, die ohne weiteren Kontext mehrere Bedeutungen haben können, sogenannte Homonyme, sollten erweitert werden, um den Zweck eindeutig beschreiben zu können. Im Gegenzug sollte jedoch auch kein überflüssiger Kontext verwendet werden, um die Länge des Namens nicht ohne weiteren Nutzen zu verlängern.

Auffindbare Namen wählen. Besteht z.B. ein Name im Extremfall lediglich aus einem einzigen Buchstaben, so ist dieser Name in einer entsprechend großen Code-Datei schwer wiederzufinden. Auch würde ein zu kurzer Name den Zweck einer Variablen, Klasse oder Methode mit hoher Wahrscheinlichkeit nicht ausreichend beschreiben.

Unterschiede hervorheben. Erfüllen beispielsweise Variablen unterschiedliche Zwecke, so sollten sich ihre Namen nicht nur leicht, sondern merklich unterscheiden, um Verwechslungen zu vermeiden.

Aussprechbarer Name definieren. Ist ein Name aussprechbar, so ist er für einen Menschen leichter zu merken.

Kommentare vermeiden. Ist es erforderlich einen Kommentar zur Beschreibung einer Klasse, Methode oder Variablen zu verfassen, ist der Name nicht aussagekräftig genug gewählt.

Kommentare sind ausschließlich dort zu platzieren, wo der Code, trotz Einhaltung der zuvor genannten Regeln, nicht ausreichend Auskunft gibt, um sein Verhalten zu verstehen. Jedoch sollte der Nutzen eines Kommentares stets hinterfragt und geprüft werden, ob sämtliche Maßnahmen zur Verbesserung der Lesbarkeit ausreichend umgesetzt wurden. Denn die in der Literatur sehr häufig vertretene Ansicht besagt, dass der beste Kommentar der ist, welche nicht geschrieben werden musste.

7 Fazit

Die Idee Software wiederzuverwenden ist fast so alt wie die Softwareentwicklung selbst, jedoch hat sich die Art, wie Software wiederverwendet wird im Laufe der Jahre stark gewandelt. Gerade die Erfindung des Internets, Online-Repositorys, Foren und weiterer Werkzeuge hat die Art, wie Software Reuse durchgeführt werden kann, maßgeblich verändert. Außerdem stehen Entwicklern heute viele Werkzeuge zur Verfügung, mit welchen sie Software zur Wiederverwendung auffinden können.

Zu den Vorteilen von Software Reuse gehören schnellere und kostengünstigere Entwicklung. Wer von den Vorteilen profitieren möchte muss zunächst verstehen, dass Software Reuse auf verschiedenste Arten durchgeführt werden kann, von der Quellcode-Wiederverwendung bis hin zu Design-Mustern. Je nachdem, was wiederverwendet werden soll, kann der Aufwand für die Wiederverwendung variieren. Um Software erfolgreich wiederzuverwenden können Risiken anhand des Reuse Failure Modes Model frühzeitig erkannt und vermieden werden. Mittels Metriken für Software Reuse lässt sich bestimmen, ob eine Software für die Wiederverwendung geeignet ist.

Die passende Wahl der Software Architektur und der Festlegung und Einhaltung von Code-Richtlinien bieten die Grundlage für Software Projekte, deren Produkt von Beginn an nicht nur auf die Erfüllung der funktionellen Anforderungen abzielt, sondern auch das Produkt in einen Zustand bringt, welcher die Voraussetzung einer einfachen Wiederverwendbarkeit erfüllt. Die Software Architektur bietet den aktuellen wie zukünftigen Entwicklern eine Orientierungshilfe und verringert bis minimiert die Abhängigkeiten der einzelnen Softwarekomponenten untereinander, sodass sich ein erforderlicher Änderungsaufwand ebenfalls minimieren lässt. Die gute Lesbarkeit des Codes, erreicht durch etablierte Code-Richtlinien, sorgt zusätzlich für eine schnelle Orientierung innerhalb des Codes und dem einfachen Verständnis der Funktionalitäten.

Zusammenfassend ist unbestreitbar, dass das Thema Software Reuse nicht nur einige Vorteile mit sich bringt, sondern wichtiger Bestandteil einer jeden Software Entwicklung sein sollte. Durch konsequente Anwendungen der Metriken und durch Analyse des ist-Zustandes kann der Entwicklungsaufwand reduziert und der Entwicklungserfolg entscheidend erhöht werden.

Literatur

- [68] Software engineering: Report on a conference ... Garmisch, Germany, 7th to 11th October 1968. NATO Scientific Affairs Div, Brussels, 1968.
- [Ap12] Appelrath, H.-J.: IT-Architekturentwicklung im Smart Grid: Perspektiven für eine sichere markt- und standardbasierte Integration erneuerbarer Energien. Springer Gabler, Berlin und Heidelberg, 2012, ISBN: 978-3-642-29208-8.
- [Ba08] Bauer, G.: Architekturen für Web-Anwendungen: Eine praxisbezogene Konstruktions-Systematik. Vieweg + Teubner in GWV Fachverlage GmbH, Wiesbaden, 2008, ISBN: 978-3-8348-0515-7.
- [Ba16] Bauer, V. M.: Analysing and supporting software reuse in practice, Dissertation, München: Universitätsbibliothek der TU München, 2016.
- [Do20] Dowalil, H.: Modulare Softwarearchitektur: Nachhaltiger Entwurf durch Microservices, Modulithen und SOA 2.0. Hanser, München, 2020, ISBN: 978-3-446-46377-6.
- [Gi22] GitHub: GitHub: Let's build from here, 4.11.2022, URL: <https://github.com/>.
- [Go11] Goll, J.: Methoden und Architekturen der Softwaretechnik. Vieweg + Teubner, Wiesbaden, 2011, ISBN: 978-3-8348-1578-1.
- [Go14] Goll, J.: Architektur- und Entwurfsmuster der Softwaretechnik: Mit lauffähigen Beispielen in Java. Springer Vieweg, Wiesbaden, 2014, ISBN: 978-3-658-05531-8.
- [Ka19] Kapil, S.: Clean Python: Elegant Coding in Python. Apress, New York, 2019, ISBN: 978-1-4842-4878-2.
- [Ma09] Martin, R. C.: Clean Code: Refactoring, Patterns, Testen und Techniken für sauberen Code. mitp, Frechen und Hamburg, 2009, ISBN: 978-3-8266-9638-1.
- [ND82] Nygaard, K.; Dahl, O.-J.: The development of the SIMULA languages. In (Wexelblat, R. L., Hrsg.): History of programming languages. ACM monograph series, Academic Press, New York, S. 439–480, 1982, ISBN: 0127450408.
- [Ro21] Roth, S.: Clean C++20: Sustainable Software Development Patterns and Best Practices. Apress, New York, 2021, ISBN: 978-1-4842-5948-1.
- [SH11] Starke, G.; Hruschka, P.: Software-Architektur kompakt: - angemessen und zielorientiert. Spektrum Akademischer Verlag, Heidelberg, 2011, ISBN: 978-3-8274-2093-0.
- [St20] Starke, G.: Effektive Softwarearchitekturen: Ein praktischer Leitfaden. Hanser, München, 2020, ISBN: 978-3-446-46376-9.
- [St22] Stack Overflow: Stack Overflow - Where Developers Learn, Share, & Build Careers, 4.11.2022, URL: <https://stackoverflow.com/>.
- [T422] T4Tutorials.com: Software reuse and software reuse oriented software engineering | T4Tutorials.com, 4.11.2022, URL: <https://t4tutorials.com/software-reuse-and-software-reuse-oriented-software-engineering/>.

- [Tr21] Tremp, H.: Architekturen Verteilter Softwaresysteme: SOA & Microservices - Mehrschichtenarchitekturen - Anwendungsintegration. Springer Fachmedien Wiesbaden und Imprint Springer Vieweg, Wiesbaden, 2021, ISBN: 978-3-658-33178-8.
- [Wa11] Wasserman, A. I.: How the Internet transformed the software industry. Journal of Internet Services and Applications 2/1, S. 11–22, 2011, ISSN: 1867-4828.
- [Wi22] Wikipedia, Hrsg.: Library (computing), 2022, URL: [https://en.wikipedia.org/w/index.php?title=Library_\(computing\)&oldid=1109362116](https://en.wikipedia.org/w/index.php?title=Library_(computing)&oldid=1109362116).
- [Wi96] William B. Frakes and Christopher J. Fox: Quality Improvement Using A Software Reuse Failure Modes Model. IEEE TRANSACTIONS ON SOFTWARE ENGINEERING 22/4, 1996.

Auswirkungen einer serviceorientierten Architektur auf den Entwicklungs- und Auslieferungsprozess

Fabian Klimpel¹, Lukas Epple², Raphael Sack³

Abstract: Das Konzept der serviceorientierten Architektur (SOA) existiert schon seit den 1990er Jahren und wird zunehmend in größeren Softwaresystemen eingesetzt. Während diese Architektur viele Vorteile mit sich bringt, bietet die Umsetzung dieser auch viele Herausforderungen. Ziel dieser Arbeit ist es die Vor- und Nachteile der serviceorientierten Architektur zu dokumentieren und Unterschiede zu anderen konventionellen Software-Architekturen, insbesondere hinsichtlich der Entwicklung und Auslieferung aufzuzeigen. Dabei soll vor allem der Kontext der agilen Softwareentwicklung betrachtet werden. Für einen qualitativen Vergleich wurde eine umfassende Literaturrecherche durchgeführt. Die Recherchen zeigten, dass insbesondere die Qualität von umfangreichen und komplexen Softwaresystemen vom Einsatz der serviceorientierten Architektur und deren Auswirkungen auf den Entwicklungs- und Auslieferungsprozess profitieren kann.

Keywords: SOA; serviceorientierte Architektur; Software-Engineering; Software Architektur

1 Einleitung

Bereits seit den frühen 1990er-Jahren werden Konzepte für die Aufteilung von Anwendungen in einzelne Dienste (engl. *Services*) angewendet, um Verantwortlichkeiten innerhalb einer Anwendung entkoppeln und somit verteilen zu können. Zunehmend wurden diese Konzepte anschließend auch in Unternehmen für die Entwicklung umfangreicher Systeme übernommen. Große Systeme konnten somit in unterschiedliche getrennte Dienste aufgeteilt und auf mehrere Rechner verteilt werden. Dies ermöglichte die Entwicklung von Systemen, welche die Ressourcenkapazität eines einzigen Rechners überstiegen. Die isolierten Services waren zudem überschaubarer und konnten leichter gewartet werden. Ebenfalls konnten diese unabhängig von dem restlichen System weiterentwickelt werden.

Neben den Möglichkeiten bei der Entwicklung entstanden ebenfalls weitere Möglichkeiten für die Auslieferung und insbesondere für Aktualisierung und Skalierung von Systemen.

1.1 Motivation

Dieses Paper ist im Rahmen der Vorlesung *Advanced Software-Engineering* an der Dualen Hochschule Baden-Württemberg Stuttgart, Campus Horb entstanden. Im Rahmen dieser

¹ DHBW Stuttgart Campus Horb, Florianstraße 15, 72160 Horb, Deutschland i20021@hb.dhbw-stuttgart.de

² DHBW Stuttgart Campus Horb, Florianstraße 15, 72160 Horb, Deutschland i20009@hb.dhbw-stuttgart.de

³ DHBW Stuttgart Campus Horb, Florianstraße 15, 72160 Horb, Deutschland i20029@hb.dhbw-stuttgart.de

Arbeit wird untersucht, welche Auswirkungen durch die Verwendung einer serviceorientierten Architektur (SOA) hinsichtlich des Entwicklungs- und Auslieferungsprozesses von Software resultieren.

Dabei wird zunächst der Begriff der SOA geklärt und definiert. Anschließend wird die geschichtliche Entwicklung bezüglich der Verwendung unterschiedlicher Softwarearchitekturen genauer untersucht. Zusätzlich wird beleuchtet, wie eine SOA realisiert werden kann. Im Anschluss daran werden die Auswirkungen der Architektur auf den Entwicklungs- und Auslieferungsprozess dargestellt.

1.2 Einführung und Definition

Für den Begriff der SOA existiert keine einheitliche, allgemein anerkannte Definition. Es handelt sich hierbei um keine konkrete Technologie, sondern um ein Konzept, welches über viele Jahre gewachsen ist und weiterentwickelt wurde. Da SOA auf unterschiedliche Bereiche angewendet werden kann, existieren viele unterschiedliche Definitionen, welche jeweils verschiedene Aspekte des Konzepts beleuchten.

Im Folgenden soll nun zunächst der Begriff des Services eingeführt werden. Anschließend sollen unterschiedliche Definitionen von SOA untersucht werden. Schließlich soll eine eigene Definition formuliert werden, die im Rahmen dieser Arbeit verwendet wird.

Ein Service ist ein eigenständiges Softwaremodul, welches zumeist die Funktionalität oder den Ablauf eines konkreten Geschäftsprozesses oder Vorgangs abbildet [Ra05]. Dabei kapselt ein Service meist mehrere zusammengehörige, feingranulare Komponenten und stellt eine präzise Schnittstelle für den Zugriff auf die Funktionalität bereit. Wichtig ist hierbei, dass der Service eine geschlossene Komponente darstellt, welche keine weiteren Abhängigkeiten nach außen besitzt [SHM08]. Dies gilt sowohl gegenüber verwendeten Bibliotheken innerhalb des Services als auch gegenüber anderen Services in einem System. Darüber hinaus sollte ein Service eine positionsunabhängige Adressierungsmöglichkeit anbieten, über welche die Schnittstelle nach außen nutzbar gemacht wird. Üblicherweise werden hierfür Netzwerkprotokolle verwendet. Die positionsunabhängige Kommunikation über standardisierte Netzwerkprotokolle ermöglicht die lose Kopplung mit anderen Services oder Konsumenten [ADM06]. Diese können also über die Schnittstelle die Funktionalität des Services nutzen, ohne dessen Position oder den verwendeten Technologiestack kennen zu müssen. Somit kann aus mehreren lose gekoppelten Services ein modulares Gesamtsystem zusammengesetzt werden, welches sehr einfach verteilt und skaliert werden kann [SHM08].

Eine Architektur für solche Systeme, welche auf einzelnen isolierten und verteilbaren Services basieren, wird *serviceorientierte Architektur* genannt [SHM08]. Dabei existieren in der Literatur unterschiedliche Ansichten, wie genau diese definiert wird. Es existieren viele verschiedene Definitionen aus unterschiedlichen Perspektiven. Diese beleuchten unterschiedliche Aspekte und sind somit zwar korrekt, aber selten einheitlich oder vollständig.

In einem Buch von Thomas Erl [Er09a] wird SOA als offene, agile und zusammensetzbare

Architektur, bestehend aus autonomen Web-Services beschrieben. Die Services sollen dabei eigenständig, wiederverwendbar und herstellerübergreifend interoperabel sein.

In einem Artikel aus dem *CrossTalk* Magazin der US Air Force [G 07] hingegen wird SOA nicht als Architektur, sondern aus architektonisches Muster beschrieben. Aus diesem Architekturmuster können laut den Autoren unbegrenzt viele konkrete Architekturen abgeleitet werden. In einem weiteren Buch von Erl [Er09b] wird SOA als Erweiterung der Objektorientierung beschrieben. Es werden zentrale Konzepte der Objektorientierung, wie beispielsweise Abstraktion, Kapselung und Wiederverwendbarkeit angewendet und durch stärkere Kapselung und unabhängige Kommunikation über Netzwerkprotokolle erweitert. Aus Software, bestehend aus interagierenden Objekten, wird Software aus interagierenden Services. Diese können außerdem mit heterogenen Technologien umgesetzt werden und über Systemgrenzen hinweg interagieren. Arnon Rotem-Gal-Oz beschreibt SOA in seinem Buch [Ro12] als Architekturstil für die Erstellung interaktiver Systeme, bestehend aus lose gekoppelten, grob-granularen und autonomen Services. Jeder Service definiert bestimmte Funktionalität beziehungsweise ein bestimmtes Verhalten und spezifiziert den Zugriff darauf über eine Schnittstelle, den sogenannten *Kontrakt*. Dieser Kontrakt wird über einen adressierbaren Endpunkt nach außen freigegeben. Er beinhaltet Nachrichten zur Benutzung der Funktionalität/des Verhaltens von externen Konsumenten.

Die Granularität der Services ist dabei mit Bedacht zu wählen. Sind die Services zu feingranular, wird das System extrem komplex und es muss mit einem riesigem Overhead umgegangen werden. Wählt man eine zu grobe Granularität, handelt es sich bei dem System mehr um ein Konglomerat mehrerer Monolithen [St21].

Folgende Definition soll die zentralen Punkte einer SOA zusammenfassen und wird im Rahmen dieser Arbeit verwendet.

Definition: SOA stellt ein technologieunabhängiges Architekturmuster für verteilbare Systeme, bestehend aus homogenen oder heterogenen, lose gekoppelten und über Nachrichten interagierenden Services dar. Ein Service stellt dabei eine autonome Softwarekomponente dar, die konkrete Funktionalität oder einen Geschäftsprozess kapselt und über eine klar definierte Schnittstelle positionsunabhängig bereitstellt.

Basis für diese Definition ist der überwiegende Konsens in der Literatur.

Um die Isolation eines Services und gleichzeitig eine klar definierte Interaktion mit diesem zu gewährleisten, werden bestimmte Komponenten benötigt. Diese müssen in einem System, welches die serviceorientierte Architektur realisiert, vorhanden sein. Neben den Services zählen dazu folgende Punkte. [Ro12]

Schnittstelle/Kontrakt

Die Schnittstelle definiert die Menge aller möglichen Nachrichten zur Interaktion mit

dem Service. Der Kontrakt eines Services ist also vergleichbar mit einem Interface in der Objektorientierung.

Endpunkt

Der Endpunkt stellt einen Unique Ressource Identifier (URI) dar, über welchen die Schnittstelle freigegeben wird. Über den Endpunkt kann der Service also gefunden und adressiert werden.

Nachrichten

Über Nachrichten kann mit einem Service interagiert werden. Ein Konsument kann eine Nachricht an einen Service-Endpunkt senden. Erhält der Service eine Nachricht, die einer Spezifikation in dessen Kontrakt entspricht, reagiert der Service auf diese.

Darüber hinaus gibt es noch weitere optionale Bestandteile, wie beispielsweise Policies, die den Zugriff auf Services regeln oder ein Serviceregister, in welchem alle verfügbaren Service-Endpunkte dokumentiert werden.

Die Verwendung einer SOA führt also zu konkreten Eigenschaften eines Systems. So ist beispielsweise die bereits genannte Verteilbarkeit ein zentraler Punkt [Ad10]. Diese resultiert aus der losen Kopplung und positionsunabhängigen Adressierbarkeit. Neben der Flexibilität hinsichtlich der Positionierung wird außerdem eine stärkere Heterogenität der einzelnen Bestandteile eines Systems ermöglicht [SHM08]. Durch die Verwendung von standardisierten und Programmiersprachen-unabhängigen Protokollen für den Nachrichtenaustausch werden Implementierungsdetails vollständig hinter dem Kontrakt verborgen. Diese Möglichkeit zur Heterogenität fördert außerdem die Wiederverwendbarkeit, da einzelne Services in den unterschiedlichsten Systemen völlig unabhängig des verwendeten Technologiestacks genutzt werden können [Ad10]. Die gewonnene Flexibilität steht größerem Overhead und steigender Komplexität gegenüber.

1.3 Anwendungsfälle

Eine serviceorientierte Architektur kann in vielen unterschiedlichen Bereichen eingesetzt werden. Jedoch gibt es Anwendungsfälle wofür eine SOA besser oder schlechter geeignet ist. Gut geeignet ist eine SOA zum Beispiel für komplexe Applikationen, bei denen es viele Komponenten mit klar trennbaren Funktionalitäten gibt. Ebenfalls bietet SOA sich gut für skalierbare Anwendungen, welche auf verschiedenen Umgebungen oder in einer Cloud laufen an. Für kleine Applikationen mit nur wenigen Funktionalitäten bietet sich SOA weniger an, da dabei der Mehraufwand für die Entwicklung und Auslieferung zu groß ist. Statische Applikationen, welche keine Verbindung zu einem Backend benötigen, profitieren ebenfalls nicht von der Verwendung einer SOA.

Spezifischere Anwendungsfälle für SOA sind in der Finanzindustrie für die Integration verschiedener Banking-Systemen, in der Logistik für die Verbindung zwischen Transport-

und Lagersysteme oder in der Telekommunikationsindustrie für die Integration zwischen Netzwerk- und Dienstsystemen. Neben den zuvor genannten Anwendungsfälle gibt es noch viele verschiedene Fälle in denen SOA in der Praxis eingesetzt werden kann. [He07]

2 Geschichtliche Entwicklung

Um die Notwendigkeit der serviceorientierten Architektur nachvollziehen zu können, lohnt es sich die Vergangenheit - vor SOA - anzusehen. In diesem Kapitel wird die Entwicklung von Software-Architekturen vom Monolithen bis hin zur Serviceorientiertheit betrachtet.

2.1 Monolithische Architektur

Als monolithisch werden Objekte bezeichnet, die „aus einem Stück bestehen[.]; zusammenhängend und fugenlos [sind]“ [Du22]. Im Kontext des Software-Engineerings sind damit jegliche Anwendungen gemeint, deren Module nicht unabhängig voneinander ausgeführt werden können und deren Funktionalitäten in einer Applikation gekapselt sind [PMA19].

Während den Anfängen des Software-Engineerings und bevor das Fachgebiet der Software-Architektur existiert hatte, existierten fast ausschließlich Monolithen [KOS06].

Ende der 1960er Jahre gab es die ersten Bemerkungen, Software nicht nur als „amorphous lump of program“ [KOS06][S.24] anzusehen, sondern gezielt die Architektur eines Software-Systems zu designen.

Monolithen haben noch heute ihre Daseinsberechtigung. Gerade kleine stand-alone Anwendungen werden häufig von einem Service dargestellt. Der große Vorteil dabei ist das einfache Testen [PMA19]. Einzelne Dienste einer Anwendung müssen nicht mit Integrationstests auf verschiedenen Status abgebildet werden, da es nur einen Dienst zum Testen gibt. Auch das Deployment ist simpel, da es nur aus einer ausführbaren Komponente besteht.

Durch die „fehlende“ Architektur wird zunächst Zeit gewonnen, da weniger geplant werden muss. Für kleine Projekte oder Prototypen ist das ideal. Gerade bei Letzterem können durch den erstmaligen Aufbau im Monolith die Komplexität und einzelne Komponenten erforscht werden [PMA19]. Aber bei großen Projekten geht schnell der Überblick verloren und die Entwicklung kommt ins Stocken [PMA19].

Die Entwicklung einer monolithischen Anwendung bedeutet eine enge Bindung zur genutzten Technologie [PMA19]. Falls der Support verwendeter Drittanbieter-Software ausläuft, oder neue Schwachstellen gefunden werden, ist es nur schwer möglich Änderungen vorzunehmen. Und gleichzeitig: Egal wie groß oder klein eine Anpassung ist, die gesamte Anwendung muss immer neu getestet, gebaut und verteilt werden. Falls ein Laufzeitfehler auftritt, hat dies den Absturz der gesamten Anwendung zur Folge [PMA19]. Und obwohl solch ein

Bug durch eine kleine Modifikation gefixt werden kann. Manchmal ist ein neuer Build zu aufwändig und es wird auf mehrere Änderungen gewartet. Darunter leidet die Qualität der Software. Auch lässt sich die Applikation nicht beliebig skalieren. Durch zum Beispiel Load Balancer kann die gesamte Anwendung horizontal skalieren, nicht jedoch einzelne Komponenten.

Zusammenfassend: Monolithen sind gut für kleine Projekte, oder erste Proof of Concepts. Falls eine Software jedoch nachhaltig, mehrere Jahre lang zuverlässig betreut und weiterentwickelt werden soll, steigt die Komplexität dieser drastisch mit der Zeit. Neue Teammitglieder müssen sich teilweise in die komplette Codebase einarbeiten, um Änderungen vorzunehmen. Es kann auch dazu kommen, dass keine neuen Arbeitskräfte gefunden werden die sich mit 20 Jahre alter Software-Technologie auseinandersetzen wollen und somit ist eine Neuentwicklung früher oder später unumgehbar.

Wenn Architekturen - und damit der Aufbau von Software - verglichen werden, dann wird dies mithilfe von „fundamentalen Grundprinzipien“ getan [Fr]. Im Folgenden werden diese Punkte aufgezählt um zu zeigen, gegen welche Grundprinzipien die monolithische Architektur verstößt. In den darauf folgenden Kapiteln werden Architekturen gezeigt die iterativ verschiedene Probleme lösen, welche letztendlich einen historischen Verlauf zu SOA aufzeigen soll:

- **Abstraktion:** „Die essenziellen Eigenschaften eines Objekts, die es von allen anderen Arten von Objekten unterscheidet und somit klar definierte konzeptionelle Grenzen in Bezug auf die Perspektive des Betrachters setzt“ [Bo93]. In einem Monolith kann es keine echte Abstraktion geben, da die Software nur aus einem Objekt - sich selbst - besteht. Natürlich existieren auf einem niedrigeren Level eine Abstraktion, aber nicht in der Architektur selbst.
- **Kapselung:** Durch die Verkapselung verschiedenster Abstraktionen können diese gruppiert und auseinander gehalten werden. Dies fördert die Änderbarkeit und Wiederverwendbarkeit [Fr]. Die Kapselung, zum Beispiel von Klassen einer Programmiersprache kann auch für Monolithen existieren, aber nicht in einer Form, die eine Wiederverwendbarkeit anstrebt.
- **Modularisierung:** Hierbei geht es um die sinnvolle Zerlegung eines Softwaresystems und dessen Gruppierung in Subsysteme und Komponenten. Module dienen als physische Container für Funktionalitäten oder Verantwortlichkeiten einer Anwendung. [Fr]. Wie bei vielen dieser Prinzipien lässt die monolithische Architektur Modularisierung zu einem gewissen Grad zu, aber nur auf einem niedrigen Level.
- **Trennung von Verantwortlichkeiten:** Unterschiedliche Verantwortlichkeiten innerhalb eines Softwaresystems sollten voneinander getrennt werden.
- **Kopplung und Kohäsion:** Die Kopplung ist das Maß für die Stärke der Assoziation zwischen Modulen. Eine Starke Kopplung verkompliziert das System [Fr]. Kohäsion

misst den Grad der Konnektivität zwischen den Funktionen und Elementen eines einzelnen Moduls.

- **Suffizienz, Vollständigkeit und Primitivität:** Suffizienz meint, dass eine Komponente alle notwendigen Merkmale einer Abstraktion erfasst und eine sinnvolle und effiziente Interaktion ermöglicht. Vollständigkeit heißt, dass alle relevanten Merkmale erfasst werden. Mit Primitivität ist gemeint, dass jede Operation, die eine Komponente ausführen kann, einfach implementiert werden kann [Fr].
- **Single Point of Reference:** Jede Entität innerhalb eines Softwaresystems sollte nur einmal definiert werden. Dadurch entstehen keine inkonsistenten Zustände.

2.2 Schichtenarchitektur

Die Schichtenarchitektur war eines der ersten Architektur-Pattern mit welchem versucht wurde die Probleme einer monolithischen Architektur zu lösen [SM09] und ist bis heute wahrscheinlich einer der am häufigsten angewendeten Software-Architekturen (vor allem im Web).

Ein Programm wird dabei in n Schichten aufgeteilt, jede Schicht ist ein Modul der Software und kommuniziert über Protokolle mit anderen Schichten. Die Aufteilung alleine löst schon fast alle oben genannten Probleme. Die Schichten bilden meist eine Hierarchie ab, wobei die Kommunikation nur strikt durch gewisse Schichten geschieht.

Vor allem lassen sich Verantwortlichkeiten sehr gut damit trennen. Als Beispiel, kann das 3-Schichten Modell aus typischen Web-Anwendungen betrachtet werden⁴:

1. **Präsentationsschicht:** das Frontend als grafische Benutzeroberfläche im Browser.
2. **Anwendungsschicht:** die eigentliche Funktionalität der Software abgeschottet von Anwendenden.
3. **Datenschicht:** eine Datenbank, auf der die Anwendungsdaten verwaltet werden.

Bei der Modellierung können die Schichten einzeln betrachtet werden, um Schnittstellen zu definieren. Entwicklerteams können daraufhin gleichzeitig an Front- und Backend arbeiten und sind dabei technologisch unabhängig voneinander. Und im Laufe des Lebenszyklus der Anwendung können einzelne Schichten ausgetauscht werden, um neue Technologie einzusetzen. Weitere Vorteile sind auch, dass sich die Geschäftslogik direkt im Code befindet und dass geheime Informationen nicht öffentlich für Benutzer zugänglich sind, sondern sich auf einer unzugänglichen Schicht befinden.

⁴ Auch wenn dies ein prominentes Beispiel ist, besteht eine Schichtenarchitektur nicht immer aus 3 Schichten

Für große Projekte mit größeren Teams ist es leichter Software in Schichten zu schreiben. Einzelne Teams können sich dabei auf eine bestimmte Schicht spezialisieren, um effizienter zu sein.

Die Verteilung von Software bringt auch Nachteile mit sich, bzw. alle Vorteile der monolithischen sind die Nachteile von dieser Architektur:

- Das Testen ist wesentlich aufwändiger. Unit-Tests sind auf den einzelnen Schichten unverändert, aber bei der Integration aller Schichten ist es schwer alle Testfälle abzudecken bzw. die Tests überhaupt zu schreiben.
- Die Installation von Software ist wesentlich aufwändiger, da mehrere Schichten meist über das Internet kommunizieren müssen.
- Der Planungsaufwand ist höher und vor allem kleine Projekte könnten unnötig Zeit an der Trennung der Schichten verlieren. Macht sich aber in der späteren Betreuung des Codes bezahlt.

Zusammengefasst lässt sich sagen: Die Schichtenarchitektur teilt **eine** Software in **verschiedene** Schichten auf. Dabei muss stets auf die Balance zwischen Kopplung und Kohäsion geachtet werden. Die Schichten können dabei verschiedenste Technologien implementieren, solange diese mithilfe wohl definierter Protokolle kommunizieren können. Die Schichten können in der Entwicklung anderer Software wiederverwendet werden, was die Entwicklung in Zukunft erleichtert. Aber die Architektur zeigt ebenfalls Probleme auf. Die Schichten trennen zwar zu einem gewissen Grad die Verantwortlichkeiten der Software, aber für große Projekte ist die statische Hierarchie der Schichten nicht fördernd. Oftmals werden bei einer 3-Schichten Architektur auch nur 2 große Monolithen entwickelt (mit einer Datenschicht) [Fr].

2.3 Serviceorientierte Architektur

Die Herausforderung der Schichtenarchitektur kann gelöst werden, indem die „Schichten“ granularer werden. Und anstatt diese in einer festen Hierarchie anzugeben - wo zum Beispiel Schicht n nur mit Schicht $n + 1$ kommuniziert - gibt es eine liberalere Kommunikation, die nicht mehr statisch vorgegeben sein muss. Anstatt, dass eine Schicht für die gesamte Geschäftslogik einer Anwendung zuständig ist, werden weitere Verantwortlichkeiten innerhalb dieser in Services aufgeteilt. Dadurch wird eine losere Kopplung und eine erhöhte Kohäsion erreicht.

Dieses Ziel hat gerade für große Softwareunternehmen einen immensen Vorteil. Anstatt einzelne Services nur in der Entwicklung wiederzuverwenden, können diese jetzt zur Laufzeit wiederverwendet werden. Die Verantwortlichkeiten sind so granular, dass sich diese in anderen Projekten unverändert wiederverwenden kann.

Langfristig bilden sich nur geringe Kosten durch den hohen Grad der Wiederverwendbarkeit der Dienste, aber bei diesem Grad der Granularität stößt die serviceorientierte Architektur an einen großen Overhead der auch Probleme mit sich bringt [So22]:

- Die Analyse, Konzeption und Implementierung ist initial wesentlich höher als bei anderen Architekturen.
- Die Wiederverwendbarkeit kann auch ein Problem darstellen, da dies nur eingeschränkt und nur auf sehr langer Zeit gesehen möglich ist. Das erfordert eine noch genauere Planung für die Zukunft.
- Die Granularität erhöht die Anzahl der Schnittstellen, deren Änderungen oftmals zu Kompatibilitätsproblemen führen können (=erhöhte Komplexität).
- In der Regel hat eine SOA-Anwendung eine schlechtere Performance und höheres Datenvolumen, da die Dienstkommunikation höhere Latenzen darstellt.
- Die Bereitstellung und Konfiguration der Bindung zwischen den Diensten ist häufig komplex und Themen wie die Authentifizierung/Autorisierung sind Herausforderungen die mit hohen Kosten verbunden sind.
- Entwicklerteams müssen über ein breiteres Wissen verfügen um SOA wirklich anwenden zu können.
- Die Kostenverwaltung der Entwicklung einer zentralen und Abteilungsübergreifenden SOA ist eine politische Herausforderung und muss von Mitarbeitenden unterstützt werden.

Die Information, die aus diesem Kapitel herausstechen sollte, ist die Tatsache, dass sich SOA erst bezahlbar macht, nachdem die gesamte Architektur steht. Bis zu diesem Punkt sind immense Aufwände seitens des Unternehmens notwendig. Bei kleinen bzw. neuen Softwareunternehmen ist der Systemaufbau mit SOA meist zu teuer und würde sich nicht lohnen, bzw. bei einer Gründung ist es schwer in die Zukunft zu blicken, um die richtigen Services herauszusuchen. Für bestehende Unternehmen heißt eine SOA, die Neuentwicklung der Softwarelandschaft. Ein Transformationsprojekt hin zu SOA kann Jahre dauern und ist ein großes für die Zukunft. Die abgeschlossene Transformation macht sich jedoch bezahlt.

Interessant zu sehen: bei diesen Architekturen gibt es einen Trade-off zwischen Komplexität in der Planung und der Komplexität in der Pflege/Entwicklung. Die Gesamtkosten eines Projektes sollten unabhängig von der Wahl einer Architektur sein, jedoch werden die Kosten und Aufwände anders über den Projektlebenszyklus verteilt.

Monolithen haben wenig initiale Aufwände, aber die spätere Pflege ist mit einem hohen Kostenaufwand verbunden. SOA hat immense Anfangskosten, welche nach dem Aufbau der Architektur wieder fallen. Und die Schichtenarchitektur ist eine Balance dazwischen. Monolithische Projekte können ohne viel Aufwand angefangen werden und falls diese ohne Abschluss scheitern, so wurde nur der minimale Aufwand erreicht. Im Gegensatz dazu kostet ein gescheitertes SOA-Projekt wesentlich mehr Geld und Aufwände, da das meiste davon in der Planung (am Anfang) aufgebracht wird.

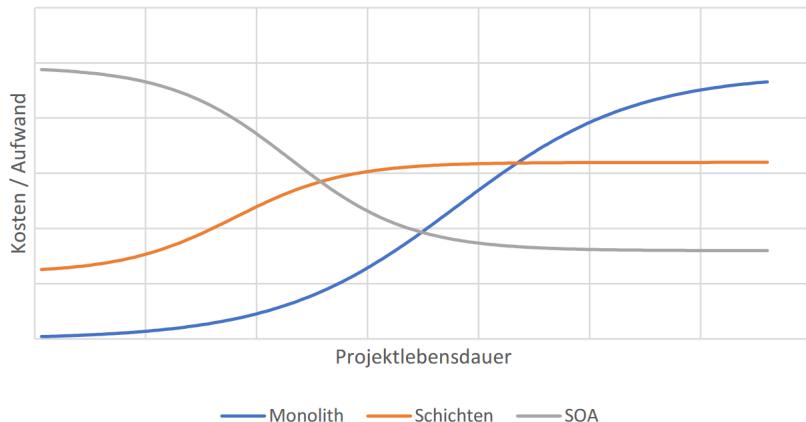


Abb. 1: Relativer Kosten-/Aufwand-Vergleich der vorgestellten Architekturen

Trotzdem kann sich der Umstieg zu einer serviceorientierten Architektur lohnen. Im Nachfolgendem wird tiefer darauf eingegangen wie sich SOA auf den Entwicklungs- und Auslieferungsprozess in der agilen Softwareentwicklung auswirkt.

3 Realisierung einer SOA

Es gibt viele unterschiedliche Wege eine SOA in der Praxis zu realisieren. Eine der verbreitetsten Möglichkeiten eine serviceorientierte Architektur zu realisieren ist mit Web-Services. Weitere Technologien zur Implementierung von SOA ist der Komponentendienst COM+ von Microsoft, Java 2 Platform Enterprise Edition (J2EE) oder die CORBA-Spezifikation. Im Folgenden werden jedoch nur Web-Services betrachtet, da dies der verbreitetste Weg ist eine SOA zu implementieren.

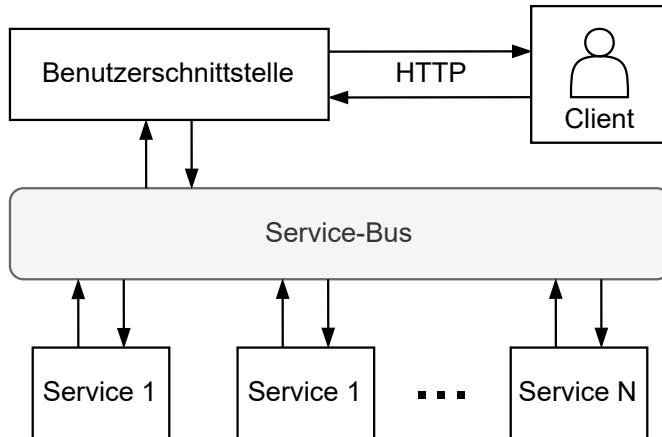


Abb. 2: SOA Aufbau

Wichtig zu sagen ist noch, dass es nicht nur einen richtigen Weg gibt, eine SOA zu implementieren. Es gibt meist viele verschiedene Wege, welche zum gewünschten Ziel führen können. In der Regel müssen mehrere Services implementiert werden, welche gemeinsam über einen Service-Bus miteinander kommunizieren können. Je nach expliziter Implementierung ist der Service-Bus dabei eine globale Instanz über die gesamte Applikation hinweg, oder eine Abstraktion für direkte Verbindungen zwischen verschiedenen Services. In Abbildung 2 wird ein beispielhafter Aufbau einer serviceorientierten Architektur dargestellt. Der Service-Bus ist dabei eine Abstraktion der Schnittstellen und an ihn angebunden sind verschiedene Service-Komponenten. Einer der Services ist dabei die Benutzerschnittstelle, mit der die Benutzer zum Beispiel über ein Frontend interagieren können.

Service-Bus

Mit der wichtigste Bestandteil eines Services ist seine Schnittstelle. Für die Realisierung der Schnittstelle kommen zwei verschiedene Designprinzipien infrage. Entweder ein nachrichtenorientiertes Design oder ein hypermedia-gesteuertes Design.

Bei dem nachrichtenorientierten Design kommunizieren die Services über einen Service-Bus. Über diesen können die einzelnen Komponenten Nachrichten austauschen. In der Regel wird dabei ein Message-Broker oder eine direkte Kommunikation der Services über TCP/IP verwendet. Für die Kommunikation zum Konsumenten wird in der Regel eine HTTP-API bereitgestellt.

Die zweite Möglichkeit ist die hypermedia-gesteuerte Implementierung. Im Gegensatz zur nachrichtenorientierten Implementierung werden bei der hypermedia-gesteuerten Implementierung nicht nur Daten, sondern auch Inhalte mit Beschreibungen möglicher Aktionen ausgetauscht. Dabei wird zum Beispiel direkt HTML Quelltext mit einer Form zurückgegeben. Dies ist vorteilhaft, wenn der Konsument über eine Webseite auf einen Service zugreifen will. [Na16]

Für die eigentliche Implementierung von dem Service-Bus gibt es viele Möglichkeiten. Ein Service-Bus kann beispielsweise als separate Software-Komponente implementiert werden. Der Service-Bus kann dabei von Grund auf implementiert werden oder es kann ein bestehender Message-Broker wie zum Beispiel RabbitMQ verwendet werden. Dabei kann der Service-Bus Nachrichten empfangen und über Routing an den richtigen Service weiterleiten. Der Vorteil daran ist, dass der Service-Bus dabei beliebig skaliert werden kann. Eine weitere Möglichkeit wäre eine in die Services integrierte Middleware, welche die benötigten Schnittstellen zur Kommunikation unter den Services zur Verfügung stellt. Dabei werden üblicherweise Web-Protokolle wie SOAP oder REST eingesetzt. [He07]

Service-Komponente

Eine Service-Komponente ist eine Software-Entität, welche als eigenständige, unabhängige und in sich geschlossene Einheit mit einem klaren Zweck besteht. Die Komponente ist dabei eine unabhängige Einheit von einer Funktionalität mit standardisierten Schnittstellen zur Kommunikation mit anderen Komponenten in einer Anwendung. In der Regel hat die Service-Komponente eine eindeutige Funktion, welche unabhängig von anderen Service-Komponenten der Anwendung ist.

Die eigentliche Implementierung des Services kann in einer beliebigen Programmiersprache geschehen. Es können auch unterschiedliche Services in unterschiedlichen Programmiersprachen implementiert werden. Dabei kann individuell für die Funktionalität eines jeweiligen Services eine optimale Programmiersprache verwendet werden. Wichtig ist dabei lediglich, dass die Schnittstellen korrekt angesprochen werden.

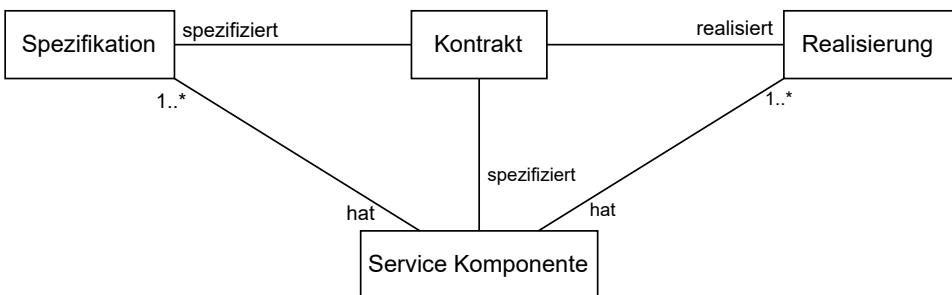


Abb. 3: Spezifikation und Realisation einer Service-Komponente [SDS04]

Die Basiselemente für die Planung und Umsetzung einer Service-Komponente sind die Spezifikation, der Kontrakt und die Realisierung. Die Zusammenhänge der Basiselemente sind in Abbildung 3 dargestellt. Im Kontrakt sind dabei die Regeln und Anforderungen der Service-Komponente spezifiziert. Also der Funktionsumfang, welcher der Service bereitstellen soll. Dieser Kontrakt wird in der Spezifikation weiter spezifiziert. In der Spezifikation wird die Art und die Verwendung der Schnittstellen definiert. Darunter zählen zum Beispiel die verwendeten Protokolle und Standards für die Kommunikation

und die Information welche Funktionen von dem Service bereitgestellt werden. Wurde alles korrekt spezifiziert kann der Service-Kontrakt implementiert werden. Für die Benutzung des Services ist dessen Implementierung nicht von Bedeutung, solange der Kontrakt des Services voll erfüllt wurde. [SDS04]

3.1 Konzepte und Technologien

In der serviceorientierten Architektur gibt es keine Vorschriften wie genau etwas zu implementieren ist und welche Technologie dabei verwendet werden soll. Jedoch gibt es einige Technologien und Konzepte, welche sich in dem Bereich der SOA etabliert haben. Die Wahl der explizit zu verwendeten Technologien ist dabei von den Anforderungen der Anwendung abhängig. Bei der Planung und dem Design sollten die Technologien jedoch genau spezifiziert werden, welche anschließend zur Implementierung verwendet werden.

Die verbreitetste Technologie zur Umsetzung von SOA sind Web-Services. Web-Services kommunizieren über standardisierte Schnittstellen in einem Netzwerk miteinander. Dabei funktioniert die Kommunikation über Netzwerkprotokolle wie zum Beispiel SOAP oder REST. [He07]

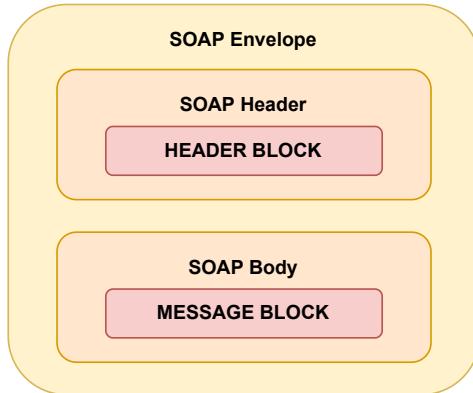


Abb. 4: SOAP Aufbau [Gi22]

Das Simple Object Access Protocol (SOAP) benutzt in der Regel das HTTP-Protokoll zur Datenübertragung. Dabei werden die Daten in Form von XML gesendet. SOAP ist unabhängig von der verwendeten Programmiersprache und wird meist in verteilten Systemen und somit auch in SOA verwendet. Der Aufbau einer SOAP-Komponente ist in Abbildung 4 dargestellt. Außen befindet sich der SOAP Envelope, welcher alle Daten enthält und das XML-Dokument als SOAP-Nachricht identifiziert. In dem SOAP Envelope befindet sich ein Header und ein Body. Der Header beinhaltet dabei zusätzliche Informationen über die Nachricht. Dies sind zum Beispiel zusätzliche Daten zur Authentifizierung. Im Body steht

der eigentliche Inhalt der Nachricht. SOAP kann sowohl für Anfragen an einen Service, als auch für dessen Antwort verwendet werden. [Gi22]

Der Aufbau in Abbildung 4 wird wie folgt im XML-Format geschrieben:

```
<?xml version="1.0"?>
<soap:Envelope xmlns:soap="http://www.w3.org/2003/05/soap-envelope/">
    <soap:Header>
        HEADER BLOCK
    </soap:Header>
    <soap:Body>
        MESSAGE BLOCK
    </soap:Body>
</soap:Envelope>
```

Um einen Service und vor allem dessen Schnittstellen zu beschreiben kann die Web Services Description Language (WSDL) verwendet werden.

WSDL ist ein Standardformat um einen Web-Service zu beschreiben. Die Beschreibung des Services wird in dem XML-Format in der WSDL-Datei definiert. Neben einer Beschreibung der Funktionalität des Services sind auch alle Informationen, um mit einem dem Web-Service zu kommunizieren, darin enthalten. [He07]

Häufig wird WSDL in Verbindung mit SOAP verwendet. Dabei kann der Client, welcher den Service anfragt, die WSDL Datei einlesen, um somit die verfügbaren Funktionen des Services abzurufen. Nun kann er die in WSDL definierten verfügbaren Funktionen über das SOAP-Protokoll verwenden. [He07]

Für die Umgebung, in der die Services ausgeführt werden, bietet sich eine skalierbare Cloud-Infrastruktur an, bei der eine schnelle Bereitstellung an Rechenkapazitäten und ein automatisches Deployment der Services realisiert werden kann.

3.2 Sicherheitskritische Aspekte der SOA

Eine serviceorientierte Architektur ist ohne IT-Sicherheit nicht denkbar. Neben klassischen, bei jeder IT-Lösung, auftretenden Aufgaben gibt es auch - durch den verteilten Ansatz von SOA - spezielle Anforderungen an das System [ZM09].

Die einzelnen Aufgaben müssen dabei nicht neu erfunden werden, sondern haben lediglich besondere Ausprägungen in der Anwendung von SOA. In allen Fällen muss sich mit folgenden Problemen befasst werden [ZM09]:

- Identitätsverwaltung
- Authentisierung

- Autorisierung (=Access Control)
- und die verschlüsselte Kommunikation zwischen den Diensten.

Nachfolgend wird beschrieben, wieso diese Aufgaben gerade in SOA kritische Herausforderungen sind und wie sie gelöst werden.

Access Control: In einer SOA müssen nicht nur viele Benutzer, sondern auch Dienste authentifiziert werden. Die Verwaltung dieser Identitäten, insbesondere über Organisationsgrenzen hinweg ist eine Notwendigkeit [ZM09].

Bei SOA existieren - anders als bei monolithischen Architekturen - viele potenzielle Schwachstellen an jedem Service. Wird eine davon ausgenutzt, so ist zwar nur ein kleiner Teil der Anwendung kompromittiert, aber die Sicherheit des Gesamten ist nicht mehr vorhanden.

Jeder einzelne Service sollte ausreichend mit geschützt sein, damit nur autorisierte Identitäten Zugriff auf eine Ressource haben. Gleichzeitig sollten Benutzer sich nicht bei jedem Service einzeln anmelden, da es nicht benutzerfreundlich ist.

Dies kann erreicht werden, indem bestehende Sicherheits-Architekturen oder Standards in SOA als weitere Services integriert werden. Der Hauptzweck hinter all diesen Standards ist die Orchestrierung der Authentifizierung zwischen mehreren Diensten.

Die dabei am häufigsten verwendeten Standards sind:

- **XACML:** Dieser Standard besteht aus mehreren Services (Points), welche anhand von Policies Anfragen einer Entität bewerten und gegebenenfalls weiterleiten. „XACML bleibt die einzige standardisierte Methode zur dynamischen Durchsetzung von Autorisierungen, indem Zugriffskontrollen von Anwendungen und Datenbanken ausgelagert und Geschäftsrichtlinien verwendet werden.“ [Ax22]

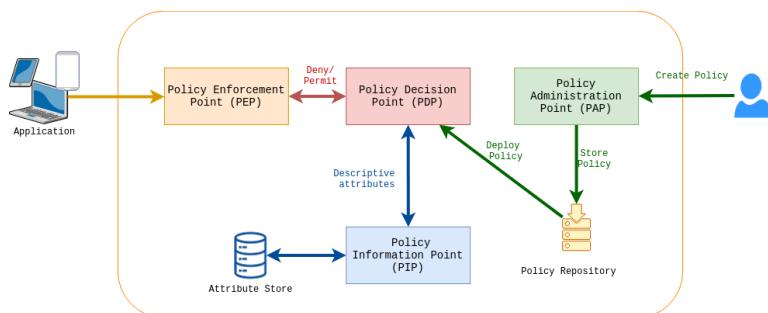


Abb. 5: XACML Flow-Diagramm

- **SAML2:** Hierbei exisitiert ein Identity Provider, an den jeder Service „weiterleitet“, um zentral an einer Stelle die Autorisierung zu überprüfen. [Wi06]

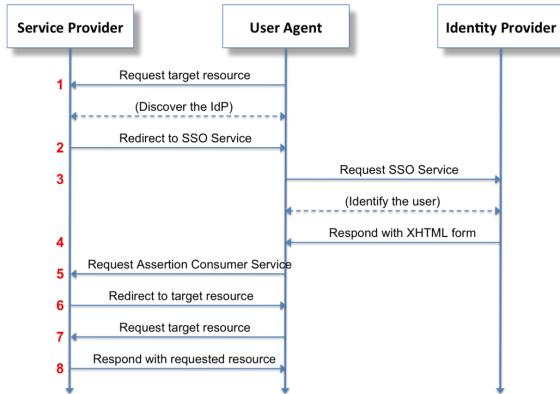


Abb. 6: SAML Sequenz-Diagramm

Die detaillierte Beschreibung dieser Standards würde den Rahmen dieser Arbeit übersteigen. Es existieren jedoch viele Ressourcen, um sich diese näher anzusehen.

Verschlüsselung: Neben den Diensten selbst, stellen auch die Kommunikationskanäle dazwischen einen Angriffsvektor dar, da diese meist Netzwerkprotokolle verwenden. Um die Integrität, Vertraulichkeit und die Authentizität der Nachrichten zu erfüllen wird der WS-Security (Web-Service Security) Standard verwendet.

WS-Security baut auf SOAP auf und klärt:

- Wie SOAP-Nachrichten signiert werden um die Integrität sicherzustellen.
- Wie SOAP-Nachrichten verschlüsselt werden um Vertraulichkeit zu gewährleisten.
- Wie Sicherheitstokens an SOAP-Nachrichten angehängt werden können um die Identität des Absenders sicherzustellen.

Die WSS-Spezifikation gibt nicht explizit vor, welche Signaturformate, Verschlüsselungsalgorithmen oder Sicherheits-Token Modelle verwendet werden müssen. Somit kann dieser für die Zukunft geändert werden, falls bessere und neuere Modelle existieren.

Das große Problem bei der Kanalverschlüsselung in SOA ist die große Anzahl der Kanäle. Egal welche Sicherheitsmaßnahmen eingesetzt werden, sie werden immer die Performance beeinträchtigen.

WSS ist ca. 8 Mal langsamer als herkömmliches HTTPS [LF04], bietet jedoch sichere Funktionalitäten.

Ein großer Performanceverlust liegt in der Generierung eines neuen Schlüssels für jede geschickte Nachricht. Als Upgrade existiert WSS-Conversation (WSSC), welches gleichzeitige Sessions unterstützt, ohne einen Schlüssel neu definieren zu müssen (verdoppelt den durchschnittlichen Durchsatz) [LF04].

4 Auswirkungen auf den Entwicklungsprozess

Der Entwicklungsprozess beschreibt den Prozess von der Planung über das Design bis zur letztendlichen Implementierung einer Anwendung. Durch die serviceorientierte Architektur gibt es in einigen Bereichen des Entwicklungsprozesses größere oder kleinere Auswirkungen im Gegensatz zu herkömmlichen Architekturen wie zum Beispiel der monolithischen Architektur. Dabei gibt es bei SOA auch andere Schwerpunkte auf welche bei der Entwicklung zu achten sind. Da es nicht nur einen richtigen Weg gibt, eine SOA zu implementieren können die Schwerpunkte je nach spezifischer Implementierung etwas abweichen.

Der Software-Entwicklungsprozess hat sich über die letzten Jahrzehnte stark weiterentwickelt. Agile Softwareentwicklung wurde immer populärer und damit einhergehend auch die Entwicklung von verteilten Applikationen. Der serviceorientierte Ansatz für die Softwarearchitektur hat sich dabei zu einer wichtigen Alternative gegenüber der traditionellen Softwareentwicklung entwickelt. [HR10]

4.1 Feldstudie

Um die Veränderungen von SOA auf den Entwicklungsprozess genauer zu untersuchen, wurden im Jahr 2010 in einer Feldstudie Softwareentwickler und IT-Manager aus fünf verschiedenen Unternehmen zu diesem Thema befragt. Bei den gestellten Fragen ging es explizit um die Auswirkungen auf die Softwareentwicklung von SOA basierenden Web-Service. Eines der Unternehmen ist ein Consulting Unternehmen und die restlichen vier Unternehmen sind direkt in der Softwareentwicklung tätig. In jedem der Unternehmen wurde SOA bereits etabliert, oder es war zu dem Zeitpunkt der Umfrage dabei zu SOA umzustellen. Die einzelnen Aussagen der Interviewteilnehmer wurden analysiert, gegeneinander verglichen und zusammengefasst. Zur Validierung der Ergebnisse haben die Teilnehmer im Anschluss noch einmal über die Ergebnisse geschaut. Die Resultate wurden in fünf Phasen des Software-Entwicklungsprozesses eingeteilt. Darunter zählen die Planungsphase, Analysephase, Designphase, Implementierungsphase und die Testphase. [HR10]

Im Folgenden werden die aus der Feldstudie ermittelten Unterschiede von SOA zu herkömmlichen Architekturen, wie zum Beispiel dem Monolithen, erläutert.

Planungsphase

Die erste Phase ist die Planungsphase. In dieser Phase gibt es größere Veränderungen gegenüber der monolithischen Architektur. Die genaue Umgebung der Applikation ist in einer serviceorientierten Architektur in der Planungsphase noch sehr unbestimmt. Viele Aspekte sind mit SOA deutlich schwerer vorherzusagen und zu kontrollieren, vor allem wenn verschiedene Services von unterschiedlichen Bereichen eines Unternehmens oder sogar von anderen externen Unternehmen stammen. Vor allem in der Planungsphase ist es essenziell, dass effektive Kommunikationskanäle zwischen den verschiedenen Stakeholdern aufgebaut werden. Dies ist zwar auch bei anderen Architekturen nötig, doch für SOA ist es wesentlich wichtiger für einen zukünftigen Projekterfolg. Ein weiterer wichtiger Punkt ist es, Standards festzulegen, welche von den Services eingehalten werden müssen. Darunter fallen zum Beispiel Protokolle zur Kommunikation zwischen verschiedenen Services. An diese Standards muss sich bei der Designphase aller benötigten Services gehalten werden. [HR10]

Analysephase

In der Analysephase fallen die wenigsten Veränderungen im Vergleich zu herkömmlichen Architekturen an. Jedoch ist die Analysephase in SOA ein sehr wichtiger Bestandteil des Entwicklungsprozesses. Ein wichtiger Punkt ist dafür zu sorgen, dass alle verwendeten Datenmodelle und Schemata alle benötigten Daten zur Verfügung haben, da bei SOA im Vergleich zu herkömmlichen Architekturen eine globale Perspektive über alle Services hinweg benötigt wird. [HR10]

Designphase

In der Designphase gibt es wie in der Planungsphase wesentliche Unterschiede im Gegensatz zu herkömmlichen Architekturen. Bei SOA ist es sehr wichtig, dass bei der Designphase von Anfang an alle benötigten Schnittstellen definiert werden. Wenn zu einem späteren Zeitpunkt etwas an den Schnittstellen zwischen den Services geändert werden soll, ist mit einem erheblichen Mehraufwand zu rechnen. Wenn bei herkömmlichen Architekturen mit Schnittstellen gearbeitet wird, ist es ebenfalls wichtig diese schon zu Beginn zu definieren, jedoch sind die Auswirkungen bei Änderungen zu einem späteren Zeitpunkt bei vergleichsweise SOA deutlich verstärkt. Die allgemeine Entwicklung von Schnittstellen verändert sich mit SOA auch stark, da in die Services Standards wie zum Beispiel WSDL eingebunden werden müssen. [HR10]

Implementierungsphase

Auf die Implementierungsphase hat SOA die meisten Auswirkungen. Mit SOA können schnell neue und verbesserte Funktionalitäten implementiert werden, ohne mit anderen Geschäftsprozesse dabei in Konflikte zu geraten. Es ist somit einfacher neue Services bereitzustellen und somit hat SOA eine sehr positive Auswirkung auf die Implementierung. Das Verwalten und Management der einzelnen Services ist nun jedoch ein etwas kritischerer

Punkt, da jeder Service ein potenzieller „single point of failure“ einer Anwendung darstellt. Je nach Aufbau der Anwendung kann man dies zum Beispiel mit redundanten Services beschränken. Bei der Implementierung der einzelnen Services ist in der Implementierungsphase ebenfalls ein hohes Maß an Kommunikation und Koordination zwischen den einzelnen, in den Entwicklungsprozess involvierten Gruppen nötig. Durch unterschiedliche Gruppen besteht jedoch auch die Gefahr, dass Services mehrere unterschiedliche Kanäle zur Kommunikation benötigen. Deswegen ist es wichtig die zuvor spezifizierten Standards für SOA einzuhalten. [HR10]

Testphase

Ebenfalls hat SOA einen großen Einfluss auf die Testphase. Dabei muss vor allem mehr Fokus auf die Integrationstests gelegt werden. Bei den Tests sind zwei verschiedene Umgebungen zu beachten. Eine Umgebung, in welcher die Service-Entwickler möglichst einfach Test-Clients erstellen können, um die Services zu testen und eine weitere Umgebung für Client-Entwickler, welche die entwickelten Applikationen gegen Test-Services testen können. Tools für automatisierte Test-Prozesse spielen dabei ebenfalls eine große Rolle. Nicht nur für funktionale Fehler, sondern auch für die Sicherheit und Performance der Services. Durch die Komplexität der Umgebung sind automatisierte Integration-Tests im Gegensatz zu einer monolithischen Architektur wesentlich wichtiger. [HR10]

Ergebnis der Feldstudie

Die Feldstudie zeigt, dass durch SOA der Entwicklungsprozess teilweise stark angepasst werden muss, um ein effektives Arbeiten zu ermöglichen. Einige Bereiche des Entwicklungsprozesses sind dabei stärker betroffen als andere. Ein größeres Augenmerk muss bei SOA auf die Planungs- und Designphase gelegt werden, um die unterschiedlichen Services inklusive deren Schnittstellen korrekt zu definieren. Die Kommunikation zwischen den verschiedenen Teams hat dabei ebenfalls einen besonders großen Stellenwert. Eine weitere Auffälligkeit ist, dass Änderungen an der anfänglichen Planung mit sehr hohen Kosten bzw. Mehraufwand verbunden sind.

4.2 Agile Softwareentwicklung

Agile Softwareentwicklung hat in der Vergangenheit immer mehr an Popularität gewonnen. Damit einhergehend hat der serviceorientierte Ansatz für die Entwicklung immer mehr an Bedeutung gewonnen. Größere Unternehmen können nun mit vielen kleineren Entwicklungsteams unabhängig voneinander parallel an einem Projekt produktiv arbeiten. Somit ist zum Beispiel jedes Team für einen Service zuständig. Bei herkömmlichen monolithischen Ansätzen nimmt die Produktivität ab einer gewissen Teamgröße nicht mehr zu oder sogar ab, da sich die Teammitglieder dabei behindern oder in die Quere kommen. Ebenfalls können in einer Applikation unterschiedliche Programmiersprachen für die Services benutzt werden,

und somit auf die Anforderungen des Services oder auf die Kompetenzen des jeweiligen Entwicklungsteams angepasst werden.

Ein wichtiger Punkt, auf welchen in dem Entwicklungsprozess zu achten ist, ist die Service-Granularität. Die Service-Granularität beschreibt den Funktionsumfang eines einzelnen Services. Bei einer hohen Granularität gibt es somit viele Services mit jeweils sehr kleinem Funktionsumfang und bei einer geringen Granularität gibt es wenige Services mit einem größeren Funktionsumfang.

In der agilen Softwareentwicklung geht es um die Reduzierung der Größe und des Umfangs der Probleme, der Reduzierung der Zeit für die Implementierung und die Reduzierung der Zeit um Feedback zu enthalten. Dafür bieten sich eine hohe Service-Granularität und somit kleine Services mit sehr beschränkten Funktionalitäten an. Somit können kleine Services mit einem kleinen Funktionsumfang eigenständig entwickelt werden. Wie klein genau ein Service sein sollte, ist jedoch schwer zu pauschalisieren, geschweige denn zu messen. In der Praxis ist jedoch eine höhere Service-Granularität und somit mehrere auf jeweils einen einzelnen Funktionsbereich zugeschnittene Services vorzuziehen. [Na16]

4.3 Modularität und Wartbarkeit

Durch die Services ist ebenso ein höheres Level an Modularität in einer Applikation gegeben. Durch die serviceorientierte Architektur muss nur die Kommunikation unter den Services über die Schnittstellen fest definiert sein. Wie ein Service im inneren aufgebaut ist, ist dabei nebensächlich. Verschiedene Services können somit wiederverwendet werden um redundante Softwareentwicklungen vermeiden. Dabei kann nicht nur der Quelltext wiederverwendet werden, sondern teilweise auch ganze Software-Komponenten. Die entwickelten Service-Komponenten können dabei auch in anderen Anwendungen und Systemen eingesetzt werden, um deren Funktionalität zu erweitern. Durch die Wiederverwendbarkeit der Komponenten kann der Entwicklungsprozess langfristig beschleunigt und die Fehleranfälligkeit reduziert werden. Allerdings gibt es dabei andere Schwerpunkte, worauf geachtet werden muss. Bei größeren Änderungen an Services muss darauf geachtet werden, dass die gesamte Applikation mit allen Services noch funktioniert. Gegebenenfalls müssen dabei noch andere Services angepasst werden. Schwerwiegender wird das Problem, wenn der Service in mehreren Applikationen verwendet wird. Dies ist ein weiterer Grund für die Wichtigkeit der Planungsphase bei einer SOA.

Neben der Modularität bringt auch die Wartbarkeit aus der Sicht des Entwicklungsprozesses langfristig deutliche Vorteile. Durch die Aufteilung in kleine Services können ohne Beachtung von anderen Services, Updates oder Erweiterungen für einen Service implementiert werden. Späteres Refactoring wird dank simplen Services anstelle einer komplexen monolithischen Applikation ebenfalls deutlich vereinfacht. Bei Funktionsupdates können dabei auch weitere Services ohne Probleme implementiert werden. Bei komplexen monolithischen Applikationen ist bei Funktionsupdates mit einem deutlichen Mehraufwand zu rechnen. Dies trifft vor allem zu, wenn einer Applikation über die Zeit immer weitere

Funktionalitäten ohne ein größeres Refactoring hinzugefügt werden oder sich viele Altlasten in dieser befinden. [Na16]

5 Auswirkungen einer SOA auf den Auslieferungsprozess

Auslieferung von Software beschreibt den Prozess der Verteilung, Installation beziehungsweise Aktualisierung und Konfiguration von Software für den produktiven Einsatz. Üblicherweise werden hierbei Softwarepakete mit zugehörigen Nutzungsrechten vom Softwarehersteller an den Softwarebetreiber übertragen.

Dabei existieren völlig unterschiedliche Herangehensweisen für den Auslieferungsprozess. Je nach Art der auszuliefernden Software und dem späteren Einsatz kann entweder manuell oder automatisiert ausgeliefert werden. Dabei existieren unzählige Zwischenstufen mit unterschiedlichen Automatisierungsgraden. Des Weiteren kann Software beispielsweise entweder über ein Netzwerk oder über ein physisches Installationsmedium ausgeliefert werden. Auch die Art und Frequenz der Durchführung von Aktualisierungen kann sich erheblich unterscheiden. Darüber hinaus existieren viele weitere Möglichkeiten und Eigenschaften, in welchen sich der Auslieferungsprozess eines Softwaresystems unterscheiden kann.

Viele der Entscheidungen für die Gestaltung des Auslieferungsprozesses hängen weniger von der Architektur der Software ab als viel mehr von der Art des Systems und von den Wünschen des Kunden. Daher werden viele mögliche Eigenschaften des Auslieferungsprozesses in diesem Abschnitt nicht behandelt. Dennoch ergeben sich aus der Wahl der Softwarearchitektur unterschiedliche Möglichkeiten beziehungsweise Vor- und Nachteile, welche insbesondere bei der Wahl der Verteilungsstrategie ausschlaggebend sind. Einige Verteilungsstrategien werden im nächsten Abschnitt vorgestellt.

Eine wesentliche Eigenschaft von Services ist ihre Eigenständigkeit und Unabhängigkeit. Um diese Eigenschaften auch im Prozess der Auslieferung von serviceorientierten Systemen in größtmöglichem Umfang beibehalten zu können, werden Services üblicherweise in virtuellen Maschinen (VMs) gekapselt und ausgeliefert. VMs sind komplexe Softwaresysteme, die eine isolierte Umgebung auf einem Rechnersystem darstellen. Sie emulieren physische Maschinen und stellen eine Umgebung bereit, welche exakt wie die physische Maschine reagiert, unabhängig von dem tatsächlichen physischen System auf welchem die VM läuft. Der große Vorteil von Virtualisierung ist, dass jegliche Software innerhalb einer VM zunächst vollständig vom außenstehenden System isoliert ist. Zum einen ist ein Service somit von unerwünschtem Zugriff von außen geschützt, da Zugriffsmöglichkeiten zunächst explizit freigegeben werden müssen. Zum anderen ist die Software innerhalb der VM völlig unabhängig von dem physischen System, auf welchem diese läuft, sodass die Software mitsamt der VM auf einen beliebigen anderen virtualisierten Host portiert werden kann. Somit ist ein Service nicht nur positionsunabhängig adressierbar, sondern auch positionsunabhängig ausführbar.

Für Microservices werden außerdem oftmals Container für die Kapselung genutzt. Diese haben den Vorteil, dass sie wesentlich leichtgewichtiger sind und somit in sehr dynamischen

Systemen eine schnellere Skalierung mit weniger Overhead ermöglichen. Jedoch ist die Software innerhalb eines Containers nicht vollständig isoliert, da beispielsweise mehrere Container auf einem Host denselben Kernel nutzen. Daraus resultieren erhebliche Einbußen hinsichtlich der Sicherheit und Isoliertheit eines serviceorientierten Systems, da sich Software, die in separaten Containern jedoch auf demselben Host läuft, gegenseitig über den Host-Kernel korrumpern kann.

In beiden Fällen wird jedoch eine Kapselung der verwendeten Technologie erreicht, sodass für die Auslieferung und Inbetriebnahme lediglich die API der Virtualisierungs- beziehungsweise Containerisierungslösung bekannt sein muss. Es wird kein weiteres Wissen hinsichtlich der Technologie des Services benötigt. Die Vorteile im Auslieferungsprozess werden allerdings mit erhöhter Komplexität beim Bauprozess aufgewogen, da hier zusätzlich die Images für die VMs oder Container erstellt werden müssen.

Die Verwendung einer SOA und die gewonnene Flexibilität durch die daraus resultierende lose Kopplung ermöglicht zusätzlich eine wesentlich agilere Auslieferung von Aktualisierungen und flexiblere Möglichkeiten für die Skalierung eines Softwaresystems.

Im Folgenden werden diese Möglichkeiten genauer erläutert.

5.1 Skalierung

Ein System, bestehend aus autonomen Services kann sowohl horizontal als auch vertikal unproportional skaliert werden. Unproportionale Skalierung meint hierbei, dass einzelne Services des Systems in beliebige Richtung skaliert werden können, während andere Services ihre Kapazität halten. Somit kann sich das Gesamtsystem dynamisch an beliebige Lastszenarien anpassen [Na16].

Weitere oder leistungsfähigere Serviceinstanzen können somit bei Bedarf ausgeliefert und zugeschaltet werden. Man nennt dieses Vorgehen auch *deploy on demand*.

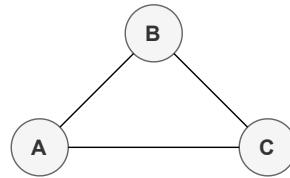
Dabei sollte beachtet werden, dass eine Skalierung lediglich dann erfolgen sollte, wenn mittel- oder langfristige Lastveränderungen im System auftreten oder eine Lastveränderung aufgrund eines vorhersehbaren Ereignisses bevorsteht. Die ständige Anpassung an kurzfristige Szenarien sorgt ansonsten für enormen Overhead aufgrund der Auslieferung und Inbetriebnahme beziehungsweise dem Herunterfahren von Services. Zusätzlich erfolgen ständige Änderungen im Lastenverteilungssystem durch zu- beziehungsweise abschalten neuer Services oder Ressourcen.

Vertikale Skalierung beschreibt in diesem Kontext die Erhöhung beziehungsweise Reduktion der verfügbaren Ressourcen eines Services. Dies kann beispielsweise bedeuten, eine Serviceinstanz durch eine andere zu ersetzen, die auf einem leistungsfähigeren Host läuft oder die zugeteilten Ressourcen der VM, in welcher die Serviceinstanz läuft, zu erhöhen [Na16].

Bei der horizontalen Skalierung werden mehrere Instanzen des gleichen Services eingerichtet und zugeschaltet. Die Gesamtlast des Systems wird dann über ein Lastenverteilungssystem auf die unterschiedlichen Instanzen aufgeteilt.

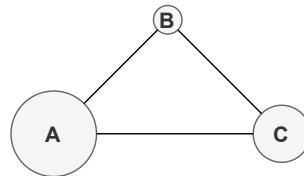
Abbildung 7 stellt die unterschiedlichen Skalierungsmöglichkeiten bildlich dar.

Belastung der einzelnen Services		
A	B	C
mittel	mittel	mittel



Unproportionale vertikale Skalierung

Belastung der einzelnen Services		
A	B	C
hoch	niedrig	mittel



Unproportionale horizontale Skalierung

Belastung der einzelnen Services		
A	B	C
hoch	mittel	mittel

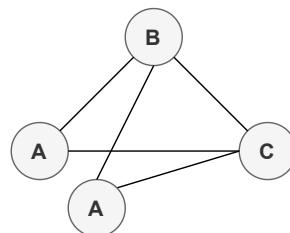


Abb. 7: Darstellung der beispielhaften Skalierungsmöglichkeiten eines Systems bestehend aus drei Services A, B und C, entsprechend bestimmter Lastszenarien

5.2 Aktualisierung

Wie bereits im Zusammenhang mit dem Entwicklungsprozess erläutert, bietet die lose Kopplung viele Möglichkeiten für die agile Weiterentwicklung eines serviceorientierten Systems. Einzelne Services können beispielsweise vollständig unabhängig voneinander weiterentwickelt und aktualisiert werden.

Daraus resultieren zumeist wesentlich frequentiertere Rollouts. Änderungen können auf Serviceebene ausgeliefert werden. Somit kann das Gesamtsystem kleinschrittig weiterentwickelt und aktualisiert werden, ohne dass jemals das komplette System neu aufgesetzt werden muss. Dabei ist zu beachten, dass dies nur möglich ist, wenn die Schnittstellen der einzelnen Services lediglich erweitert, nicht aber verändert oder verkleinert werden. Ansonsten kann es zu Inkompatibilitäten innerhalb des Gesamtsystems kommen.

Im Folgenden werden nun drei Verteilungsstrategien für die Auslieferung von Aktualisierungen eines SOA Systems vorgestellt [St21].

Rollende Auslieferung

Bei der rollenden Auslieferung von Services werden in einer geklusterserten Umgebung nach und nach einzelne Hosts aus der Produktivumgebung genommen. Die neue Version wird auf dem Host eingerichtet. Anschließend wird der Host wieder in die Produktivumgebung eingepflegt. Dies ermöglicht die schrittweise Aktualisierung des gesamten Systems.

Der Vorteil ist, dass nicht direkt alle Instanzen aktualisiert werden, wodurch das Risiko im Fehlerfall begrenzt wird. Außerdem ist die Umsetzung der rollenden Auslieferung relativ einfach. Nachteil an dieser Verteilungsstrategie ist, dass einzelne Hosts zeitweise vom Netz genommen werden.

Abbildung 8 visualisiert den Ablauf einer rollenden Aktualisierung.

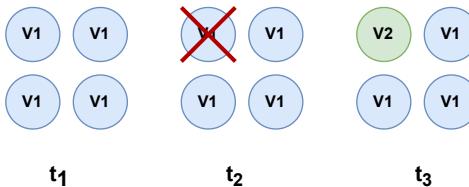


Abb. 8: Ablauf einer rollenden Aktualisierung

Blue-Green Auslieferung

Es wird parallel eine zweite Server-Instanz mit dem aktualisierten Service gestartet. Anschließend werden alle Nachrichten, die an den Service adressiert sind, über ein Lastenverteilungssystem an die aktualisierte Instanz geleitet. Im Falle eines Fehlers wird die Last wieder auf die alte Instanz geleitet. Tritt kein Fehler auf, so kann die alte Instanz heruntergefahren werden.

Vorteil dieser Strategie ist, dass praktisch keine Downtime auftritt. Außerdem kann ein Rollback relativ einfach realisiert werden, indem die Last wieder auf die alte Instanz geleitet und die aktualisierte Instanz wieder heruntergefahren wird.

Der Nachteil dieser Strategie ist, dass im Falle eines Fehlers 100 % der Benutzer betroffen sind, bis der Fehler erkannt und die Last wieder auf die alte Instanz umgeleitet ist.

Abbildung 9 visualisiert den Ablauf einer Blue-Green Aktualisierung.

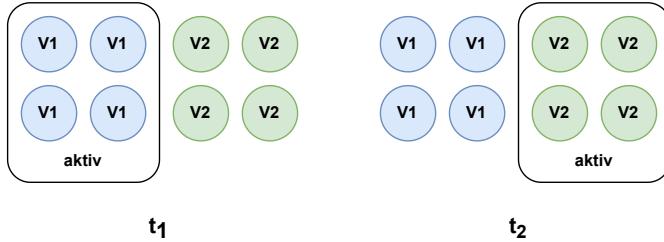


Abb. 9: Ablauf einer Blue-Green Aktualisierung

Canary Auslieferung

Die Canary Strategie ähnelt der Blue-Green Strategie, soll jedoch deren Probleme bei Auftreten eines Fehlers beheben. Sie kann angewendet werden, wenn jeweils mehrere Instanzen der zu aktualisierenden Services im System aktiv sind. Bei der Canary Strategie werden zunächst nur wenige aktualisierte Serviceinstanzen gestartet. Ein Teil der Anfragen werden anschließend über das Lastenverteilungssystem auf die neuen Instanzen geleitet. Treten bei der Benutzung keine Fehler auf, werden anschließend alle zu aktualisierenden Services im Blue-Green Verfahren ersetzt. Die Verwendung der Canary Strategie ist nur möglich, wenn gleichzeitig mehrere Versionen eines Services laufen können.

Vorteil dieser Strategie ist, dass im Fehlerfall nur eine Teilmenge der Benutzer betroffen ist. Außerdem gelten die gleichen Vorteile, die bereits bei der Blue-Green Strategie genannt wurden.

Der Nachteil dieser Strategie ist eindeutig die Komplexität in der Umsetzung. Die Verwendung der Canary Strategie ist nur zu empfehlen, wenn sie über ein Continuous Delivery (CD) System automatisiert ist.

Abbildung 10 visualisiert den Ablauf einer Canary Aktualisierung.

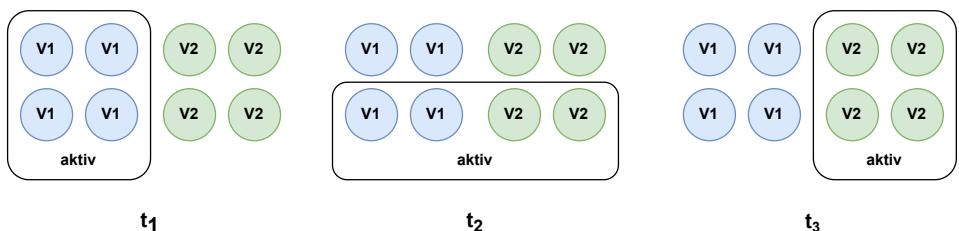


Abb. 10: Ablauf einer Canary Aktualisierung

5.3 Zusammenfassung der Auswirkungen

Es wurden unterschiedliche Aspekte der Auslieferung von serviceorientierten Systemen beleuchtet. Im Folgenden werden nun die konkreten Auswirkungen einer SOA auf den Auslieferungsprozess von Softwaresystemen zusammengefasst.

Zunächst ist die Einrichtung der einzelnen Services, unabhängig der Implementierungs-Technologie über das API der Virtualisierungs- oder Containersoftware ein wichtiger Punkt. Virtualisierung wird zwar auch bei Systemen mit anderen Architekturen verwendet, ist aber zentral für SOA, da sie homogene Auslieferungs- und Installationsprozesse für heterogene Services ermöglicht.

Auch die verteilte Auslieferung der Systeme und die daraus resultierende erhöhte Komplexität des Auslieferungsprozesses ist eine zentrale Auswirkung der Verwendung einer SOA. Alle Services des Systems müssen einzeln ausgeliefert werden und werden zudem üblicherweise auf unterschiedlichen Hosts eingerichtet. Dadurch wird der Auslieferungsprozess wesentlich komplexer als beispielsweise die Auslieferung eines monolithischen Systems. Um diese Komplexität zu verringern, wird zunehmend *serverlos* ausgeliefert. Dabei wird der Code der Services an einen Anbieter für serverlose Auslieferung weitergegeben, welcher dann beliebig viele Instanzen in gewünschter Konfiguration auf eigener Infrastruktur ausliefert. Dabei übernimmt dieser Anbieter die gesamte Verwaltung der Infrastruktur und stellt die Funktionalität der Services über ein Netzwerk bereit. Die Verwendung einer serverlosen Auslieferung hat den Vorteil, dass der Auslieferungsprozess erheblich vereinfacht wird und keine eigenen Hosts mehr verwaltet werden müssen. Im Austausch mit diesen Vorteilen stehen die Vertrauenswürdigkeit der Anbieter und die Tatsache, dass der Herausgeber der Services keinerlei Kontrolle darüber hat, wo diese laufen [St21].

Auch die Auswirkungen auf die Skalierung eines Systems wurden bereits beschrieben. Bei der Auslieferung von monolithischen Systemen wird beispielsweise eine Instanz des Systems auf einem Host eingerichtet. Um diese Systeme skalieren zu können, wird je nach Art des Systems entweder die bestehende Instanz auf einen leistungsfähigeren Host verschoben oder es werden weitere Instanzen des Systems auf anderen Hosts aufgesetzt, die sich die vorhandene Last anschließend teilen.

Bei serviceorientierten Systemen werden die einzelnen Services hingegen einzeln ausgeliefert und skaliert.

Für die flexible Skalierbarkeit der Systeme und dem damit einhergehenden *deployment on demand* wird eine Art Service Provider benötigt, der bei Bedarf weitere oder leistungsfähigere Serviceinstanzen bereitstellen und ausliefern kann [Na16]. Diese Möglichkeiten führen zu deutlich erhöhter Komplexität der Infrastruktur, wenngleich sie enorme Vorteile für die Flexibilität eines Systems bieten.

Eine weitere Auswirkung der SOA auf den Auslieferungsprozess sind wesentlich kleinschrittigere Aktualisierungen und daraus resultierend wesentlich frequentiertere Auslieferungen ohne Downtime des Systems (beispielsweise durch Blue-Green oder Canary Auslieferung).

Insgesamt ist also erkennbar, dass die Verwendung einer SOA zu einem komplexeren Auslieferungsprozess führt. Um diese Komplexität handhabbar zu machen, werden zumeist umfangreiche DevOps Systeme für die Unterstützung im Entwicklungs- und Auslieferungsprozess verwendet.

Diese führen bei zielgerichteter Benutzung zu einer besseren Organisation, die sich bereits im Entwicklungsprozess positiv auswirkt. So ist beispielsweise eine gut organisierte Quellcodeverwaltung eine essenzielle Basis, um die isolierte Entwicklung homogener Services gut strukturiert durchführen zu können und aus diesen zu einem späteren Zeitpunkt komplexe Systeme zusammensetzen zu können. Darauf hinaus unterstützen diese Systeme das Konfigurations- und Releasemanagement, um die Zusammensetzung, Parametrisierung und Versionierung der serviceorientierten Softwarelösungen zu automatisieren. Dies reduziert Fehler und Inkonsistenzen, sowohl im Stadium der Entwicklung, als auch im Auslieferungsprozess. Eine gute Organisation und konsistente Daten sind beispielsweise auch nötig, um Systeme für Wartungsarbeiten, Recovery- und Testszenarien reproduzierbar zu machen.

In vielen DevOps Systemen lassen sich außerdem sogenannte *Pipelines* einrichten, über welche umfangreiche Ausführungsfolgen automatisiert werden können. Eine solche Pipeline kann zum Beispiel nach Fertigstellung einer Aktualisierung automatisiert gestartet werden. Im Durchlauf der Pipeline werden dann beispielsweise die Paketierung und Einrichtung der Software in einer VM angestoßen werden und der aktualisierten Service wird als VM-Image für den Kunden bereitgestellt oder direkt auf dessen Zielhosts ausgerollt.

Die Möglichkeit, Systeme in Form von einzelnen Services auszuliefern und dadurch unabhängig voneinander erweitern, skalieren und aktualisieren zu können, birgt je nach Anwendungsfall also enorme Vorteile, führt im gleichen Zug jedoch auch zu erheblicher Steigerung der Komplexität bei der Auslieferung. Diese Komplexität kann durch geeignete Hilfssysteme zwar wieder reduziert werden, dies ist jedoch mit sehr hohem initialen Aufwand verbunden.

Die Verwendung einer SOA lohnt sich hinsichtlich der Auswirkungen auf den Auslieferungsprozess also nur, wenn die dadurch gewonnenen Vorteile auch ausgeschöpft werden.

Literatur

- [Ad10] Adnan Gohar: Analyzing Service Oriented Architecture (SOA) in Open Source Products, Master Thesis, School of Innovation, Design and Engineering, 7.10.20210, URL: <https://www.diva-portal.org/smash/get/diva2:360992/FULLTEXT01.pdf>.
- [ADM06] Ang, H.-W.; Dandashi, F.; McFarren, M.: Tailoring DoDAF For Service-Oriented Architectures. In: MILCOM 2006 - 2006 IEEE Military Communications conference. S. 1–8, 2006, ISBN: 2155-7586.
- [Ax22] Axiomatics, Hrsg.: eXtensible Access Control Markup Language (XACML), 3.08.2022, URL: <https://axiomatics.com/resources/reference-library/extensible-access-control-markup-language-xacml>.
- [Bo93] Booch, G.: Object-Oriented Analysis and Design with Applications (2nd Ed.) Benjamin-Cummings Publishing Co., Inc, USA, 1993, ISBN: 0805353402.
- [Du22] Duden: monolithisch auf Duden online, 2022, URL: <https://www.duden.de/rechtschreibung/monolithisch>.
- [Er09a] Erl, T.: Service-oriented architecture: Concepts, technology, and design. Prentice Hall PTR, Upper Saddle River, NJ und Munich, 2009, ISBN: 0131858580.
- [Er09b] Erl, T.: SOA: Principles of service design. Prentice Hall, Upper Saddle River, NJ, 2009, ISBN: 9780132344821.
- [Fr] Frank Buschmann; Regine Meunier; Hans Rohnert; Peter Sornmerlad; Michael Stal: Wiley - Pattern-Oriented Software Architecture: A System of Patterns, Volume 1./.
- [G 07] G. A. Lewis; E. Morris; S. Simanta; L. Wrage: Common Misconceptions about Service-Oriented Architecture. In: 2007 Sixth International IEEE Conference on Commercial-off-the-Shelf (COTS)-Based Software Systems (ICCBSS'07). S. 123–130, 2007.
- [Gi22] Gillis. Alexander S.: Simple Object Access Protocol (SOAP), 2022, URL: <https://www.computerweekly.com/de/definition/Simple-Object-Access-Protocol-SOAP>.
- [He07] Heutschi, R.: Serviceorientierte Architektur: Architekturprinzipien und Umsetzung in die Praxis ; mit ... 52 Tabellen: St. Gallen, Univ., Diss., 2007. Springer, Berlin und Heidelberg, 2007, ISBN: 978-3-540-72357-8.
- [HR10] Haines, M. N.; Rothenberger, M. A.: How a service-oriented architecture may change the software development process. Communications of the ACM 53/8, S. 135–140, 2010, ISSN: 0001-0782.
- [KOS06] Kruchten, P.; Obbink, H.; Stafford, J.: The Past, Present, and Future for Software Architecture. IEEE Software 23/2, S. 22–30, 2006, ISSN: 0740-7459.

- [LF04] Lascelles, F.; Flin, A.: WS security performance. Secure conversation versus the X509 profile. 2004.
- [Na16] Nadareishvili, I.; Mitra, R.; McLarty, M.; Amundsen, M.: Microservice Architecture: Aligning Principles, Practices, and Culture./, 2016, URL: <https://docs.broadcom.com/doc/microservice-architecture-aligning-principles-practices-and-culture>.
- [PMA19] Ponce, F.; Márquez, G.; Astudillo, H.: Migrating from monolithic architecture to microservices: A Rapid Review: Concepción, Chile, November 4-9, 2019. In: 2019 38th International Conference of the Chilean Computer Science Society (SCCC). S. 1–7, 2019, URL: <http://ieeexplore.ieee.org/servlet/opac?punumber=8956485>.
- [Ra05] Rausch, T.: Service Orientierte Architektur: Übersicht und Einordnung, 26.12.2005, URL: https://web.archive.org/web/20081010033719/http://www.till-rausch.de/assets/baxml/soa_akt.pdf.
- [Ro12] Rotem-Gal-Oz, A.: SOA Patterns. Manning, 2012, ISBN: 978-1933988269.
- [SDS04] Stojanovic, Z.; Dahanayake, A.; Sol, H.: Modeling and design of service-oriented architecture. In (Wieringa, P., Hrsg.): 2004 IEEE international conference on systems, man & cybernetics theme. IEEE, Piscataway (N.J.), S. 4147–4152, op. 2004, ISBN: 0-7803-8567-5.
- [SHM08] Sanders, D. T.; Hamilton Jr, P. J.; MacDonald, P. R. A.: Supporting A Service-Oriented Architecture./, 2008, URL: <https://dl.acm.org/doi/pdf/10.5555/1400549.1400595>.
- [SM09] Savolainen Juha; Myllarniemi Varvana: Layered architecture revisited — Comparison of research and practice. In: 2009 Joint Working IEEE/IFIP Conference on Software Architecture & European Conference on Software Architecture. S. 317–320, 2009.
- [So22] Solutions, Martin Schmidt - Asgard: Serviceorientierte Architektur (SOA) - Knowledge Base - Asgard Solutions, 26.11.2022, URL: <https://www.asgard-solutions.de/KnowledgeBase/Softwaretechnik/Serviceorientierte-Architektur-SOA.aspx>.
- [St21] Storz, C.: The Journey to Microservices & Deployment Strategies: From monolith to microservices! Learn all about microservices & deployment strategies, hrsg. von harness.io, harness.io, 2021, URL: <https://harness.io/blog/microservices-deployment-strategies>.
- [Wi06] Wisniewski, T.; Whitehead, G.; Hinton, H.; Cahill, C.; Cantor, I.; Nate; Klingenstein; RI, B.; Morgan; John; Bradley; Individual, J.; Hodges; Individual, J.; Brennan, L.; Alliance, E.; Tiffany, L.; Alliance, T.; Hardjono, M.; Trustgenix: Assertions and Protocols for the OASIS Security Assertion Markup Language (SAML) V2. 0–Errata Composite./, 2006.

- [ZM09] Ziegler, S.; Müller, A.: Service-orientierte Architekturen: Leitfaden und Nachschlagewerk./, 2009, URL: <https://www.bitkom.org/sites/default/files/file/import/bitkom-soa-leitfaden.pdf>.

Infrastructure as Code - Nutzen für und Integration in DevOps

Gabriel Sperling¹, Reinhold Jooß²

Abstract: Im Umfang dieser Arbeit wird die Ausübung von Infrastructure as Code beleuchtet und wie diese zu einer effizienteren Softwareentwicklung führen kann. Dem Leser werden die Grundlagen von IaC und dessen Prinzipien nähergebracht. Dadurch werden die Vorteile im Gegensatz zu einer manuellen Konfiguration verdeutlicht. Außerdem werden die Anwendungsbereiche von IaC untersucht. Innerhalb dieser Ausarbeitung werden des Weiteren die Risiken welche während der Arbeit mit IaC auftreten können, dargelegt und anschließend erläutert wie das Eintreten dieser vorgebeugt werden kann. Eine Integration von IaC in Softwareentwicklung wird ebenfalls betrachtet und mit Hilfe des Softwaretools Ansible in einem Anwendungsbeispiel näher beleuchtet. Im Gesamten soll der Nutzen von Infrastructure as Code in DevOps verdeutlicht werden. Dabei soll diese Arbeit Aufklärung darüber schaffen wie es am besten eingesetzt wird und was es bei dessen Nutzung zu beachten gilt.

Keywords: Infrastructure as Code; DevOps; Cloud Age

1 Probleme der manuellen Konfiguration als Motivation für IaC

Dieser Abschnitt soll aufzeigen, welche Komplikationen und Hindernisse entstehen, sollte eine Infrastruktur manuell konfiguriert werden. Dadurch können neben den Risiken, welche dies für ein System mit sich bringt, auch weitere Vorteile einer automatisierten Konfiguration aufgezeigt werden und so eine Einleitung in den Nutzen von IaC gegeben werden.

Die Motivation, DevOps und IaC in die interne Infrastruktur zu integrieren, geht aus den Schwächen einer manuellen Konfiguration der IT-Infrastruktur hervor. Für ein Unternehmen ist es von großer Bedeutung eine stabile und leistungsstarke Umgebung für ihre Mitarbeiter und Software zu bieten. Durch hohe Qualität der Infrastruktur können Arbeitsprozesse effizienter durchgeführt und das Aufkommen von Fehlern und entstehenden Schäden verringert werden. Dementsprechend wird beim Aufbau der Infrastruktur ein großes Augenmerk auf die Zusammenstellung der Hardwarekomponenten gelegt. Die Anforderungen an die entstehende Recheneinheit müssen exakt definiert werden, um anschließend die benötigten Komponenten zusammenstellen und installieren zu können. Sobald die Infrastruktur

¹ DHBW Stuttgart Campus Horb, Informatik, Florianstraße 15, 72160 Horb am Neckar, Deutschland i20031@hb.dhbw-stuttgart.de

² DHBW Stuttgart Campus Horb, Informatik, Florianstraße 15, 72160 Horb am Neckar, Deutschland i20016@hb.dhbw-stuttgart.de

aufgebaut wurde, sollte diese dementsprechend ihren Zweck erfüllen, ohne dass in naher Zukunft mit Änderungen zu rechnen ist. Aufgrund von kontinuierlichen Änderungen in der Umgebung sind Überarbeitungen oder Erweiterungen der Infrastruktur jedoch beinahe unvermeidbar. Sobald diese auftreten muss vorerst erneut eine Planung stattfinden um den Änderungen gerecht zu werden, was einen hohen Ressourcenaufwand mit sich bringt. Die manuelle Konfiguration wird daher grundsätzlich zwar als stabile, jedoch sehr unflexible und kostenintensive Verwaltungsform betrachtet. Heutzutage wird dies als ein Hindernis betrachtet, da ein modernes Software-Engineering auf kurzschrifige und regelmäßige Releases abzielt, um eine erhöhte Flexibilität beim Implementierungsvorgang zu erreichen. Mit Hilfe von IaC soll daher eine automatische Konfiguration ermöglicht werden, um den Arbeitsaufwand für die Verwaltung der Infrastruktur zu verringern und damit für effizientere Releases im Software-Engineering zu sorgen.

Die Verringerung des Arbeitsaufwands kann anhand eines Beispiels verdeutlicht werden. Angenommen ein Unternehmen besitzt eine hohe Anzahl an verschiedenen Tochtergesellschaften, deren Webseiten über den Mutterkonzern geregelt werden. Sollte eine manuelle Konfiguration angewandt werden, muss vorerst ein leistungsstarker Server aufgebaut werden, welcher den Workload aller Tochtergesellschaften verarbeiten kann. Des Weiteren müssen anschließend die jeweiligen virtuellen Maschinen für die einzelnen Tochtergesellschaften vollständig aufgebaut und eingerichtet werden. Durch das Nutzen von Infrastructure as Code hätte ein Definieren einer VM als Code die Automatisierung der VM-Erstellung für die Tochtergesellschaften ermöglicht. Für identische Server hätte eine einzelne Definition des gewünschten Servers gereicht. Anschließend kann durch eine Automations-Software anhand der Definition eine beliebte Menge an Servern erstellt werden, welcher der Definition entsprechen. Falls nun bei einer der Tochtergesellschaften eine erhöhte Anzahl an Kunden auf deren Webseite festgestellt wird, wäre es möglicherweise notwendig die Speicherkapazität der VM zu erhöhen. Manuell müsste eine neue Festplatte eingebaut und auf dem Server konfiguriert werden. Anschließend muss sie der VM zugewiesen werden. Mit IaC könnte dieser Vorgang beschleunigt werden, indem die Speicherkapazität im Code als ein Parameter definiert wird, welcher beliebig angepasst werden kann. Der gesamte Overhead, welcher von Beginn an durch das Erwerben und Aufbauen der entsprechender Hardware entsteht, würde verfallen beim Anwenden von IaC. Grund hierfür ist das Umsteigen auf cloudbasierte Dienste diverser Anbietern in der heutigen Zeit. Ressourcen wie Speicherplatz, Arbeitsspeicher oder CPUs können beliebig zusammengestellt und eingesetzt werden um eine Infrastruktur zu generieren und zu erweitern. Die Vorteile einer Nutzung von IaC gegenüber einer manuellen Konfiguration werden in Abbildung 1 verdeutlicht.

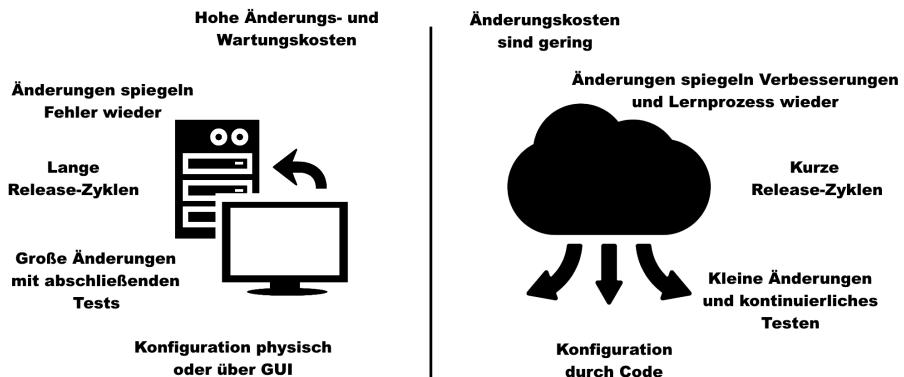


Abb. 1: Manuelle Konfiguration und Konfiguration in der Cloud Age im Vergleich [Mo20]

2 Grundlagen von IaC

In diesem Abschnitt soll dem Leser ein Einblick in die Grundlagen von IaC gegeben werden. Dabei wird vermittelt aus welchen Umständen IaC entstand und wieso dieses angewandt wird. Dafür werden verschiedenen Prinzipien betrachtet, die es beim Erstellen der Infrastruktur zu beachten gilt. Abschließend werden die verschiedenen Bestandteile der Infrastruktur, welche sich später mit IaC definieren lassen aufgezeigt.

2.1 Einführung in IaC

Auf Grund einer stetigen Nachfrage nach schnellen und regelmäßigen Software-Releases und der Entwicklung komplexer werdenden Systeme existieren mittlerweile viele technische und organisatorische Praktiken um die Herausforderungen des modernen Markts der Softwareentwicklung zu befriedigen. Diese Praktiken werden unter dem Begriff „DevOps“ zusammengefasst und versuchen die Softwareentwicklung (**Development**) mit den Aufgaben der IT (**IT-Operations**) zu kombinieren. Unter diese Praktiken fällt auch Infrastructure as Code [Gu19, S. 580-581].

Infrastructure as Code ist ein relatives junges Konzept der Informatik, welches aus komplexen Anforderungen der „Cloud Age“ (Bereitstellung von virtuellen Ressourcen über die Cloud) hervorkam. Durch die in der Cloud vorhandene Vielzahl an Ressourcen ergaben sich verschiedenste Anforderungen im Bereich der Verwaltung. Die verwendeten Ressourcen innerhalb eines Unternehmens welche die Infrastruktur zusammenstellten wurden in der

,Iron Age“ (physische Hardware zum Aufbau eines Systems) manuell konfiguriert und erstellt. Somit war es möglich die Infrastruktur gezielt auf den Anwendungsbereich zu spezifizieren und anzufertigen. Auf einen genauen Aufbau, hohe Stabilität und eine Gesamtheit der Struktur wurde somit großen Wert gelegt. Grund hierfür sind die hohen Kosten die eine Änderung der Struktur mit sich bringt. Im Umkehrschluss führt dies wiederum zur Gefahr zukünftige Änderungen des Systems mit einem hohen Aufwand zu verbinden. So kann es in extremen Fällen beispielsweise vonnöten sein, das komplette System aufgrund des rückständigen Zustands komplett abzubauen, um es zu erneuern. In der modernen Zeit wurde dieses Risiko deutlich verringert. Durch die Cloud Age sind die Kosten zur Bereitstellung und Änderung der Infrastruktur mit nur sehr geringem Aufwand verbunden, da die Ressourcen nicht mehr physisch sondern virtuell gegeben sind. Jedoch brachte diese Änderung der Technologie nicht direkt eine Vereinfachung der Systemverwaltung mit sich [Mo20]. Da sich eine Systemverwaltung der Cloud Age fundamental von einer in der Iron Age unterscheidet, muss die Arbeitsweise mit dieser angepasst werden, um ein optimales Vorgehen zu erreichen. Die geringen Kosten und Einfachheit kleiner Änderungen können in der Cloud Age ausgenutzt werden um eine erhöhte Qualität in geringer Zeit zu erreichen.

2.2 Eigenschaften und Prinzipien beim Arbeiten mit IaC

Mit Hilfe von IaC soll das Vorgehen die Infrastruktur zu verwalten vereinfacht und automatisiert werden. Sie ist eine Möglichkeit neue Zusätze an die Infrastruktur schnell auszuliefern, Änderungen einfach vorzunehmen und ein durch diese entstehendes Risiko zu verringern. Verwendete Tools in der Entwicklung und von weiteren Teilnehmern des Betriebs werden vereinheitlicht und die entstehenden Systeme möglichst sicher und effizient gestaltet, um später eine qualitativ hochwertige Verwaltung zu erlangen. Dabei ist es wichtig einen Fokus auf die Vorteile der Cloud Age, mit Hilfe verschiedener Core Practices und Prinzipien zu legen. Dazu gehört die gesamte Infrastruktur als Code zu definieren, um eine Wiederverwendbarkeit der Ressourcen zu garantieren, diese konsistent zu halten und eine möglichst hohe Transparenz für die Anwender zu generieren. Dafür wird zumeist eine deklarative Programmiersprache verwendet, wodurch der Code einen hohen Informationsanteil über die erstellte Systemressource liefert. Des Weiteren ist es wichtig den erstellten Code ständig zu testen und anschließend auszuliefern. Durch eine Integration neuer Änderungen, die im Laufe der Entwicklung getestet wird, können in geringeren Zeitabständen Erweiterungen ausgeliefert werden. Dadurch kann die Qualität des Endprodukts von Beginn an entwickelt werden, entgegengesetzt zum späteren Testen, in welchem die Qualität vielmehr „hineingetestet“ wird. Damit korrespondierend ist es Änderungen und Bestandteile klein und isoliert zu halten. Neben einer besseren Übersicht und Verständnis des Codes, wird somit für ein stabiles System gesorgt, welches bei Änderungen nicht leicht zerfällt. Wie unschwer zu erkennen ist, ähneln diese Praktiken dem Vorgehen, welches im Bereich des Software-Engineerings angewandt wird. Die verschiedenen Prinzipien welche beim Anwenden von IaC empfohlen werden besitzen eine ähnliche Affinität. So ist es von großer Wichtigkeit das zu entstehende System reproduzierbar zu gestalten. Dadurch ist das System bei Fehlerfällen

problemlos wiederherzustellen. Damit geht einher das Testen und die Reproduzierbarkeit dieser Tests konsistent zu halten und ebenso eine Replikation desselben Systems mühelos vorzunehmen. Im Falle von einer Verteilung mehrerer Systeme an verschiedene Verbraucher kann durch die Reproduzierbarkeit des Systems eine schnelle Auslieferung geschehen. Sobald eine Infrastruktur als Code definiert wurde sollte eine simple Reproduktion des Systems möglich sein, was dem Anwender enorme Freiheiten beim Erstellen, wie auch beim Löschen einer Infrastruktur gibt. Dadurch können die Risiken durch unvorhergesehenen Fehler geschwächt werden. Ein weiteres Vorgehen welches sich einen Nutzen aus der schnellen und dynamischen Änderung von IaC macht sind es eben diese Fehler in Kauf zu nehmen. Da durch IaC eine Beachtung von Fehlern nicht so ausschlaggebend ist wie bei manuellen Systemen, ist es von umso größerer Bedeutung, die zu definierenden Systeme als unzuverlässig zu betrachten und mit Fehlern zu rechnen. Ein Austausch des Systems, sollte dieses den Anforderungen ungenügend sein oder das Ersetzen eines unbrauchbaren Systems, aufgrund von Ausfällen oder Fehlern muss unbedingt in der Herstellung bedacht werden. Daraus folgend muss das System für eine ununterbrochene Laufzeit sorgen. Sollte eine Ressource ausfallen, kann das System durch das schnelle Ausliefern einer neuen Instanz der Ressource gerettet werden. Damit kommen wir zum nächsten Prinzip, der Verwerfbarkeit von zu erstellenden Infrastrukturtreilen. Es ist wichtig sich klarzumachen, wie dynamisch haltbar ein System durch die Virtualisierung in der Cloud Age ist. Die einzelnen Bestandteile sollten daher nicht als unerlässlich, sondern viel eher als entbehrbar betrachtet werden. Wird die Infrastruktur entsprechend designt, wird dem Anwender weitergehend Freiheit gewährt, dieses jederzeit neu zuformen, ohne die zu leistende Arbeit des Systems zu unterbrechen. Die einzelnen Bestandteile sollten dementsprechend möglichst minimal gestaltet werden, um Komplikationen, welche durch eine weite Variation der Bestandteile entstehen könnte, zu verhindern. Der Verwaltungsaufwand steigt nicht nur mit der Anzahl an Bestandteilen sondern auch mit deren verschiedenen Typen. Um den Aufwand möglichst gering zu halten, sollte daher mit einer möglichst geringen Anzahl an Typbestandteilen gearbeitet werden, da es beispielsweise leichter ist eine Vielzahl an identischen Servern zu handhaben, als eine geringe Anzahl vollkommen verschiedener Server. Hiermit wird wiederum das oben genannte Prinzip der Reproduzierbarkeit betont, welches dieses Vorgehen weitergehend ergänzt [Mo20].

Ein weiteres zu befolgendes Prinzip, ist die einheitliche Entwicklung des Systems. Das zukünftige Arbeiten wird unumgänglich eine Erweiterung des bestehenden Systems mit sich bringen. Dabei besteht die Gefahr das System durch ad hoc Änderungen weniger flexibel zu machen, da verschiedene Bestandteile mit ihrer ursprünglichen Aufgabe verfremdet werden. Die erfolgenden Erweiterungen sollten daher ebenfalls über den Code definiert und somit parametrisiert werden. Wiederum entsteht dadurch die Notwendigkeit alle Prozesse, die beim Verändern des Systems durchgeführt werden, zu automatisieren. Prinzipiell ist es gewünscht alle vorzunehmenden Prozesse reproduzierbar zu gestalten. Sollte eine Änderung on-the-fly vorgenommen werden besteht die bereits genannte Gefahr einer Inkonsistenz durch Verlust an Flexibilität, welche bei zukünftigen Anpassungen des Systems wiederum zu Fehlern führen können. Zwar sollten sich diese Fehler durch den einfachen Austausch

der Ressource beheben lassen, können jedoch beispielsweise für einen unerwünschten Ausfall des Systems oder Teile dessen führen. Eine Lösung können z.B. Skripts liefern, welche zur Konfiguration beisteuern. Dadurch wird gewährleistet das gleiche Vorgehen beim Konfigurieren einer Ressource vorzunehmen.

2.3 Aufbau einer Infrastruktur

Da nun die verschiedenen Punkte, welche beim Erstellen der Infrastruktur zu beachten sind, erläutert wurden, gilt es zu klären, welche Bestandteile der Infrastruktur angehören. Generell können diese in drei Kategorien unterteilt werden.

- Anwendungen (z.B. Anwendungspakete, Containerinstanzen)
- Laufzeitplattformen für Anwendungen (z.B. Server, Container Cluster, Datenbank Cluster)
- Infrastrukturplattformen (z.B. Recheneinheiten, Netzwerkstrukturen, Speicher)

Diese drei Kategorien können für ein besseres Verständnis als Schichtenmodell angesehen werden, welche sich gegenseitig ergänzen. Die Anwendungen stellen dabei die oberste Ebene der Struktur dar. Sie stellen dem Anwender die benötigten Werkzeuge für die gewünschte Arbeitsumgebung zur Verfügung. Geregelt werden diese in der Laufzeitplattform für Anwendungen. Hier können Konfigurationen vorgenommen werden, um die Anwendung mit verschiedenen Services und Berechtigungen auszustatten, so z.B. der Anbindung an eine Datenbank. Als unterste Ebene kann die Plattform für die Infrastruktur gesehen werden. In dieser werden die grundlegenden Ressourcen, welche zur Aufführung der einzelnen Bestandteile benötigt werden, bereitgestellt. Dazu gehören unter anderem Rechenressourcen, wie Virtual Machines oder auch Server und Container. Des Weiteren werden hier Speicherressourcen eingerichtet, welche in verschiedenen Formen bereitgestellt werden können. Beispielsweise Blockspeicher durch eine Virtualisierung von Partitionen, Dateisysteme auf einem Netzlaufwerk oder auch Objektspeicher in einer spezifischen Ausführung. Dabei können die Speichereinheiten natürlich gezielt auf die Anwendungsbereiche verteilt werden, z.B. für einen dedizierten Server oder verschiedene Container. Zuletzt werden in dieser Schicht auch verschiedene Netzwerkressourcen betrachtet. Somit können für die Anwendung nötige Gateways, Routing oder auch VPNs konfiguriert werden. Damit verbunden sind somit auch Proxys, sowie Netzwerkadressen oder DNS Einträge.

Für die Umsetzung von IaC gibt es zwei unterschiedliche Lösungsansätze. Hierbei wird zwischen statischer und dynamischer IaC unterschieden. Statische IaC-Lösungen stellen hierbei den traditionellen Lösungsweg dar. Hierbei werden Skripts erstellt und die Konfiguration wird auf die Infrastruktur einmal angewendet. Diese Infrastruktur ändert sich hierbei nicht, sofern das erstellte Skript nicht erneut angewendet wird, unabhängig davon ob das Skript manuell oder automatisiert durch eine Pipeline ausgeführt wird. Auf der anderen

Seite befinden sich die dynamischen IaC-Lösungen, die aktuell weiterentwickelt werden und damit statische Lösungen erweitern sollen. Das Prinzip dahinter beruht auf externen Signalen, von denen Teile der Skripts abhängen. Die Infrastruktur wird automatisch auf die ändernden Signale angepasst und wieder neu angewendet. Beispielsweise kann eine Infrastruktur in Abhängigkeit von der Last konfiguriert sein, sodass diese bei steigender oder sinkender Auslastung dementsprechend Ressourcen zur Verfügung stellt oder freigibt. Dies ist besonders vorteilhaft für eine sogenannte „pay-per-use“ Infrastruktur in beispielsweise einer Cloud. Im Gegensatz dazu ist es möglich diese Funktionalität durch eine Pipeline auf eine statische Lösung anzuwenden, jedoch wird dadurch der dynamische Aspekt ausgelagert, was unter anderem zu weiterem Aufwand führt. Dadurch wird es schwerer das gesamte System zu testen und zu analysieren, weshalb der dynamische Ansatz vorzuziehen ist. Jedoch wird damit auch eine robustere und automatisierte Lösung zum Testen der dynamisch implementierten Infrastruktur benötigt [So22].

Zu guter Letzt, wird in diesem Kapitel TOSCA (Topology and Orchestration Specification for Cloud Applications) kurz beschrieben. TOSCA stellt einen Standard für das Design von Infrastrukturen dar, der aktuell entwickelt und erforscht wird. Hierbei wird die Infrastruktur durch einen Graph visualisiert, der aus wiederverwendbaren Knoten und Kanten besteht. Dafür werden die Knoten und Beziehungen kategorisiert, um einfache Bausteine für die Entwicklung der Infrastruktur zur Verfügung zu stellen. Dabei bieten Plattformen, die den Standard implementieren bereits unterschiedliche Bausteine für bestimmte Anwendungen an, wie beispielsweise MySQL [Ar17]. Ein solcher Standard ist hilfreich, um die Kompatibilität von unterschiedlichen Plattformen und Werkzeugen für IaC zu verbessern. Dadurch werden neue Möglichkeiten in der Orchestrierung von Infrastruktur offen gelegt und komplexere Strukturen werden ermöglicht.

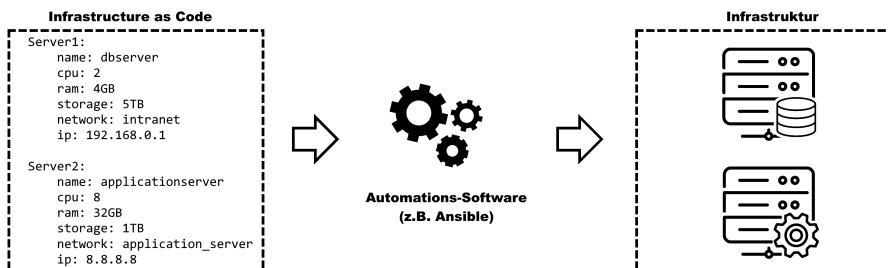


Abb. 2: IaC zur automatisierten Erstellung der Infrastruktur

3 Nutzen und Vorteile von IaC

Um den Nutzen von IaC zu verdeutlichen, können verschiedene Argumente gegen die Einführung von IaC betrachtet werden, welche [Mo20] hervorhob. Diese können durch die entstehenden Vorteile bei einer frühzeitigen Anwendung von IaC entkräftet werden.

Gegenargument: Eine Automatisierung von Änderungen bringt keinen ausreichenden Mehrwert bei einer geringen Anzahl an Änderungen.

Beim Planen der Infrastruktur wird vorzugsweise davon ausgegangen das System einmal zu erschaffen und es anschließend als finalisiert zu betrachten und es somit zukünftig nicht mehr entscheidend anzupassen. Objektiv gesehen bleiben Systeme jedoch in den wenigstens Fällen auf den anfangs festgelegten Nutzen beschränkt und werden stetig erweitert. Beispielsweise könnte durch ein Betriebssystem-Update die Notwendigkeit entstehen dutzende Server zu aktualisieren. Durch eine Automatisierung mit IaC wären diese Änderungen innerhalb kürzester Zeit erledigt. Vor allem in der Cloud Age profitieren Anwender von vielen Änderungen, um ihre Systeme stabiler zu machen. Eine Automatisierung der Infrastruktur bringt daher mehr Vorteile, als ein System durch viele manuelle Anpassungen instabil werden zu lassen.

Gegenargument: Es sollte zuerst gebuilded und anschließend automatisiert werden.
IaC anzuwenden zeigt zu Beginn einer Systemerstellung einen sehr hohen Arbeitsaufwand, da vorerst alle Ressourcen definiert werden müssen, bevor diese tatsächlich erstellt werden können. Meist wird daher als schnellere Alternative ein neues cloudbasiertes System manuell erstellt, mit der Absicht dessen Automatisierung später hinzuzufügen. Dadurch entstehen grundsätzlich drei Probleme. Erstens werden viele der Vorteile einer Automatisierung nicht umgesetzt. Eine schnelle Verteilung der benötigten Systeme kann nicht durchgeführt werden, nachdem ein System bereits größtenteils aufgebaut wurde. Zweitens können Tests leichter erstellt und direkt in die Automatisierung eingebunden werden. Dadurch kann beim ersten Aufbau der Infrastruktur bereits eine schnelle Fehlerbehebung stattfinden und ein Rebuild bei Problemen vorgenommen werden. Als letzter Punkt kann festgestellt werden, dass die Automatisierung bestehender Systeme ein sehr komplexes Vorhaben ist. Die bestehende Infrastruktur muss möglicherweise angepasst und erweitert werden. Aufgrund der Probleme sollte die Automatisierung bereits zu Beginn und während des Aufbaus einer Infrastruktur betrachtet werden. Durch IaC wird ein solches Vorgehen gewährleistet.**Gegenargument: Es muss sich zwischen Schnelligkeit und Qualität entschieden werden.**
Es besteht grundsätzlich die Annahme bei der Bearbeitung einer Aufgabe die Qualität des Produkts und die Schnelligkeit der Erstellung in einen relativen Vergleich stellen zu müssen. Es kann keine hohe Qualität gewährleistet werden, wenn die Erstellung des Systems möglichst schnell ausgeführt werden muss. Dementsprechend kann kein qualitativ hochwertiges und stabiles System erstellt werden, wenn kein großer Zeitraum dafür verwendet wird. Wie sich laut [HKF18] herausstellt ist diese Annahme jedoch falsch. Teams, welchen die Prinzipien der agilen Verfahren aus dem Software-Engineering Bereich bewusst sind, schaffen es demnach auch durch schnelle Releases ein hohe Qualität aufzuzeigen. Dieses Methodik kann auf IaC übertragen werden, um somit eine stabile Infrastruktur zu erschaffen, welche einen Fokus auf Schnelligkeit sowie Qualität legt.

4 Integration von IaC in Softwareentwicklung

Nachdem nun der Nutzen und die Vorteile von Infrastructure as Code aufgezeigt wurden, geht es weiter mit dem praktischen Teil von IaC: Die Integration von IaC in der Softwareentwicklung. Hierbei gibt es drei verschiedene und wichtige Methoden, die bei der Integration zu beachten sind, die im Weiteren genauer erläutert werden: Neben dem Konzept, dass die gesamte Infrastruktur als Code definiert wird, ist es wichtig diese in kleine Teile zu unterteilen und den definierten Code kontinuierlich zu testen.

Weiterhin werden in diesem Kapitel sogenannte „Bad“ und „Best“ Practices für die Implementierung von Infrastructure as Code dargestellt. Damit wird ein Überblick darüber gegeben, was zu beachten ist und wie es richtig gemacht wird.

Zu Letzt wird knapp zusammengefasst, welche Tools für die Umsetzung von IaC genutzt werden. Hierbei wird insbesondere auch darauf eingegangen, was bei der Verwendung dieser Tools zu beachten ist.

Die erste der drei wichtigsten Methoden, welche bei der Integration von IaC zu beachten sind, beschäftigt sich mit den Überlegungen, was alles als Code definiert werden sollte. Offensichtlich ist es am besten jegliche Bestandteile der Infrastruktur in Code zu formulieren, da es dadurch theoretisch möglich wird die Prozesse um jede einzelne Komponente zu automatisieren. Praktisch ist dies jedoch nicht umsetzbar, weswegen es essentiell ist die wichtigsten Bestandteile zu identifizieren und diese bestmöglich zu definieren. Dazu gehören unter anderem der sogenannte „infrastructure stack“. Dieser beinhaltet alle Elemente, die über eine Cloud Plattform bereitgestellt werden können, die für die Softwareentwicklung notwendig sind. Außerdem sollen Abbildungen von Servern als Code definiert werden, da hierdurch das mehrfache Aufsetzen von Servern erleichtert und automatisiert wird. Ebenso sollten jegliche wiederverwendbare Konfigurationen in IaC-Skripts definiert werden, wie beispielsweise Server-Konfigurationen, die Anwendungen, Benutzer und notwendige Dateien enthalten. Wenn nun der bestmögliche Teil der Infrastruktur in Code definiert wurde, müssen die entstandenen Dateien in einem geeigneten System oder geeigneter Anwendung verwaltet werden. Diese entsprechen nicht den Tools, die für das Deployen der Skripts zuständig ist. Beispiel hierfür sind Version Control Systems, wie Git.

Der Code wird entweder in einer deklarativen Sprache oder imperativen Sprache geschrieben. Deklarative Sprache haben den Vorteil, dass sie nützlich für die Definition des gewünschten Endzustands einer Komponente sind. Hiermit können übliche Konfigurationen einfach und konsistent verwendet werden. Auf der anderen Seite stehen imperative Sprachen, die flexibler sind. Mit dem imperativen Ansatz ist es möglich Variablen einzubauen, die die Infrastruktur auf eine bestimmte Situation anpasst, wobei der Code wiederverwendet wird und nicht für jeden Fall neu geschrieben werden muss [Mo20]. Deshalb ist es notwendig für den jeweiligen Use Case abzuwägen, welche Art der Sprache am besten geeignet ist. Es ist auch möglich beide Sprachen durch die Verwendung verschiedener Tools zu mischen, wobei

dies mit Vorsicht zu genießen ist. Ebenso stellen die richtige Auswahl der Komponenten und Systeme eine große Rolle, wie auch die Auswahl eines geeigneten Version Control Systems.

Jetzt wo der Code erstellt ist, beschäftigt sich die zweite Methode mit dem Testen des Codes. Das Ziel nach dem sich gerichtet werden kann besteht hierbei, dass der Code immer auf die Struktur erfolgreich anwendbar sein soll. Die wichtigsten Dinge, die getestet werden sollten sind, dass der Code die Infrastruktur in der gewollten Konfiguration hinterlässt, dass die Leistung und Sicherheit der Infrastruktur den Anforderungen entspricht und dass der Code überhaupt anwendbar ist. Ein Problem stellen hier Tests für deklarativen Code dar, da diese Tests bei jeder Änderung des Codes auch geändert werden müssen und damit wenig Nutzen und viel Aufwand haben. Progressive Testing stellt im Allgemeinen eine gute Methode zum Testen dar. Dabei werden schnelle und einfache Tests zuerst durchlaufen, die dann auf komplexeren und aufwändigeren Tests aufbauen. So können grobe Fehler früher entdeckt und durch einfache Tests besser aufgedeckt werden. Eine Pipeline zu verwenden ist empfehlenswert, da dadurch der Testprozess automatisiert und mit dem Deployment verbunden werden kann. Für jede Änderung können dann die Tests durchlaufen werden, um somit die Zuverlässigkeit des Codes zu garantieren. Beim Testen sind ebenfalls die Abhängigkeiten der einzelnen Komponenten zu beachten. Heißt für Tests sind teilweise Nachahmungen von Services oder fertige Komponenten insgesamt notwendig [Mo20].

Die letzte wichtige Methode bei der Umsetzung von IaC besteht daraus, die Infrastruktur in kleine Komponenten aufzuteilen. Dabei ist es wichtig, dass Änderungen auf einer Komponente so wenig Änderungen wie möglich auf anderen Komponenten verursachen. Ebenso sollen zusammenhängende Komponenten eine starke Bindung haben. Wichtige Vorgehensweise bei der Definition der Komponenten sind folgende [Mo20]:

- Dopplungen vermeiden
- Unabhängige Teile schaffen
- Komponenten mit nur einem Zweck erstellen
- Komponenten voneinander isolieren
- keine Kreisbeziehungen schaffen

Nun können die Best und Bad Practices für die Integration von IaC in der Softwareentwicklung zusammengefasst werden. Hierbei werden neben allgemeinen Eigenschaften der Infrastruktur auch die Verwendung von Tools und den Skripts selber angesprochen.

Zu guter Letzt ist es noch wichtig die Tools, die für eine effektive Umsetzung von IaC verwendet werden, anzusprechen. Vorweg ist zu erwähnen, dass es kein „one-size-fits-all“ Tool gibt, da in der Praxis viele unterschiedliche Werkzeuge existieren, die alle ihren eigenen Beitrag zu IaC leisten. Dazu gehören unter anderem Tools für das Bereitstellen von Infrastruktur, wie Terraform oder Tools für das Erstellen und Konfigurieren

Best Practices	Bad Practices
Parametrisierte Skripts	Hard-coded Werte in Skripts
Schnell auf und abbauende Infrastruktur	Weiterhin viel manuelle Konfiguration
Reusability fördern	Zu große Skripts
Richtige Kombination von Tools	Zu viele Tools verwenden

Tab. 1: Best und Bad Practices bei der Integration von IaC [Gu19]

von Containern, wie Docker. Ebenso existiert Ansible, was für die Konfiguration von verschiedenen Komponenten verwendet werden kann. Wichtig ist es gut abzuwägen, welche Tools verwendet und wie sie miteinander verbunden werden, da es ansonsten zu Risiken führen kann, die im Voraus vermieden werden können [Gu19]. Dies wird später näher erläutert.

5 Risiken und Herausforderungen beim Arbeiten mit IaC

Während der Integration von IaC-Praktiken in der Softwareentwicklung entstehen unterschiedliche Herausforderungen, die bestanden werden müssen und entsprechende Risiken, die daraus resultieren können. In diesem Kapitel werden insbesondere die Probleme des sogenannten „Configuration Drift“, der Qualitätssicherung und des Sicherheitsaspektes von IaC aufgezeigt. Hierbei wird ebenfalls darauf eingegangen, was getan werden kann, um diese Risiken zu mindern und wie gegen die Probleme vorgegangen wird.

5.1 Problem: Sicheres Ändern der Skripts und Infrastruktur

Ein Hauptmerkmal von IaC ist die Flexibilität, die es ermöglicht die Infrastruktur nach Belieben anzupassen. Dafür werden dementsprechend Skripts verwendet, die zum Teil automatisiert ausgeführt werden. Wenn hier mit einer dynamischen Infrastruktur gearbeitet wird, ist es allgemein sinnvoll lieber öfters Änderungen durchzuführen und diese klein zu halten. Damit sollen Teile des Systems verändert werden und nicht das ganze System auf einmal. Somit wird es ermöglicht, die Infrastruktur kontinuierlich auszubauen und anzupassen, ohne darauf warten zu müssen, dass das gesamte System steht. Aufgrund dessen entsteht das erste Problem: kleine Änderungen allein, reichen meistens nicht aus, um das System in einem brauchbaren Zustand zu hinterlassen. Es benötigt mehrere kleine Änderungen. Hierfür gibt es zwei Möglichkeiten, dieses Problem zu umgehen. Erstens, es wird versucht den alten Code solange beizubehalten bis genug neue Einzelteile bestehen, um den alten Code zu ersetzen. Die zweite Möglichkeit wäre, sowohl den alten als auch den neuen Code in den Skripts zu verwenden und dann explizit mit Variablen anzugeben, welche der beiden Funktionalitäten genutzt werden soll.

Ein weiteres Problem, das beim kontinuierlichen Ändern der Infrastruktur besteht, ist dass bestimmte Services durch die Änderungen ausfallen können. Genau hierfür ist es

wichtig Prozesse zu integrieren, mit denen regelmäßig der geschriebene Code zuerst getestet wird, bevor Änderungen durchgeführt werden. Eine Pipeline kann hierfür Abhilfe schaffen. Natürlich ist es nicht möglich Fehler auszuschließen, weshalb es notwendig ist einen Plan bereit zu haben, um ein System wieder herstellen zu können, falls eine Änderung zum Ausfall führt. Hierbei ist es wichtig im Voraus zu planen, da es große Folgen haben könnte, wenn kritische Services nicht mehr erreichbar sind. Mit Hilfe eines Plans, ist es möglich voraus zuschauen und im Notfall schnell zu handeln, ohne überlegen zu müssen, wie Systeme wiederhergestellt werden können.

Das letzte Problem, das bei Änderungen auftreten kann, hängt mit Teilen der Infrastruktur zusammen, die Daten halten. Diese Daten können durch zu viele unsichere Änderungen korrupt werden. Hierbei gibt es auch mehrere Möglichkeiten, um die Auswirkungen dieses Problems gering zu halten. Einmal ist es möglich, die wichtigen Instanzen von den Änderungen auszuschließen, sodass es erst gar nicht zur Datenkorruption kommen kann. Dies ist gewiss nicht immer sinnvoll, weshalb es geschickter ist vor Änderungen ein Backup der Daten durchzuführen, um im Fall der Korruption, die Daten wieder herstellen zu können [Mo20, S. 355-384].

5.2 Problem: Sicherheitslücken in Skripts

Ebenso wie bei normalem Code sind IaC-Skripts gewisse Sicherheitslücken ausgesetzt, die am besten von Anfang an vermieden werden sollen. Hierbei ist es am häufigsten der Fall, dass kritische Informationen, wie Passwörter, in den Skripts unverschlüsselt wiederzufinden sind. Dies sollte aus offensichtlichen Gründen vermieden werden. Meistens entsteht dieses Problem dadurch, dass für eine schnelle Probe die kritische Information eingetragen wird mit der Intention diese danach wieder zu löschen. Oftmals wird dies nicht sauber durchgeführt und die Informationen bleiben weiterhin bestehen und bilden somit das Risiko. Ein weiteres häufiges Problem steht der Nutzen des HTTP Protokolls ohne Transport Layer Security (TLS) in den Skripts dar. Um eine gewisse Sicherheit zu gewährleisten, sollte natürlich TLS und HTTPS verwendet werden. Weitere Probleme sind, dass schwache Verschlüsselungsalgorithmen eingesetzt werden oder dass leere Passwörter verwendet werden. Diese stellen ganz klar ein unnötiges Risiko dar, welches durch gründliche Vorsorge verhindert werden kann [RPW19, S. 165-172].

Einfach Gegenmaßnahmen, wie der Einsatz von HTTP mit TLS und das Verwenden von starken Passwörtern können schnell die Sicherheit von IaC-Skripts erhöhen. Ebenso sollten vertrauliche Informationen mit den richtigen Algorithmen verschlüsselt werden.

5.3 Problem: Zu viele unterschiedliche Tools

Im vorherigen Kapitel zur Integration von IaC wurde aufgezeigt, dass mittlerweile viele unterschiedliche Tools existieren, die verschiedene Funktionalitäten bieten. Viele dieser Tools ermöglichen andere Prozesse, weshalb es oftmals dazu kommt, dass mehrere Tools

gleichzeitig verwendet werden. Nicht selten kommt es dann dazu, dass zu viele Tools auf einmal eingesetzt werden, mit denen viele Nebenwirkungen einhergehen. Ein Hauptproblem stellt dar, dass in den meisten Fällen die Tools eine eingeschränkte Kompatibilität aufweisen, was unter anderem darauf zurückzuführen ist, dass jedes Tool ein eigenes Format für die Beschreibung der Infrastruktur verwendet. Dies macht die Nutzung unübersichtlich und es wird allgemein schwerer eine Konsistenz über alle Plattformen beizubehalten. Damit wird ebenfalls die Qualitätssicherung aufwändiger und unter anderem eingeschränkt, was sich im Ganzen negativ auf die Flexibilität der Infrastruktur auswirkt. Um diesem Risiko entgegenzuwirken, ist es zum einen sinnvoll Standards wie TOSCA einzusetzen und aktiv weiterzuentwickeln. Außerdem sollten eine Kombination von unterschiedlichen Werkzeugen mit gutem Gewissen verwendet werden. Damit bleibt eine einheitliche Struktur gegeben und die Vorteile von IaC werden optimal genutzt [Gu19, S. 580-587].

5.4 Problem: Testen der Skripts

Es wurde bereits erklärt, warum es wichtig ist ein Testprozess zu integrieren, damit das Ändern der Skripts nicht zu einer fehlerhaften Infrastruktur führt. Genau dies stellt jedoch noch eine weitere Hürde dar. Denn neben Standardpraktiken, fehlt es noch an Frameworks und Umgebungen, um zuverlässig und automatisiert IaC-Skripts testen zu können. Das Hauptproblem besteht hierbei, dass normalerweise gewartet werden muss, bis die gesamte Infrastruktur umgesetzt ist, bevor geprüft werden kann, ob die Konfiguration richtig eingestellt wurde. Dazu kommt noch, dass die flexible Verteilung es schwer macht, mit einem Debugger vernünftig Fehler aufzudecken und zu beheben [Gu19, S. 580-587].

Deshalb ist es wichtig entsprechende Tools und Mechanismen zu entwickeln, um Integrationstests zu ermöglichen. Jede Konfiguration bzw. jede Änderung der Konfiguration muss getestet werden, weshalb es sehr unpraktisch ist, immer wieder warten zu müssen bis die ganze Infrastruktur auf die neuen Anforderungen angepasst wurde. Natürlich gibt es bereits Lösungen, die versuchen das Verifizieren der Skripts zu automatisieren, jedoch sind diese noch zu wenig verbreitet und es wird in Zukunft wichtig sein, sich weiter für die Entwicklung dieser Tools einzusetzen [So22].

5.5 Problem: Configuration Drift

Das letzte Problem, welches unter anderem größere Folgen mit sich trägt, ist der sogenannte „Configuration Drift“. Hierbei sind Systeme betroffen, die gleiche oder ähnliche Instanzen von Komponenten aufweisen. Configuration Drift beschreibt das Problem, dass nach einer gewissen Zeit diese eigentlich gleichen Instanzen unterschiedliche Eigenschaften zeigen. Eine Ursache für dieses Problem ist, dass nur an einzelnen Komponenten der Infrastruktur Änderungen vorgenommen werden anstatt an allen, die betroffen sind. Dabei ist es nicht relevant, ob diese Änderungen manuell oder automatisiert geschehen. Das kann dazu führen, dass Änderungen, die für alle Instanzen vorgesehen sind, nicht mehr mit allen kompatibel

sind. Dies wiederum, fordert eine aufwändige Fehlersuche, die ohnehin durch fehlender Debugging-Möglichkeiten schwer genug ist. Deshalb kann es im Gesamten dazu führen, dass ein größerer Widerstand vor dem Einsatz von automatisierten Skripts besteht, da der Configuration Drift zu Inkompabilitäten führt, die schwierig zu finden sind [Mo20, S. 17-20].

Um dies gegenzuwirken, ist es zum einen wichtig die Zeit zu minimieren zwischen dem automatisierte Prozesse laufen. Dadurch wird verhindert, dass Updates in den Tools und kleine Änderungen zu weiteren Problemen führen. Außerdem sollten „ad hoc“ Änderungen komplett vermieden werden, da diese meist die Ursache des Configuration Drifts darstellen. Natürlich ist es meistens praktischer schnell eine einzelne Instanz auf neue Anforderungen anzupassen, jedoch werden dadurch eigentlich gleiche Systeme nicht mehr synchron und somit nicht mehr gemeinsam veränderbar sein. Dies schränkt die Flexibilität erheblich ein. Ein konkreter Ansatz, um diesen Problemen entgegenzuwirken stellt eine „Schedule“ dar, die dafür sorgt, dass der Code in den Skripts regelmäßig auf die Infrastruktur angewendet wird, auch wenn sich nichts geändert hat. Damit ist es auch möglich von einer zentralen Einheit jede betroffene Instanz zu erreichen und somit einheitlich und gleichzeitig Änderungen anzuwenden [Mo20, S. 349-351].

6 Anwendungsbereiche und weitere Entwicklungen von IaC

Neben dem Einsatz in der Softwareentwicklung, gibt es noch andere Bereiche, die von den Vorteilen von IaC profitieren können. Einer dieser Bereiche stellt die Cloud dar, genauer gesagt, das Bereitstellen von Ressourcen einer Cloud-Infrastruktur. Neben den Charakteristiken der Cloud, die die Welt der Informatik zu dem machen was sie heute ist, ist es wichtig Wege zu finden, um die Vorteile der Cloud effizient nutzen zu können. Dafür kann IaC eingesetzt werden, um die Prozesse rund um die Verwendung von Cloud-Ressourcen zu vereinfachen.

Die Nutzung von Cloud-Ressourcen wird charakterisiert durch die virtuelle Abbildungen derer Leistung auf physisch existierende Hardware. Ebenso sind die Prozesse zur Bereitstellung der Leistungen automatisiert und schnell durchführbar. Hierfür ist es wichtig, dass diese Systeme schnell anpassbar und leicht zu verwalten sind. Diese Schnelligkeit soll dazu führen, dass diese Systeme eine bessere Qualität haben. Das lässt sich damit begründen, dass unter anderem Änderungen nicht aufwändig und Verbesserungen häufiger und schneller möglich sind. Solche Systeme sind ebenfalls in kleine Komponenten aufgeteilt, die flexibler zu verwalten sind [Mo20].

Um nun Cloud Dienstleistungen effektiv konfigurieren und bereitstellen zu können ist hierfür Infrastructure as Code notwendig. Manuelle Praktiken stellen hier eine große Hürde gegenüber dem schnellen und flexiblen Charakter der Cloud dar. Deshalb können IaC-Skripte verwendet werden, um die automatisierte und kontinuierliche Konfiguration zu ermöglichen. Die Skripte schaffen eine Grundlage für das schnelle Anpassen der Cloud-Infrastruktur, während die einfache Beschaffenheit der Skripte eine leichte Verwaltung zur Folge hat. Über

die Skripte ist es ebenfalls ein geringerer Aufwand, die kleinen, flexiblen Komponenten effizient und übersichtlich zu verwalten und nachzuvollziehen. Damit wird eine klare Übersicht der Cloud-Komponenten erhalten, ohne dass sich wichtige Informationen im Hintergrund verstecken.

Der Einsatz von IaC für die Bereitstellung und Konfiguration von Cloud-Ressourcen ist jedoch noch nicht komplett ausgereift. Für IaC werden viele Tools bereitgestellt, was an sich positive Auswirkungen auf die Menge der Möglichkeiten hat, jedoch tauchen hier wieder Probleme auf, was ebenfalls bereits bei den Risiken angesprochen wurde. Ein weiteres Problem mit den Tools ist, dass jedes eine eigene Art und Weise hat die Infrastruktur zu beschreiben und unter anderem auch auf spezielle Anwendungsfälle begrenzt ist [Bh18]. Dies führt dazu, dass für den Einsatz von IaC für die Cloud spezialisiertes Personal notwendig ist, das diese Tools beherrscht und die Anwendungsfälle kennt [Ar18]. Um dieses Problem zu lösen, gibt es einen Ansatz, der auf „Model Driven Engineering“ basiert. Hierbei wurden Tools entwickelt, die abstrakte, high-level Modelle in IaC-Code umwandeln. Damit soll die Entwicklung von IaC-Skripten vereinfacht und effizienter werden.

In [Ar18] wird ein Ansatz basierend auf Model Driven Engineering unter dem Namen „DICER“ untersucht und erläutert. Dabei bietet DICER mehrere Features, die die Effizienz der Erstellung von Skripten für IaC steigern soll. Zum einen bestehen die Modelle, in denen die Cloud-Infrastruktur abgebildet werden soll, aus Komponenten, die im Software Engineering üblich sind. Heißt es sollen beispielsweise mit UML „deployment-diagrams“ eingesetzt werden. Dadurch soll, die Zeit zur Entwicklung von IaC verringert werden. Ebenso können hierbei häufig auftretende Komponenten wiederverwendet werden, in dem diese als vorgefertigte Teile des Codes bereitgestellt werden. Durch die Modelle wird eine weitere Ebene der Abstraktion eingeführt, weshalb DICER für eine Prüfung der Konsistenz sorgt, um den Entwicklungsprozess von IaC für die Cloud effektiv zu gestalten. DICER wird als Plugin für die Eclipse IDE installiert worüber dann die UML Deployment Modelle in ausführbaren IaC-Code umgewandelt werden. Dieser Code kann dann schließlich über ein geeignetes Tool eingelesen und auf eine Cloud-Infrastruktur angewendet werden. Die Modelle werden in drei Teile umgewandelt:

- IaC-Code nach TOSCA Standard, der die Hauptkomponenten der Infrastruktur und deren Eigenschaften enthält
- Chef „recipes“, die Aktionen enthalten, wie das Installieren, Starten und mehr
- Python Code, der für die Weiterreichung von Parametern zwischen Komponenten zuständig ist

Ein Anwendungsfall für den Einsatz von DICER stellen die Data-Intensive Architectures (DIA) dar. Diese stellen viel Rechenleistung zu Verfügung, um Daten aus Quellen einzulesen, diese zu Verarbeiten und dann wieder neue Daten zu produzieren. DIAs bestehen aus unterschiedlichen Technologien, um ihre Funktionen bereitzustellen zu können. Dazu gehören

Technologien, um Daten zu speichern, Nachrichtenwarteschlangen und Ausführungsumgebungen. Dabei sind diese in Cluster aufgeteilt und bestehen aus mehreren verteilten Systemen in der Cloud. Dazu kommt noch, dass sie an vielen Stellen konfiguriert werden können, um für eine optimale Verarbeitung der Daten zu sorgen. Um das beste Ergebnis zu erzielen, müssen Konfigurativen immer wieder neu aufgesetzt und getestet werden [Ar18]. Dafür eignet sich der Ansatz mit Model Driven Engineering am besten, da sich damit diese komplexe Systeme schnell und einfach modellieren lassen, wodurch dann DICER ausführbaren IaC-Code generiert.

7 Anwendungsbeispiel mit Ansible

Dieses Kapitel beschreibt das Automatisierungstool *Ansible*. Anhand von Anwendungsbeispielen soll gezeigt werden wie IaC mit Hilfe des Tools in der Praxis angewandt werden kann.

7.1 Was ist Ansible?

Ansible ist ein Tool, das von Red Hat entwickelt wurde, um verschiedenste Aufgaben im Bereich der IT zu automatisieren. Dazu gehören Configuration Management, Network Management und noch viele mehr. Durch den Aufbau auf Infrastructure as Code, ist Ansible einfach zu bedienen und ermöglicht übersichtlich verteilte Systeme zu verwalten [Re22b].

Eine Umgebung, in der Ansible eingesetzt wird um Systeme und deren Konfigurationen zu verwalten, besteht im Endeffekt aus drei Komponenten: Die erste Komponente stellt das zentrale Kontrollsysteem dar, auf dem Ansible installiert ist. Hier ist die zweite Komponente wiederzufinden, das sogenannte *Inventory*. In dieser INI- oder YAML-Datei werden die Zielsysteme, die Ansible verwalten soll, festgehalten. Diese stellen die dritte Komponente der Umgebung dar [Re22a]. Damit nun Aufgaben und Prozesse auf den Zielsystemen ausgeführt werden können, werden diese in sogenannten *Playbooks* definiert. Playbooks sind weitere YAML-Dateien, die von Ansible ausgeführt werden können, um die Konfiguration der Zielsysteme automatisieren zu können. In den Playbooks werden Module aufgerufen, die vordefinierten Code darstellen. Ansible kopiert diese und führt sie über eine SSH-Verbindung auf dem Zielsystem aus. Hierbei bietet Ansible viele eingebaute Module, aber es gibt auch weitere, die für bestimmte Services, wie beispielsweise Amazon Web Services, existieren [Re22b].

Die Vorteile von Ansible sind zum einen die einfache und übersichtliche Bedienbarkeit durch die deklarativen Inventory- und Playbook-Dateien. Ein weiterer Vorteil ist die Eigenschaft von Ansible „agentless“ zu sein. Das heißt, damit Ansible mit den Zielsystemen kommunizieren kann, ist es nicht notwendig extra Software auf den Systemen zu installieren. Es muss lediglich Python auf den Zielsystemen installiert sein, da damit die Module ausgeführt

werden, und eine SSH-Verbindung zum Zielsystem muss aufgebaut werden können. Ansible wird ebenfalls durch eine aktive Community unterstützt, die viele Erweiterungen anbietet, um weitere Services anzuwenden und mehr Use Cases ermöglichen zu können.

Im weiteren Abschnitt wird für ein Anwendungsbeispiel eine Instanz einer „Amazon Elastic Compute Cloud (EC2)“ verwendet. Hier wird kurz erläutert was eine EC2-Instanz ist. Über die Amazon Web Services Cloud kann eine EC2-Instanz Rechenleistung für unterschiedliche Anwendungen bereitstellen. Dabei wird ein virtueller Server mit dessen Eigenschaften für CPU-Leistung, Speicher und mehr konfiguriert. Dieser läuft dann auf der Infrastruktur von Amazon Web Services. Diese Instanzen können leicht skaliert werden, sodass Änderungen in der angeforderten Rechenleistung gleich umgesetzt werden können. Damit werden Kosten gespart und mehr Rechenleistung kann schnell bereitgestellt werden. EC2 wird für viele unterschiedliche Anwendungsbereiche eingesetzt. So können die Instanzen für Anwendungs-Server, kleine Datenbanken, Virtual Maschines und mehr verwendet werden. Da die Konfiguration der Instanzen sehr ausgiebig sein kann und viele Instanzen auf einmal laufen können, eignet sich Ansible sehr gut für die übersichtliche Verwaltung solcher EC2-Instanzen [Am22].

7.2 Anwenden von Ansible

In diesem Abschnitt werden Beispielkonfigurationen eines Hosts mit Hilfe von Ansible gezeigt. Dafür wurde eine EC2-Instanz von Amazon verwendet, um den Host zu definieren. Die Installation und Initialisierung von Ansible, sowie das verbinden einer EC2-Instanz mit dem Inventory werden nicht erläutert. Für nähere Informationen dazu wird empfohlen einen Einstiegskurs zu Ansible durchzuführen.

Um in Ansible Aktionen durchzuführen werden neben ad hoc Befehlen vor allem Playbooks verwendet, um verschiedene Befehle automatisiert auf dem Zielsystem durchzuführen. Die ad hoc Befehle werden über die Kommandozeile ausgeführt und dienen vor allem dazu kleine Anfragen auf einem der angelegten System durchzuführen. Dadurch kann verhindert werden eine extra Datei anzulegen um einen Befehl zu Testen. So kann zum Beispiel ein Ping an eines der angelegten Host-System gesendet werden um zu erkennen, ob dieses erreichbar ist. In Abbildung 3 wird ein solcher Ping auf allen Hosts ausgeführt, wobei nur einer angelegt ist. Daher wird nur ein pong empfangen. Als Inventory Informationen wird *hosts* verwendet. Als User welcher mit dem Server kommunizieren soll wird *ec2-user* angegeben.

```
suuyaas@SuuyaasPC:/mnt/s/Suuyaas/Documents/DHBW/AdSWE/AnsibleTest$ ansible all -i hosts -u ec2-user -m ping  
44.208.36.243 | SUCCESS => {  
    "ansible_facts": {  
        "discovered_interpreter_python": "/usr/bin/python3"  
    },  
    "changed": false,  
    "ping": "pong"  
}
```

Abb. 3: ad hoc Befehl in Ansible, zur Ausführung eines Pings auf ein angelegtes Hostsystem

Da eine Automatisierung durch Ansible mit IaC ermöglicht werden soll, liegt das Hauptaugenmerk um Konfigurationen durchzuführen auf den Ansible-Playbooks. Diese führen verschiedene Tasks nacheinander aus. Sie werden durch den Befehl *ansible-playbook -i [Zielinventory] [Playbook-Datei]* ausgeführt. An oberster Stelle eines Playbooks wird dessen Namen angegeben und zur Eingrenzung auf welchem Host und als welcher User dieses ausgeführt werden soll. In Listing 1 ist eine solche Deklaration für ein Playbook zu sehen, welches für die Installation von MariaDB verantwortlich sein soll.

```
1 ---  
2   - name: mariadb test #playbook  
3     hosts: TestServer #host name  
4     remote_user: ec2-user #remote user on host platform  
5     become: yes #root rights
```

List. 1: Beschreibung eines Playbooks mit dem Namen *mariadb test*

Innerhalb dieses Playbooks können nun verschiedene Tasks angegeben werden. Diesen werden prozedural abgearbeitet. Des Weiteren können innerhalb des Playbooks verschiedene Variablen definiert werden, welche beispielsweise beim Ausführen des Playbooks zusammen mit dem Befehl angegeben werden können oder durch einen anderen Playbook Ablauf gesetzt werden. In Listing 2 ist zu erkennen, wie Tasks zum Installieren von MariaDB, dem anschließenden Erstellen einer zugehörigen conf-File, einer log-File und dem abschließenden Starten von MariaDB definiert sind. Ebenfalls anzumerken ist das Nutzen der Variable *mysql_port*. Diese wird in der Konfiguration Datei, welche als Template in Zeile 17 angegeben wird verwendet und kann somit angepasst werden. Des Weiteren wird in Zeile 18 ein *notify* Befehl genutzt, welcher anhand des Namens innerhalb der Handler in Zeile 26 geregelt wird. Ein Notify-Befehl wird nur einmal ausgeführt, solange keine changes erfasst werden. Sollte sich das Playbook in gewissen Maßen ändern, wird der Handler zum Neustarten nicht ausgeführt ein weiteres Mal durchgeführt. Ändert sich jedoch der Task *create mysql configuration file*, also wird beispielsweise die *mysql_port*-Variable angepasst, so muss MariaDB neugestartet werden um weiterhin korrekt durchgeführt zu werden. Da sich die Datei *src_my.cnf.j2* nun verändert hat, bemerkt Ansible die Änderung und ein Change wird getriggert. Anhand des Changes wird nun der Notify Befehl ausgeführt und der Handler *restart mariadb* startet den Service neu. Des Weiteren kann innerhalb eines Tasks die Ausführung eines anderen Tasks inkludiert werden, indem mit *include_tasks: [Pfad zur Task-Datei]* eine Referenz auf den entsprechenden Task gesetzt wird (Zeile 11).

```

7 vars:
8   mysql_port: 3306
9
10 tasks:
11 - include_tasks: selinux.yaml #Putting the include at the top of tasks means
12   it will be executed first
13 # path: // Same direcotry, so no path is required
14 - name: installing mariadb
15   yum: name=mariadb-server state=latest
16
17 - name: create mysql configuration file
18   template: src=my.cnf.j2 dest=/etc/my.cnf
19   notify: restart mariadb #triggers at end of task and only once (multiple
20     notifies possible)
21
22 - name: create mariadb log file
23   file: path=/var/log/mysqld.log state=touch owner=mysql group=mysql mode=0775
24
25 - name: start mariadb service
26   service: name=mariadb state=started enabled=yes
27
28 handlers: #handlers for notifies
29 - name: restart mariadb
30   service: name=mariadb state=restarted
#-include: selinux.yaml // Putting the include at the end of the file/top
      hierarchy executes the target as a standard playbook. That means e.g.
      conditionals or failtesting of this playbook won't apply on the included
      one.

```

List. 2: Beschreibung eines Playbooks mit dem Namen *mariadb test*

Die Ausführung Tasks können wiederum anhand von Conditionals präziser geregelt werden. So kann die Ausführung eines Tasks beispielsweise auf eine Variable beschränkt werden oder anhand von weiteren Parametern wie dem Betriebssystem oder Error-Logs des Zielhosts ausgemacht werden (siehe Listing 3).

```

1 vars:
2   unicorn: true
3
4 tasks:
5 - name: don't install on debian machines
6   yum: name=httpd state=latest #install httpd server
7   when: (ansible_os_family=="RedHat" and
8         ansible_distribution_major_version=="6") #condition on operating system

```

```
9   - name: are unicorns real or fake
10    shell: echo "unicorns are fake"
11    when: not unicorn #unicorn variable is true, task will be skipped
```

List. 3: Das MariaDB Beispiel wird umgeben von einem Block, um ein Conditional auf alle Tasks zu beziehen

Um Tasks eine weitere Abstraktionsebene zu schaffen können diese in Blocks eingeteilt werden. Blöcke können beliebig in weitere Blöcke unterteilt werden. Auf einen Block bezogen können anschließend wiederum Befehle ausgeführt. Vor allem Conditionals oder verschiedene Error-Handlings bieten hier einen Vorteil um eine Ansammlung an Tasks bedingt zu betrachten und auszuführen. Das Beispiel in Listing 4 spiegelt das MariaDB Beispiel wieder, jedoch erfolgt die Installation ausschließlich unter der Bedingung, dass der Zielhost der Betriebssystem-Familie von "RedHat" angehört (Zeile 25).

```
1 tasks:
2   - block:
3     - name: installing mariadb
4       yum: name=mariadb-server state=latest
5
6     - name: create mysql configuration file
7       template: src=my.cnf.j2 dest=/etc/my.cnf
8       notify: restart mariadb #triggers at end of task and only once
9         (multiple notifies possible)
10
11    - name: create mariadb log file
12      file: path=/var/log/mysqld.log state=touch owner=mysql group=mysql
13        mode=0775
14
15    - name: start mariadb service
16      service: name=mariadb state=started enabled=yes
17
18    when: ansible_os_family=="RedHat" #Conditional for whole block instead of
19      each task
20    become: yes
```

List. 4: Das MariaDB Beispiel wird von einem Block umgeben, um ein Conditional auf alle enthaltenen Tasks zu beziehen

Eine Weitere Möglichkeit um ein Ansible Projekt weitergehend zu abstrahieren und strukturieren wird durch Roles bereitgestellt. Diese können über eine Ansible-Galaxy erstellt werden. Eine neue Rolle kann durch den Befehl `ansible-galaxy init [Role-Name]` erstellt werden. Ein großer Vorteil von Ansible-Galaxies ist die Verfügbarkeit von diversen Community-Galaxies. So kann auf <https://galaxy.ansible.com/> nach verschiedenen von Anwendern erstellten Rollen gesucht werden, welche durch vorgefertigte Konfigurationen

eine Verwaltung bestimmter Systeme erlauben und frei erweitert werden können.

Das Erstellen einer Rolle kann vereinfacht als eine Unterteilung des Playbooks in verschiedene Bereiche betrachtet werden. Eine Rolle kann anschließend über ein übliches Ansible-Playbook gestartet werden (siehe Listing 5).

```

1  ---
2  - name: deploy common and mariadb role
3    hosts: TestServer
4    remote_user: ec2-user
5    become: yes
6
7    roles:
8      - common
9      - roles_example

```

List. 5: Ein Playbook zum Ausführen der Rollen *common* und *roles_example*

Innerhalb des Role-Directories bestehen nun verschiedene Directories für die jeweiligen Bestandteile eines Playbooks. Diese sind wie folgt.

- *tasks*, zur Definition der verschiedenen Tasks.
- *handlers*, zur Definition verschiedener Handler.
- *vars*, zur Definition verschiedener Variablen.
- *defaults*, zur Definition von Standardwerten, sollten Variablen nicht explizit gesetzt sein.
- *templates*, zum Anlegen verschiedener Dateien, welche als Vorlage dienen.
- *meta*, zum Definieren verschiedener Metadaten der Rolle. Eine nützliche Funktion hierbei ist das Definieren verschiedener Role Dependencies. Diese binden die Rolle und deren Tasks an die entsprechende Definition, wodurch z.B. eine Rolle auf alle Zielsysteme angewandt werden kann, solange diese mit einem bestimmten Betriebssystem ausgeführt werden (siehe Listing 6).

```

1  dependencies:
2    - {role: apache, when: "ansible_os_family=='RedHat'"} #Depend on Apache
      role, if our host is of OS type RedHat

```

List. 6: Ausschnitt der *main.yaml* des *meta*-Directories, welche eine Role Dependency definiert

Eine weitere Funktion von Ansible, welche diese Arbeit abschließen soll, ist die Möglichkeit ein Error-Handling innerhalb der Tasks zu betreiben. Da ein Ansible Playbook unterbrochen

wird, sobald ein Fehler bei dessen Ausführung auftritt, ist ein geeignetes Error-Handling von Vorteil, um ungewollte Abläufe innerhalb der Automatisierung zu verhindern. Hierbei sind vor allem die Befehle *ignore_error*, welcher die Ausführung des Playbooks trotz eines Errors fortführt und *failed_when*, welcher einen Error anhand einer Bedingung erzeugt. Des Weiteren kann mit Hilfe des *changed_when* Befehls ein *changed*-Event in einen erfolgreichen *ok*-Event, anhand einer Bedingung umgewandelt werden. So kann z.B. eine Änderung der conf-File (siehe Listing 2) ignoriert werden, um den *notify*-Befehl zu überspringen. Ein Beispiel zum Error-Handling ist in Listing 7 zu sehen.

```
1  ---
2  - name: testing error handling
3    hosts: TestServer
4    remote_user: ec2-user
5    become: yes
6
7    tasks:
8      - name: testing ignore errors
9        user: name=Max password={{uPassword}}
10       ignore_errors: yes
11
12      - name: next task
13        shell: echo hello world
14
15      - name: quick echo
16        shell: echo $PATH
17        register: result
18        changed_when: false #results in "ok" on change
19
20      - debug: msg="Stop running playbook if the play failed"
21        failed_when: result is failed
22
23      - name: echo failed #doesnt actually fail
24        shell: echo I failed
25        register: output
26
27      - debug: msg="Okay really stop the playbook this time"
28        failed_when: output.stdout.find("failed") != -1
29
30      - name: just adding another task in here to show you that it will stop
31        shell: echo hello world
```

List. 7: Ansible Beispiel Task mit Error-Handling

Literatur

- [Am22] Amazon Web Services Inc.: What is Amazon EC2?, 2022, URL: <https://docs.aws.amazon.com/AWSEC2/latest/UserGuide/concepts.html>, Stand: 05. 12. 2022.
- [Ar17] Artac, M.; Borovssak, T.; Di Nitto, E.; Guerriero, M.; Tamburri, D. A.: DevOps: Introducing Infrastructure-as-Code. In: 2017 IEEE/ACM 39th IEEE International Conference on Software Engineering Companion. S. 497–498, 2017, ISBN: 978-1-5386-3868-2.
- [Ar18] Artac, M.; Borovsak, T.; Di Nitto, E.; Guerriero, M.; Perez-Palacin, D.; Tamburri, D. A.: Infrastructure-as-Code for Data-Intensive Architectures: A Model-Driven Development Approach. In (IEEE Computer Society Conference Publishing Services, Hrsg.): 2018 IEEE 15th International Conference on Software Architecture Companion. S. 156–165, 2018, ISBN: 978-1-5386-6585-5.
- [Bh18] Bhattacharjee, A.; Barve, Y.; Gokhale, A.; Kuroda, T.: (WIP) CloudCAMP: Automating the Deployment and Management of Cloud Services. In (IEEE Computer Society Conference Publishing Services, Hrsg.): 2018 IEEE International Conference on Services Computing. S. 237–240, 2018, ISBN: 978-1-5386-7250-1.
- [Gu19] Guerriero, M.; Garriga, M.; Tamburri, D. A.; Palomba, F.: Adoption, Support, and Challenges of Infrastructure-as-Code: Insights from Industry. In (IEEE Computer Society Conference Publishing Services, Hrsg.): 2019 IEEE International Conference on Software Maintenance and Evolution (ICSME). S. 580–589, 2019, ISBN: 978-1-7281-3094-1.
- [HKF18] Humble, J.; Kim, G.; Forsgren, N.: Accelerate: The Science Behind Devops: Building and Scaling High Performing Technology Organizations. IT Revolution Press, 25 NW 23rd Pl, Suite 6314, Portland, OR 97210, 2018, ISBN: 978-1942788331.
- [Mo20] Morris, K.: Infrastructure as code: Dynamic systems for the cloud age. O'Reilly, Beijing u. a., December 2020, ISBN: 978-1-098-11467-1.
- [Re22a] RedHat: Getting started with Ansible, 2022, URL: https://docs.ansible.com/ansible/latest/getting_started/index.html, Stand: 05. 12. 2022.
- [Re22b] RedHat: How Ansible Works, 2022, URL: <https://www.ansible.com/overview/how-ansible-works>, Stand: 05. 12. 2022.
- [RPW19] Rahman, A.; Parnin, C.; Williams, L.: The Seven Sins: Security Smells in Infrastructure as Code Scripts. In (IEEE Computer Society Conference Publishing Services, Hrsg.): 2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE). S. 164–175, 2019, ISBN: 978-1-7281-0869-8.

- [So22] Sokolowski, D.: Infrastructure as Code for Dynamic Deployments. In (Association for Computing Machinery, New York NY, United States, Hrsg.): Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE '22). 2022, ISBN: 978-1-4503-9413-0.

Kongressstag 2

Einsatz intelligenter Werkzeuge zur Softwareentwicklung

Jonathan Schwab¹, Felix Wochele², Jonas Weis³

Abstract: Zahlreiche moderne Werkzeuge zur Unterstützung der Softwareentwicklung sind mittlerweile nutzbar. Diese Arbeit schafft einen Überblick über verschiedene Einsatzbereiche. Behandelt werden die Bereiche Codevervollständigung, Codegenerierung, Codeanalyse, Refactoring, Dokumentation und Kollaboration. Neben der Vorstellung grundlegender Konzepte und Umsetzungen, steht dabei die Analyse des Nutzens und der möglichen Risiken oder Grenzen im Vordergrund. Außerdem gibt die Vorstellung einer oder mehrerer Werkzeuge je Kategorie die Möglichkeit zum Transfer in die Praxis. Neben der Funktionalität wurde bei der Auswahl vor allem auf die Unterstützung möglichst vieler Programmiersprachen geachtet.

Keywords: Künstliche Intelligenz; Softwareentwicklung; Intelligente Werkzeuge; Codevervollständigung; Refactoring; Codegeneration; Codeanalyse; Dokumentation; Kollaboration

1 Einleitung

Das Thema beschäftigt sich mit verschiedenen intelligenten Werkzeugen zur Unterstützung bei der Softwareentwicklung. Viele Prozesse der Softwareentwicklung enthalten repetitive, triviale oder inferentielle Vorgänge. Der Einsatz von intelligenten Werkzeugen soll hier Abhilfe schaffen. Konkrete Vorteile können Zeiter sparnis, eine Steigerung der Codequalität oder eine bessere Nachverfolgbarkeit sein. Dies steigert die Wirtschaftlichkeit von Softw areprojekten und schafft dem Entwickler mehr Zeit für komplexe Arbeit. Um die Werkzeuge differenziert zu bewerten ist es notwendig die zugrunde liegenden Konzepte zu verstehen. Somit lassen sich neben den Vorteilen auch vorhandene Grenzen und Risiken erkennen.

Ziel dieser Arbeit ist das Schaffen eines Überblickes über intelligente Werkzeuge zur Erleichterung der Softwareentwicklung. Dabei sollen Konzepte aus den Bereichen Codevervollständigung, Codegenerierung, Codeanalyse, Refactoring, Dokumentation und Kollaboration genauer betrachtet werden. Wesentlicher Bestandteil ist das Herausstellen von Nutzen und Risiken beim Einsatz derartiger Werkzeuge. Zu jedem Konzept wird eine Marktrecherche durchgeführt, welche besonders geeignete Werkzeuge vorstellen soll. Anhand dessen wird eine Handlungsempfehlung gegeben, welche beim Bearbeiten zukünftiger Projekte zur Unterstützung herangezogen werden kann.

¹ DHBW Stuttgart - Campus Horb, Informatik, Florianstraße 15, 72160 Horb am Neckar, Deutschland, i20030@hb.dhbw-stuttgart.de

² DHBW Stuttgart - Campus Horb, Informatik, Florianstraße 15, 72160 Horb am Neckar, Deutschland, i20036@hb.dhbw-stuttgart.de

³ DHBW Stuttgart - Campus Horb, Informatik, Florianstraße 15, 72160 Horb am Neckar, Deutschland, i20035@hb.dhbw-stuttgart.de

Mittlerweile wird die Auswahl an intelligenten Werkzeugen zur Softwareentwicklung immer größer. Neben herkömmlichen Verfahren ermöglicht vor allem künstliche Intelligenz einen großen Fortschritt in diesem Bereich. Aufgrund der Breite dieses Sektors kann nicht jedes intelligente Werkzeug vorgestellt werden. Daher folgt eine Fokussierung auf die bereits aufgezählten Konzepte, da diese sowohl im Unternehmens-, als auch privaten Bereich Potential besitzen. Beispielsweise wären Werkzeuge zum Testen von Software eine weitere Möglichkeit gewesen. Diese sehen wir allerdings aufgrund der Vorlesung *Software Engineering I* als ausreichend bekannt an. Das Thema *Requirements Engineering* bietet auch eine Vielzahl an Werkzeugen an. Es ist allerdings derart umfangreich, dass es wenige Seiten nicht ordnungsgemäß darstellen könnten. Nur eines von vielen Unterthemen wäre das Testmanagement. Außerdem werden derartige Werkzeuge in der Regel vom Unternehmen festgelegt und variieren damit stark. Sie finden im privaten Bereich teilweise kaum Einsatz. Daher werden sie in der folgenden Arbeit nicht näher betrachtet. Der Fokus liegt stattdessen auf Verfahren, welche auch im privaten Bereich, beispielsweise in Freizeitprojekten, Anwendung finden. Informationen zum Thema *Requirements Engineering* können aber beispielsweise unter [Ch13] gefunden werden.

2 Codevervollständigung

Sind in einem Projekt die Anforderungen definiert und von Hand erste Modelle und Architekturen entworfen, müssen diese implementiert werden. Dieses Kapitel behandelt die automatische und intelligente Codevervollständigung in Integrierten Entwicklungsumgebungen, also die Unterstützung während diesem Vorgang.

2.1 Allgemeine Konzepte

Automatische Codevervollständigung ist heutzutage in nahezu allen IDEs vorhanden. Es ist eines der meistgenutztesten Features von Softwareentwicklern [GKF06]. Anhand verschiedener Ansätze werden dabei wahrscheinliche Vervollständigungen vorgeschlagen. Diese bestehen aus verschiedenen Dingen und variieren je nach Sprache. Übliche Vorschläge sind aber beispielsweise im Kontext verfügbare *Methoden, Events, Klassen, Interfaces, Strukturen, Enums, Schlüsselwörter, lokale Variablen, Parameter oder Attribute, Dateien, Farben und Datentypen*. Durch verschiedene Methoden wird die Wahrscheinlichkeit dieser berechnet und in der Regel anhand einer Textbox zur Auswahl gestellt. Alternativ wird die wahrscheinlichste Option direkt ausgegraut angezeigt und kann beispielsweise durch die Tab-Taste akzeptiert werden. Abbildung 1 zeigt diese Optionen anhand von Visual Studio auf.

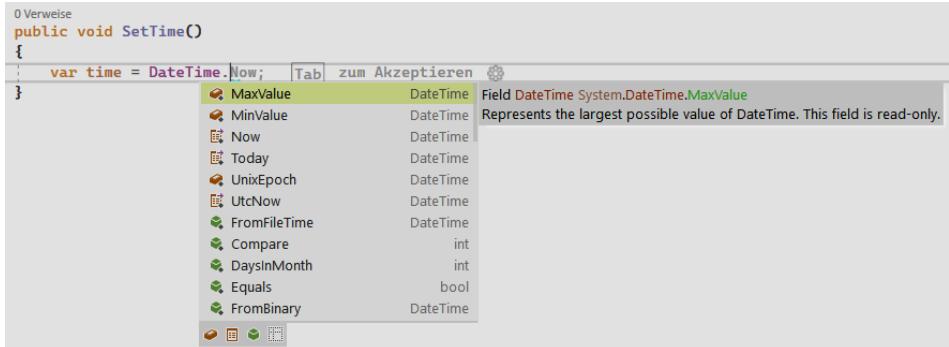


Abb. 1: Code Vervollständigung in Visual Studio 2022

Je mehr Zeichen bereits geschrieben wurden, desto genauer können die Vorschläge werden. Dies liegt daran, dass durch die schon geschriebenen Zeichen gewisse Optionen auszuschließen sind. Die Eingabe wird also ständig geparst und es ergeben sich mögliche Ergebnisse mit dem gleichen Präfix. Das Parsen kann beispielsweise durch reguläre Ausdrücke oder das Aufbauen von abstrakten Syntaxbäumen umgesetzt werden. Hierbei wird von der lexikalischen Analyse gesprochen. Ein anderer klassischer Ansatz ist die semantische Analyse. Diese verwendet ein Grammatikmodell der Sprache um anhand definierter Regeln nur Methoden anzuzeigen, welche auch legal sind. Dabei werden beispielsweise private Methoden herausgefiltert, wenn auf diese kein Zugriff vorhanden ist [MCB15]. In der Regel werden beide Analysearten verwendet.

Die Bestimmung der besten Ergebnisse ist dann durch verschiedene Implementierungen umsetzbar. Einfachste Lösung ist die Sortierung nach alphabetischer Reihenfolge oder dem Zeitpunkt der letzten Modifikation einer Methode. Auch denkbar ist die Priorisierung nach Lokalität, also beispielsweise zuerst die aktuelle Klasse, dann das Projekt und zuletzt Imports [RL08].

Naheliegend ist auch die Sortierung nach der Anzahl der bisherigen Verwendungen eines Vorschlags. Diese Arten zählen zu den statistischen. Auch diese erzielen heutzutage bessere Ergebnisse, da öffentliche Repositories die notwendigen Daten zur Verfügung stellen. Anhand dieser können die Auftrittswahrscheinlichkeiten verschiedener Elemente ausgelesen werden um somit bessere Vorschläge zu erzielen. Verwendung findet dabei beispielsweise das N-Gram-Modell. Dieses zählt die Vorkommen verschiedener Kombinationen aus N Elementen [Ro00]. Problematisch ist, dass diese Modelle nur Abhängigkeiten zwischen wenigen Elementen feststellen. Außerdem funktioniert es nur, wenn Statistiken zu allen Element-Kombinationen vorhanden sind.

Moderne Systeme nutzen immer häufiger einen anderen Ansatz: Künstliche Intelligenz. Im Gegensatz zur statistischen Analyse werden nicht nur Statistiken gesammelt sondern anhand von Daten wiederkehrende Muster erkannt. Diese schaffen es besser umfangreichen

Kontext miteinzubeziehen und auf in den Trainingsdaten nicht vorhandene Elemente zu reagieren. Zur genauen Umsetzung gibt es verschiedenste Optionen. In [Sc19b] werden einige der verwendbaren rekurrenten neuronalen Netze gegenüber gestellt. Erwähnenswert ist außerdem der Algorithmus Best-Matching-Neighbour, welcher sich als sehr effizient herausgestellt hat [BMM09].

2.2 Nutzen und Risiken

Die Nutzung von Codevervollständigung bringt zahlreiche Vorteile mit sich. Der größte ist die Zeitersparnis. Diese unterscheidet sich je nach Effektivität des Tools. Statt dem Ausschreiben ganzer Methodenaufrufe reicht oftmals der erste Buchstabe. Selbst wenn der gesuchte Vorschlag erst an dritter Stelle ist, muss nur zweimal die Pfeiltaste zur Auswahl betätigt werden. Im schlimmsten Fall müssen bereits mehrere Buchstaben vorhanden sein, damit die gesuchte Empfehlung kommt. Trotzdem findet eine enorme Zeiteinsparung statt. Gleicher gilt auch für die anderen Elemente, wie Variablen und Co. Einige Tools sprechen von ca. 40% weniger Tastaturanschlägen [Ki22]. Außerdem werden durch die Vorschläge Tipp- und Logikfehler vermieden. Syntaktisch oder semantisch nicht korrekte Eingaben rufen keine Vorschläge hervor. Wird stattdessen die Eingabe durch Vorschläge vervollständigt, kann von syntaktischer und semantischer Korrektheit ausgegangen werden. Ein weiterer Vorteil ist, dass kein Detailwissen mehr benötigt wird. Sobald in objektorientierten Sprachen beispielsweise ein Punkt hinter das Objekt gesetzt wurde, werden dessen zugreifbare Methoden und Felder angezeigt. Dabei wird in der Regel bereits gefiltert, ob eine Zuweisung oder nur ein Aufruf stattfindet. Findet ersteres statt, werden nur Methoden und Felder mit Rückgabewert angezeigt. Besonders bei Nutzung fremder Bibliotheken ist dies von großem Vorteil. Somit können selbst schlecht dokumentierte Anwendungen und Bibliotheken verwendet werden, sofern die Namen verständlich gewählt wurden. Durch die Nutzung von Inline-Dokumentation sind die jeweiligen Methoden oftmals sogar direkt in der IDE beschrieben. Neben dem Vermeiden von Logik-Fehlern helfen diese Tools bereits beim Formulieren der Logik. Beispielsweise wird bei Zuweisung eines Wertes an eine Variable automatisch erkannt, wenn der Datentyp nicht übereinstimmt. Daher folgt ein Vorschlag zum Casting der Eingabe. Bei verschiedenen Collection-Arten werden dagegen beispielsweise auf diese anwendbare Strukturen wie `foreach` vorgeschlagen. Das Gerüst dieser kann automatisch vervollständigt werden. Ein weiteres Beispiel ist die Rückgabe in einer Methode. Wird das passende Keyword geschrieben, kommen beispielsweise Vorschläge aus lokalen Variablen mit dem passenden Rückgabetypr. Die Beispiele sind zahlreich.

Diese Vorteile müssen allerdings mit Vorsicht genutzt werden. Die automatische Vervollständigung verleitet beispielsweise dazu, Vorschläge blind anzunehmen und einfach davon auszugehen, dass die dahinterliegende Funktionalität passend ist. Besonders bei fremden Bibliotheken wäre es dagegen sinnvoll, dies zu überprüfen oder die technische Dokumentation zu studieren. Gegebenenfalls hat eine verwendete Methode besonders

schlechte Performance, ist bereits veraltet oder nicht Thread-safe, dies wird aber für die Anwendung benötigt. Hier muss zwingend vorsichtig vorgegangen werden.

2.3 Marktanalyse

Es gibt verschiedene Tools zur Nutzung mit mehreren Sprachen. Bezuglich Microsoft und Visual Studio wird **IntelliSense** bzw. **IntelliCode** verwendet. Dieses unterstützt JavaScript, TypeScript, JSON, HTML, CSS, SCSS, C++, C#, J#, Visual Basic, XML, XSLT, SQL. Bei Nutzung der IDE Visual Studio Code ermöglichen Erweiterungen außerdem dutzende weitere Sprachen. Es basiert auf tausenden Repositories, deren Qualität durch eine Mindestanzahl an Sternen sichergestellt werden soll.

Ähnlich viele Sprachen in verschiedenen Editoren unterstützt die alleinstehende Anwendung **Kite**. Diese wurden anhand von über 25 Millionen Dateien trainiert und soll die Tastenanschläge um ca. 40% reduzieren [Ki22].

Besonders erwähnenswert ist allerdings das Projekt **Tabnine**. Es handelt sich um eine moderne, sehr umfangreiche Erweiterung. Bezuglich der Funktion Codevervollständigung ist diese kostenlos. Unterstützt werden 20 verschiedene IDEs und folgende Sprachen: Angular, C, C++, C#, CSS, Dart, Go, Haskell, HTML, Java, Javascript, Kotlin, Matlab, NodeJS, ObjectiveC, Perl, PHP, Python, React, Ruby, Rust, Sass, Scala, Swift und TypeScript. Erkennbar ist das definierte Ziel, nicht auf spezielle Sprachen beschränkt zu sein. Besonderheit ist außerdem die Möglichkeit eigene Modelle zu trainieren indem ausgewählte Repositories verbunden werden. Dies ermöglicht die optimalen Vorschläge bezogen auf spezielle Projekte. So kann beispielsweise der Programmierstil innerhalb eines Unternehmens besser miteinbezogen werden. Dazu muss allerdings die kostenpflichtige Version (12€ pro User je Monat) in Verwendung sein.

The screenshot shows a code editor in Visual Studio Code with the following code:

```

const express = require('express');
const app = express();
const port = process.env.PORT || '8080';

app.get('/', (req, res) => {
  res.send('hello tabnine');
})

app.listen(port, () => [
  col
])
  ↴ ⚒ console.log('Listening at port %s', port); tabnine
  ↴ ⚒ console.log('Listening at port: ' + port);
  ↴ ⚒ console.log('Listening at port',
  ↴ ⚒ console.log(port
  ↴ ⚒ console.log('Listening on port %
  ↴ ⚒ confirm
  ↴ ⚒ console
  ↴ ⚒ const
  ↴ ⚒ continue
  ↴ ⚒ ...
]

```

A code completion dropdown is open at the end of the line `app.listen(port, () => [`. The suggestions include various logging statements using the `console.log` method, such as `console.log('Listening at port %s', port)` and `console.log('Listening at port: ' + port)`. Other suggestions like `confirm`, `console`, `const`, and `continue` are also visible.

Abb. 2: Code Vervollständigung mit TabNine in VS Code

Abbildung 2 zeigt die Vervollständigung durch TabNine in Visual Studio Code. Erkennbar ist, dass mehr als nur ein Methodename vervollständigt wird. Anhand des Befehls `app.listen` (`port, ...`) schlägt die Software einen passenden Text für das Logging unter Einbezug der Parameter vor. Als Grundlage von *Tabnine* gilt das Modell GPT-2. Die verwendeten Daten stammen von GitHub-Repositorien, für welche Qualität sichergestellt wurde. Neue Daten finden regelmäßig Einsatz im Training, um auch neuste Entwicklungen miteinzubeziehen.

3 Codegenerierung

In bestimmten Fällen ist es mittlerweile nicht mehr notwendig Quellcode von Hand zu schreiben. Stattdessen kann spezifiziert werden, was der Programmcode tun soll und aus dieser Information automatisch Code generiert werden. Dieser Anwendungsfall kann zwar nicht so weitläufig eingesetzt werden wie die Codevervollständigung, spart stellenweise aber noch mehr Arbeit ein.

3.1 Allgemeine Konzepte

Programmiersprachen dienen dazu Datenstrukturen und Algorithmen formal aber vom Menschen lesbar auszudrücken. Hierbei unterscheidet man zwischen High-Level- und Low-Level Programmiersprachen (Abbildung 3). Low-Level Programmiersprachen befinden sich auf einer niedrigen, High-Level Programmiersprachen auf einer höheren Abstraktionsebene. Ziel dieser Abstraktion ist es, die Lesbarkeit zu erhöhen und die Komplexität

zu reduzieren. Hierfür bieten High-Level Programmiersprachen beispielsweise Kontrollstrukturen und Datentypen an. Bei der Codegenerierung wird diese Idee aufgegriffen und erweitert. Im Allgemeinen geht es um die Frage, ob es nicht einfachere und kompaktere Darstellungsweisen für Problemlösungen gibt, die anschließend in Programmcode einer tieferen Abstraktionsebene umgewandelt werden können.

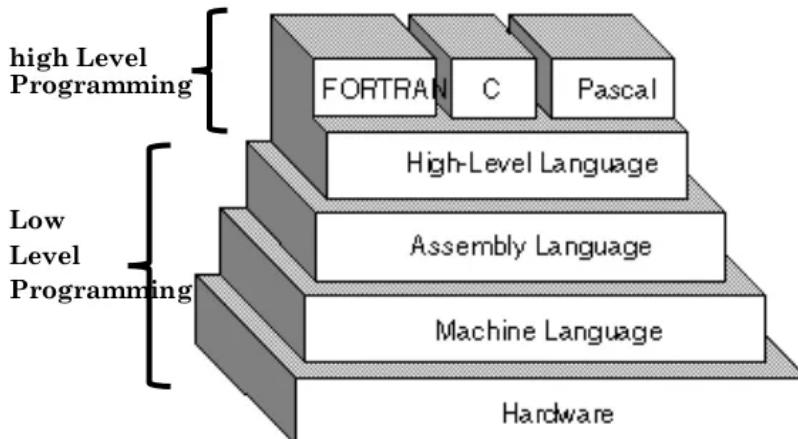


Abb. 3: Abstraktionsebenen von Programmiersprachen [Th21]

Ein **Compiler** ist wohl das bekannteste Werkzeug zur Codegenerierung. Dieser wandelt Quellcode einer höheren Programmiersprache in Befehle einer niedrigeren Sprache um. Ein Compiler für die Programmiersprache Java besteht aus verschiedenen Bestandteilen: Ein Scanner liest Lexeme und generiert Tokens. Der Parser generiert eine abstrakte Syntax. Im Semantik-Check wird eine getypte abstrakte Syntax generiert. Zuletzt generiert ein Code-Generator Bytecode für die JVM. Compiler sind ein fester Bestandteil von nahezu jedem Softwareentwicklungsprozess und werden aus diesem Grund der Vollständigkeit halber hier angemerkt, jedoch nicht näher behandelt.

Modellgetriebene Softwareentwicklung (MDSD) ist ein Oberbegriff für Techniken, die aus formalen Modellen automatisiert lauffähige Software erzeugen. [St07, p. 11]

Die formalen Modelle werden meist im zugeschnittenen Modellierungssprachen wie DLS oder UML spezifiziert. Formal bedeutet in diesem Zusammenhang, dass das Modell

einen bestimmten Aspekt der Software vollständig beschreibt. Dies ist essentiell für die anschließend automatische Codeerzeugung. Ein wichtiger Begriff in diesem Kontext ist das Round-Trip Engineering. Dies beschreibt die bidirektionale Synchronisation zwischen Modellen und Quellcode. So können die Vorteile der Modellentwicklung (Übersichtlichkeit und Dokumentation) mit den der Quellcodeprogrammierung (Präzision und Spezifikation) kombiniert werden.

Die Vorteile einer modellgetriebenen Softwareentwicklung sind [St07, p. 13-16]:

- (a) **Abstraktion:** Die Modelle sind einfacher und allgemeiner als der zu generierende Quellcode.
- (b) **Einheitliche Architektur:** Die Softwareerzeugung aus den Modellen erfolgt nach streng formalen Vorschriften unter Berücksichtigung eines vorgegebenen Rahmens.
- (c) **Softwarequalität** Das Projekt wird durch den modellgetriebenen Entwicklungsprozess in eine einheitliche, testbare und dokumentierte Architektur gegossen. Selbstverständlich ist dies dennoch keine Garantie für gute Softwarequalität.
- (d) **Entwicklungsgeschwindigkeit:** Durch eine höhere Softwarequalität ist das Projekt besser Wartbar und Komponenten sind besser Austauschbar, was langfristig die Entwicklungsgeschwindigkeit erhöht. Zudem gibt es immer eine Designdokumentation, was die Übersichtlichkeit erhöht.
- (e) **Interoperabilität und Plattformunabhängigkeit:** Durch die modellgetriebene Entwicklung soll eine Plattform- und Frameworkunabhängigkeit durch Standardisierung erreicht werden. Die Modelle sollen streng vom generierten Programmcode entkoppelt und somit von diesem unabhängig sein. Dies ist jedoch meist nur in der Theorie möglich und kann beispielsweise beim Round-Trip Engineering nicht gewährleistet werden.

Die Nachteile der modellgetriebenen Entwicklung sind:

- (a) **Hoher Initialisierungsaufwand:** Der Initialisierungsaufwand ist sehr hoch und zahlt sich entsprechend nur bei großen Projekten aus.
- (b) **Hoher Einarbeitungsaufwand:** Da die modellgetriebene Entwicklung nicht so verbreitet ist wie die Verwendung herkömmlicher Programmiersprachen, jedoch eine hohe Komplexität aufweist, ist ein hoher Einarbeitungsaufwand notwendig.

Werkzeuge für die modellgetriebene Entwicklung fangen bei einfachen UML-Codegeneratoren (z.B. Visual Paradigm) an und hören bei Komplettsystemen für bestimmte Domänen (z.B. ASCET für eingebettete Automobilsoftware) auf [St07].

Durch AI-basierte Werkzeuge gibt es die Möglichkeit zur **Quellcodeerzeugung aus natürlicher Sprache**. Dies stellt sich vor allem für stark spezialisierte Bereiche als nützlich

heraus. Ein Beispiel hierfür ist die Generierung von SQL Statements oder einfachen Programmieranweisungen aus natürlicher Sprache.

Eine populäre Möglichkeit hierfür ist das AI-Modell OpenAI Codex. Dieses stellt auch die Basis für das Werkzeug GitHub Copilot dar, welches gemeinsam von OpenAI und GitHub entwickelt wurde [Op21]. Dieses Modell kann über die API oder dem Browser-Playground von OpenAI genutzt werden. In einem Test wurde ein Ratespiel in JavaScript mittels dem OpenAI Codex Model generiert (Quellcode 1). Aus dem in den Kommentaren angegebene Text in natürlicher Sprache wurde jeweils der darunter stehende Quellcode generiert.

```

1  /* Generiere eine zufällige Zahl zwischen 0 und 10 */
2  var randomNumber = Math.floor(Math.random() * 10);
3
4
5  /* Frage den Nutzer nach einer Texteingabe */
6  var userInput = prompt('Gib eine Zahl zwischen 0 und 10 ein');
7
8  /* konvertiere die Nutzereingabe in eine Zahl */
9  var userNumber = parseInt(userInput);
10
11 /* wenn die generierte Zufallszahl gleich der userNumber ist "gewonnen", sonst "verloren" ausgeben */
12 if (randomNumber === userNumber) {
13     alert('gewonnen');
14 } else {
15     alert('verloren');
16 }
```

Quellcode 1: Code aus natürlicher Sprache

Auch die Generierung von SQL Statements liefert ein korrektes Ergebnis (Quellcode 2).

```

1  ### Postgres SQL tables, with their properties:
2  #
3  # Mitarbeiter(mid, vorname, nachname)
4  # Teilnahme(mid, pid)
5  # Projekt(pid, bezeichnung)
6  #
7  ### Liste die Mitarbeiternamen der Mitarbeiter des Projektes "Test" auf
8  -----
9  SELECT m.vorname, m.nachname
10 FROM Mitarbeiter m
11 JOIN Teilnahme t ON m.mid=t.mid
12 JOIN Projekt p ON t.pid=p.pid
13 WHERE p.bezeichnung='Test'
```

Quellcode 2: SQL Statement aus natürlicher Sprache

Selbst etwas komplexere Aufgaben, wie API-Aufrufe und Deserialisierung sind möglich. Hierzu musste jedoch mit der textuellen Eingabe etwas experimentiert werden (Quellcode 3) [Op22].

```
1 //API aufrufen: https://jsonplaceholder.typicode.com/todos/1
2 //Ergebnisse Deserialisieren & Ausgeben
3 //Quellcode in C#:
4 -----
5 using System;
6 using System.IO;
7 using System.Net;
8 using System.Runtime.Serialization.Json;
9
10 namespace ConsoleApp1
11 {
12     class Program
13     {
14         static void Main(string[] args)
15         {
16             string url = @"https://jsonplaceholder.typicode.com/todos/1";
17             WebRequest req = WebRequest.Create(url);
18             req.Method = "GET";
19             WebResponse resp = req.GetResponse();
20             var json = new DataContractJsonSerializer(typeof(Todo));
21             var todo = (Todo)json.ReadObject(resp.GetResponseStream());
22             Console.WriteLine(todo);
23         }
24     }
25
26     [DataContract]
27     public class Todo
28     {
29         [DataMember]
30         public bool completed { get; set; }
31         [DataMember]
32         public int userId { get; set; }
33         [DataMember]
34         public int id { get; set; }
35         [DataMember]
36         public string title { get; set; }
37
38         public override string ToString()
39         {
40             return $"{completed} {id} {title}";
41         }
42     }
43 }
44 */
```

Quellcode 3: API Aufruf und Deserialisierung aus natürlicher Sprache

Nüchtern betrachtet bieten solche Modelle dem Entwickler jedoch kaum eine Zeitsparnis und die Ergebnisse müssten zudem stets manuell auf Korrektheit überprüft werden. Als unterstützendes Werkzeug können solche Modelle jedoch nützlich sein. Wenn der Entwickler beispielsweise nicht mit einer Programmiersprache vertraut ist, können Befehle (wie z.B. die Ein- und Ausgabe) mittels natürlicher Sprache umschrieben werden. Für die Erstellung komplexer Software sind solche Generierungsverfahren jedoch ungeeignet, da die Umschreibung mit natürlicher Sprache meist zu unscharf und inkonsistent ist [Ch21].

Weitere Möglichkeiten, die jedoch nur gewisse Spezialgebiete der Softwareentwicklung abdecken sind folgende [Do08]: Annotationen, Präprozessoranweisungen, O/R-Mapper, Interface-Definition-Languages oder Codeerzeugung durch grafischen Oberflächendesigner.

3.2 Vorteile und Grenzen

Codegenerierung kann in der Softwareentwicklung unterschiedlich eingesetzt werden und bestimmte Vorgänge erleichtern (wie bereits in den Unterabschnitten beschrieben). Während die modellgetriebene Softwareentwicklung ein gesamtes Paradigma darstellt, sind Werkzeuge wie OpenAI Codex nur für bestimmte Randgebiete sinnvoll. Unabhängig vom verwendeten Codegenerator gilt weiterhin das bekannte Garbage In, Garbage Out Prinzip der Informatik. Darüber hinaus muss für jeden Anwendungsfall abgeschätzt werden, ob das verwendete Werkzeug einen Zeit-, Qualitäts oder Produktivitätsgewinn darstellt oder eher hinderlich ist. Zu erwähnen ist außerdem, dass diese Entwicklung noch lange nicht abgeschlossen ist. Es bleibt abzuwarten was bessere Modelle und immer mehr Beispieldatensätze in Zukunft ermöglichen werden.

4 Analyse

Wurden Teile eines Programms programmiert, ist es wichtig dessen Funktion und Leistung zu überprüfen. Neben der Identifikation von Problemen geht es dabei vor allem um Optimierungen. Dieser Abschnitt beschäftigt sich mit automatisierten Verfahren zur Analyse von Software. Zunächst werden die allgemeinen Konzepte statische und dynamische Analyse erklärt. Darauffolgend werden die Vorteile und Grenzen dieser Verfahren erläutert. Abschließend wird ein Werkzeug zur Analyse von Software vorgestellt.

4.1 Allgemeine Konzepte

Die **statische Quellcodeanalyse** ist ein Verfahren, welches Quellcode unabhängig von der Kompilierung oder Ausführung nach verschiedenen Kriterien auswertet. Dies soll Fehler, Inkonsistenzen und Unsicherheiten im Programmcode aufdecken. Statisch bedeutet in diesem Kontext, dass der Code nicht ausgeführt, jedoch unter anderem auch semantisch ausgewertet wird. Verwendete Verfahren sind [Fo08, p. 12]:

- (a) **Taint Analyse:** Analyse zur Verhinderung von böswilligen Benutzereingaben, die Code auf dem Hostcomputer ausführen (z.B. SQL Injection)
- (b) **Datenfluss Analyse:** Analyse, welche Daten zwischen Programmteilen ausgetauscht werden und welche Abhängigkeiten daraus entstehen.
- (c) **Kontrollfluss Analyse:** Analyse zur Evaluation und Integritätsprüfung des Programmablaufes mit dem Ziel der Feststellung von Anomalien (z.B. Endlosschleifen)
- (d) **Lexikalische Analyse:** Syntaktische Prüfung des Quellcodes und Generierung von Tokens

Historisch ist das Programmierwerkzeug Lint der Pionier der statischen Codeanalyse. Ursprünglich wurde Lint für die Programmiersprache C entwickelt. Nach und nach wurden Abwandlungen für andere Programmiersprachen, darunter beispielsweise JavaScript, TypeScript und Python, entwickelt. Die Entwicklung von Lint lief synergetisch zu der Entwicklung moderner Compiler [Da88, p. 2]. Ursprünglich sind Lint Programme so gedacht, dass diese vom Benutzer explizit ausgeführt werden müssen und anschließend den Quellcode analysieren. In modernen IDE sind Lint-ähnliche Verfahren meist automatisch integriert. Diese analysieren kontinuierlich den Quellcode und zeigen mögliche Probleme direkt an (Abbildung 4). Zudem werden solche Analysewerkzeuge als zusätzliche Plugins für viele IDEs angeboten [Da88][Wi22a].

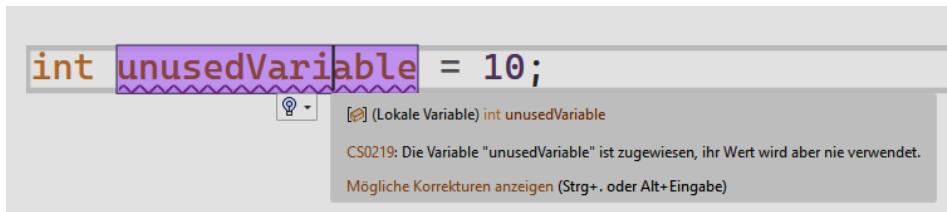


Abb. 4: Statische Codeanalyse in Visual Studio

Die Möglichkeiten der statischen Codeanalyse lassen sich in die Kategorien *Stylechecking*, *Semantische Analyse* und *Weiteres* unterteilen.

Bei der Softwareentwicklung wird häufig im voraus ein Programmierstil, auch oft Coding Conventions genannt, festgelegt. Dieser baut auf den üblichen Vorgehensweisen des verwendeten Programmierparadigma auf. Die meisten Programmiersprachen bieten ein Katalog von solchen Codierungsrichtlinien an. So existiert beispielsweise für die objekt-orientierte Sprache Java ein Dokument mit einer Empfehlung für einen zu verwendenden Programmierstil [Or97].

Für Projekte ist es gegebenenfalls sinnvoll die Codierungsrichtlinien der verwendeten Programmiersprache anzupassen und zu erweitern. Beispiele für Codierungsrichtlinien sind die Anwendung von Entwurfsmustern, die Festlegung der Namenskonventionen, der Umfang der Dokumentation oder die Gestaltung von Funktionsaufrufen.

Ein **Style Checker** ist ein Werkzeug, welches die Einhaltung des definierten Programmierstils überprüft. Dies ermöglicht es beispielsweise Verstöße gegen Benennungskonventionen oder einer Überschreitung der maximal zulässigen Zeilenlänge von Code zu erkennen. Style Checker bilden jedoch nur eine Teilmenge davon ab, was mit statischer Codeanalyse möglich ist, da meistens keine semantische Quellcodeanalyse vorgenommen wird. So kann beispielsweise nicht erklärt werden, dass in Quellcode 4 eine Endlosschleife produziert wurde [gm12][Wi22d].

```

1  while (1<10)
2  {
3  }
```

Quellcode 4: Endlosschleife

In der **semantischen Analyse** wird der Quellcode auf inhaltliche Konsistenz und mögliche Fehler geprüft. So können beispielsweise folgende Probleme erkannt werden: Memory Leaks, Pufferüberläufe, Divisionen durch null, Array Zugriffe außerhalb der Grenzen, Endlosschleifen, mögliche Nullverweise, nicht verwendete Variablen und Methoden, Anti-Patterns, mögliche Code Injections, Substitutionen und Generalisierungen oder toter Quellcode. Diese Art der Analyse ist besonders wichtig, da sie folgenschwere Fehler aufdecken soll.

Wird ein objektorientiertes Programmierparadigma verwendet besteht eine **weitere Möglichkeit** in der Bestrebung durch statische Quellcodeanalysen das Design zu bewerten. Dies ist natürlich nicht vollumfänglich möglich, jedoch gibt es bestimmte statisch ermittelbare Metriken, die auf ein gutes oder schlechtes Design schließen lassen. Diese lassen sich aus den Prinzipien für ein agiles, objektorientiertes Design ableiten [Ma02]. Folgende Metrischen sind hierfür denkbar und teilweise auch schon in statischen Analysewerkzeugen implementiert [Fo08][Da88][Jeb][Wi22c]:

- (a) Lines of Code (Single Responsibility)
 - Innerhalb einer Komponente, Komponente oder Methode
- (b) Ringabhängigkeiten
- (c) Coderedundanzen (Don't repeat yourself, Abstraction)
- (d) Große Vererbungsstrukturen (Composition over inheritance)
- (e) Abhängigkeiten auf Implementierungen statt auf Interfaces (Dependency Inversion, Open-Closed)
- (f) Größe der Interfaces (Interface Segregation)
- (g) Fehlende Implementierungen für Basismethoden (Liskov Substitution)

Bei der **dynamischen Analyse** liegt der Fokus auf Bereichen, die nicht von der statischen Quellcodeanalyse abgedeckt werden können. Dies sind primär Fehler in der Anwendungslogik, fehlende Sicherheitsvalidierung und Performance.

Wesentlich wird die dynamische Analyse von sogenannten Profiling Werkzeugen abgewickelt. Diese können verschiedene Messwerte zu der entwickelten Software ermitteln. Dazu gehört die Anzahl der Funktionsaufrufe und Funktionsdurchläufe, die Speicherauslastung, nicht freigegebenen Speicherbereiche, nebenläufige Prozesse und Deadlocks. Durch statistische Analysen kann herausgefunden werden, welche Programmteile lohnenswert optimierbar sind, um eine bessere Leistungs- und Speicherperformance zu erhalten. Auch Fehler in der Anwendungslogik, die sich in Deadlocks oder Memoryleaks manifestieren, können aufgedeckt werden [Pa19][Wi22b][No22][Sc19a].

4.2 Vorteile und Grenzen

Durch Automatisierte Analyseverfahren können tausende Zeilen Quellcode in kürzester Zeit mit einer sehr hohen Präzision durchsucht und ausgewertet werden. Mögliche Problemstellen werden unmittelbar angezeigt und können meist auch zur Quelle zurückverfolgt werden (z.B. nicht validierte Parameter). Über Konfigurationen können die Codeconventions festgelegt werden und Antipatterns definiert werden, die anschließend bei der statischen Codeanalyse überprüft werden. So kann auch in großen Projekten eine Einheitlichkeit und Stabilität im Quellcode erreicht und somit die Softwarequalität erhöht werden. Häufig auftretende Probleme oder Muster, die zu Schwierigkeiten im Entwicklungsprozess führen können, sind in den Analysewerkzeugen bereits vorkonfiguriert. Intellegente Analysewerkzeuge bieten zudem die Möglichkeit einer automatischen Problembehebung. Hier wird der Quellcode automatisch refaktoriert (Abschnitt 5) oder abgeändert.

Auch wenn Analysewerkzeuge eine große Hilfe für Softwareentwickler darstellen, ist es wichtig die Grenzen dieser zu kennen. Analysewerkzeuge sind nur so gut, wie sie programmiert und konfiguriert wurden. Sind bestimmte Antipatterns oder Konstrukte für unsauberer Code nicht in der Konfiguration und Programmierung berücksichtigt, werden diese auch nicht erkannt. Zudem lassen sich auch technisch nicht alle Problemfelder ermitteln. So können Logikfehler in komplexen Geschäftsprozessen häufig nicht identifiziert werden, da das Analysewerkzeug kein Verständnis hierfür besitzt. Auch bei Frameworkspezifischen Anwendungsfeldern können Probleme auftreten, da die Analysewerkzeuge auf die Framework Version abgestimmt sein müssen. Werden spezielle Techniken verwendet, wie z.B. Dependency Injection, die bei der Implementierung des Werkzeuges nicht berücksichtigt sind, kann es ebenfalls zu Schwierigkeiten kommen. So kann es in diesem Fall sein, dass Abhängigkeiten nicht mehr nachvollzogen werden können, weil keine klassische Referenzherzeugung und Übergabe mehr stattfindet. Sprachen die ständig weiter entwickelt werden, erzwingen ebenfalls eine kontinuierliche Weiterentwicklung der zugehörigen Analysewerkzeuge [Fo08][Wi22c][Sc19a].

4.3 Beispiel Werkzeug

Als eine Suite von Beispielwerkzeugen werden die Produkte des Softwareherstellers JetBrains gewählt. Die Werkzeuge sind entweder in die IDEs, die von JetBrains angeboten werden integriert, oder als separates Plugin für Visual Studio oder den Standalonebetrieb verfügbar. Angeboten werden:

- (a) ReSharper (Statische Codeanalyse, Refactoring)
- (b) DotTrace (Dynamische Analyse, Leistungsprofiling)
- (c) DotMemory (Dynamische Analyse, Memoryprofiling)
- (d) DotCover (Statische und dynamische Analyse, Testabdeckung, Unittesting)

JetBrains bietet elf IDEs für verschiedene Programmiersprachen an, die alle auf einem gemeinsamen Framework basieren [Je20]. Somit werden die Sprachen Objective C, Swift, PHP, Python, C#, Visual Basic, Go, Rust, Ruby, Kotlin, JavaScript und Java unterstützt [Jea].

5 Refactoring

Nach einer Analyse stehen in der Regel Änderungen im Code an. *Refactoring*, oder zu Deutsch *Refaktorierung*, beschreibt die nachträgliche Veränderung von Quellcode in Software. Konkret wird unter Beibehaltung des Verhaltens eines Programms eine verbesserte Struktur erzielt. Es ist also abgegrenzt von der Entwicklung neuer Funktionalitäten. Dies sollte vor allem bei großen Projekten ständiger Bestandteil sein. Schlechter Code erzeugt sonst immer größere Probleme und senkt somit auf Dauer die Produktivität [Ma09, S. 28]. Stattdessen sollte sauberer Code angestrebt werden. Dies bezieht sich sowohl auf Softwarearchitektur, als auch generelle Lesbarkeit von Code. Clean Code wird zum einen durch die im vorherigen Kapitel (Abschnitt 4) genannten Kriterien, oder auch die ISO-Normen, auf höherer Ebene analysiert. Eine Sicht auf niedrigerer Ebene bietet Grady Booch, Autor von *Object-Oriented Analysis and Design with Application*:

Sauberer Code ist einfach und direkt. Sauberer Code liest sich wie wohlgeschriebene Prosa. Sauberer Code verdunkelt niemals die Absicht des Designers, sondern ist voller griffiger (engl. crisp) Abstraktionen und geradliniger Kontrollstrukturen. [Ma09, S. 34]

Durchgehend sauberer Code ist zum Beispiel aufgrund von sich verändernden Anforderungen schwer zu erreichen. Daher ist Refactoring ein derart wichtiger Prozess. Da Refactoring aber auch bedeutet, dass in dieser Zeit keine neuen Funktionen entwickelt werden, ist die Balance sehr wichtig.

5.1 Betroffene Stellen

Betroffen sind zum Einen die Ergebnisse von Analysen, welche Probleme aufgezeigt haben. Diese wurde im vorherigen Kapitel (siehe Abschnitt 4 erläutert). Weitere unsaubere Stellen im Code werden auch "Code-Smell" genannt [Fo00, S. 67]. Martin Fowler kategorisiert diese in 22 Arten der Probleme [Fo00, S. 67 - S. 82]. Diese können teilweise in der Analyse erkannt werden, erfordern aber oftmals auch manuelle Überprüfung. Im Folgenden sind die wichtigsten zusammengefasst und erklärt:

- (i) **Redundanter Code** verringert die Änderbarkeit, sorgt für doppelte Entwicklung und großen Overhead.
- (ii) **Große Methoden, Parameterlisten oder Klassen** sind schwer verständlich und weniger Wiederverwendbar. Weitere mögliche Probleme sind schlechte Testbarkeit und Verletzung des Single Responsibility Prinzips.
- (iii) **Hinzufügen von nicht zugehörigen Funktionalitäten zu einer Klasse** verletzt das *Single Responsibility*-Prinzip und verschlechtert somit die Wartbarkeit. Gleiches gilt für das Gegenteil, also die Aufteilung einer Funktionalität auf verschiedene Klassen. Jede Klasse sollte also eine Verantwortlichkeit haben.
- (iv) **Unübersichtliche Gruppen aus zusammengehörigen Parametern anstelle der Nutzung eines Objektes** schaden der Lesbarkeit und sorgen für geringere Wartbarkeit. Änderungen müssen immer an allen Stellen, statt nur in der definierten Klasse erfolgen.
- (v) **Verschachtelte Switch-Befehle** behindern die Lesbarkeit. Je mehr Ebenen im Code sind, desto unübersichtlicher wird es.
- (vi) **Nachrichtenketten** schaden der Performance durch zu viele Zwischenaufrufe.
- (vii) **Klassen ohne eigene, nicht-triviale Verantwortlichkeit** schaden der Wartbarkeit und deuten auf schlechte Architektur hin.
- (viii) **Unangebrachte Abhängigkeiten von Details statt Schnittstellen** sorgen für schlechte Austauschbarkeit. Gleiches gilt für funktional gleiche Klassen, welche unterschiedliche Schnittstellen anstelle einer gemeinsamen besitzen.
- (ix) **Übermäßige Kommentare** implizieren eine Unverständlichkeit des Codes und schaden der Lesbarkeit.
- (x) **Nichtssagende Namen** sorgen für schlechte Lesbarkeit und Problemen bei späteren Änderungen.

Es könnten viele weitere genannt werden, genauere Details bietet auch Robert Martin in seinem Buch *Clean Code* [Ma09].

5.2 Allgemeine Konzepte zur Lösung

In seinem Buch *Refactoring* stellt Martin Fowler einen großen Katalog von möglichen Refactorings auf [Fo00, S. 99 - 387]. Im Folgenden findet eine Betrachtung einiger ausgewählter Refactorings statt, welche automatisch von Software umgesetzt werden können. Viele weitere sind manuell durchführbar, aber nicht Teil dieser Arbeit.

- (a) **Extrahieren oder Verschieben von Methoden und Klassen:** Ist eine Methode zu lange (siehe i), kann ein Teil von ihr ausgewählt und in eine eigene Methode extrahiert werden. Die Software erkennt Rückgabewert und Parameter automatisch, nur der Methodenname muss gewählt werden. Problematisch zur Auslagerung können dabei die lokalen Variablen sein. Diese müssen oftmals von Hand angepasst werden, zum Beispiel durch die Aufteilung in kleinere Variablen. Klassen dagegen werden oftmals in einer anderen Klassendatei definiert. Software ermöglicht das Verschieben in eine eigene Datei, was der Übersichtlichkeit dient.
- (b) **Vereinfachen von Ausdrücken und Statements:** Aufgrund vorheriger Analysen wurden mögliche Vereinfachungen im Code erkannt (Unterabschnitt 4.3). Die Software macht daraufhin Vorschläge zur Umwandlung. Dies kann beispielsweise die Invertierung einer If-Abfrage sein, welche für höhere Übersichtlichkeit sorgt. Oder auch die Umwandlung einer Verkettung von If-Abfragen in ein Switch-Statement. Ein anderes Beispiel wäre die Vereinfachung eines bedingten Ausdruckes, da die gleiche Logik in kompakterer Form erreicht werden kann.
- (c) **Umbenennung von Bezeichnern:** Aussagekräftige Bezeichner sind besonders wichtig (siehe x), da Quellcode sich selbst erklären soll. Software ermöglicht das Umbenennen eines Bezeichners an all seinen Vorkommen. Dies erspart Arbeit und die Gefahr, nicht alle Verwendungen zu finden. Oftmals weißt ein Stylechecker (Abbildung 4.1) auf problematische Bezeichner hin und bietet Gegenvorschläge.
- (d) **Migration von Datentypen:** Wird einer Variable oder einem Feld ein falscher Typ zugewiesen, muss dieser migriert bzw. gepräst oder der Datentyp des Felder verändert werden. Software kann diese Lösungen automatisch vornehmen.
- (e) **Einführung von Feldern oder Variablen:** Oftmals findet eine hohe Verschachtlung statt, was ein Zeichen für Code-Smell ist (siehe v). In der Regel ist eine Extraktion in erklärende Variablen sinnvoll. Dies kann durch Software erledigt werden, nur die Auswahl des Namens muss stattfinden.
- (f) **Parameter ergänzen, entfernen oder überladen:** Wird eine Methode aufgerufen mit anderen Parametertypen, als aktuell definiert, gibt es verschiedene Möglichkeiten. Entweder wird die Parameterliste verändert oder es muss eine Überladung der Methode mit passenden Parametern erzeugt werden. Dies kann eine Software automatisch übernehmen.

- (g) **Schnittstelle implementieren:** Implementiert eine Klasse ein Interface, muss es alle dessen Methoden überschreiben. Software ermöglicht das Einfügen aller Methoden des Interfaces mit leerem Methodenkörper.
- (h) **Attribute kapseln:** Ein Kernprinzip der objektorientierten Programmierung ist die Datenkapselung über Getter und Setter. Diese können ausgehend von einem Feld automatisch erzeugt werden durch Software.
- (i) **Maximale Abstraktion verwenden:** Sofern es möglich ist, sollte immer maximale Abstraktion und minimales Detail verwendet werden (siehe viii). Wird daher in der Analyse erkannt, dass eine höhere Abstraktion verwendbar ist, kann der Objekttyp automatisch ausgetauscht werden.
- (j) **Verwenden und Austausch von Modifizierern:** Verschiedene Modifizierer für Variablen können Vorteile bieten, so beispielsweise wenn ein String als Konstante definiert wird. Wurde in der Analyse erkannt, dass dies möglich ist, kann der Modifizierer durch die Software automatisch eingefügt werden. Gleiches gilt bezüglich Sichtbarkeit. Ist eine höhere Kapselung möglich, also beispielsweise protected statt public, kann die Änderung automatisch vollzogen werden.
- (k) **Formatierung:** Neben Architektur und Benennungen ist vor allem die Formatierung des Codes ausschlaggebend für Übersichtlichkeit. Software bietet hier die Möglichkeit, diese nach einem definierten Schema vorzunehmen. Es handelt sich um ein sogenanntes *Beautify* des Codes.
- (l) **Entfernung von totem Code:** Wird in der Analyse Code erkannt, der nie erreicht wird, kann dieser durch Software automatisch entfernt werden. Somit wird toter Code verhindert.
- (m) **Nutzung von besseren Sprachfeatures:** Viele Programmiersprachen entwickeln sich ständig weiter. Neue Features sind dem Programmierer aber oftmals unbekannt oder ihr Nutzen nicht verständlich. Wird bei der Analyse eine Stelle erkannt, welche besser durch ein neues Sprachfeature ersetzbar ist, kann Software dies automatisch umwandeln. Gleiches gilt für veraltete Features, welche verwendet werden. Beispielsweise wäre dies die Nutzung eines Switch anstelle von If-Else.

Die vorgestellten Konzepte können je nach Refactoring-Tool um weitere Fähigkeiten ergänzt sein.

5.3 Nutzen und Risiken

Der hauptsächlichen Vor- und Nachteile von Refactoring allgemein wurde zu Beginn des Kapitels bereits erläutert. Im Folgenden soll dies in Bezug auf den Einsatz von Software zum Refactoring analysiert werden.

Hauptsächlicher Vorteil ist die Einsparung von trivialer und redundanter Arbeit. Alle durch die Software automatisch erledigten Aufgaben, welche nach der Analyse anstehen, könnten auch manuell umgesetzt werden. Dazu wäre aber ein Vielfaches an Arbeit notwendig. Der Entwickler spart sich stattdessen viele Klicks und Tastenanschläge (siehe a-m), sowie Recherche und Denkarbeit zur genauen Umsetzung einer Aktion (siehe b, i, j, m). Außerdem sorgt die Umsetzung nach klaren Regeln dafür, dass weniger Fehler geschehen. Die manuelle Umwandlung birgt immer das Risiko von Denk- oder Schreibfehlern, sowie Unvollständigkeit (siehe beispielsweise a-c). Die Software garantiert, dass alle betroffenen Stellen identifiziert und bearbeitet werden. Durch die einfache Umsetzung und höhere Sicherheit ist die Motivation zum Refactoring höher. Aus den Umwandlungen resultierende Vorteile können performanter (siehe j, m) und verständlicherer Code (siehe a-m) sein.

Refactoring-Software bringt jedoch auch Gefahren mit sich. Entwickler haben die Gefahr, sich zu sehr auf derartige Tools zu verlassen und viele Dinge nur noch mit ihrer Hilfe umsetzen zu können. Wenn beispielsweise verschiedene Datentypen unterschiedliche Performance erreichen und dies relevant für die Anwendung ist, wäre es falsch keine eigene Recherche durchzuführen und sich auf die Software zu verlassen. Diese zieht Performance gegebenenfalls einfach nicht in Betracht (beispielsweise bezogen auf b, d, i, m). Ein weiterer Punkt ist, dass Code nur aufgrund von noch fehlenden Implementierungen zum Zeitpunkt der Analyse Gründe zum refaktorieren hat, welche später hinfällig sind (beispielsweise durch i, j, l). Die Einfachheit des Refactoring verleitet hierbei ggf. zu verfrühten Aktionen. Dies bezieht sich beispielsweise auf die öffentliche Sichtbarkeit von Methoden einer Klasse, welche aktuell aber nur privat verwendet werden. Später haben diese jedoch eine öffentliche Verwendung. Derartige Software muss also vorsichtig genutzt werden, da großflächige Änderungen schnell durchgeführt sind.

Aktuelle Studien legen nahe, dass im Jahr 2021 die meisten Refactorings noch manuell und ohne Tools durchgeführt werden [EM21]. Grund dafür ist unter anderem fehlendes Vertrauen. Oftmals wird Code an vielen Stellen verändert und zum Nachvollziehen dieser Änderungen müssen Tools wie GitHub verwendet werden. Auch fehlt das Hintergrundwissen, wie genau eine Änderung vollzogen wird. Nicht zuletzt benötigen die Tools zumindest für komplexere Operationen auch eine Einarbeitungszeit. Es ist möglich, dass sich anfangs die Zeiteinsparung kaum lohnt, da der Weg zum Refactoring ohne genaue Kenntnisse zu lange ist [EM21].

5.4 Marktanalyse

ReSharper ist ein kostenpflichtiges Tool von JetBrains. Die Preise variieren je nach Paket, liegen aber etwa bei 350€ je Nutzer pro Jahr. Mit Laufzeit oder dem Preis größerer Pakete verringert sich der Preis für einzelne Tools. Es kann in C#, Visual Basic, XAML, ASP.NET, ASP.NET MVC, Python, JavaScript, TypeScript, Ruby und Rails, CSS, HTML, XML, Java, JSON, PHP und C++ verwendet werden. In einigen Sprachen ist *ReSharper* direkt in der IDE inkludiert (beispielsweise PyCharm oder IntelliJ), bei den restlichen Sprachen handelt

es sich um eine Erweiterung für Visual Studio. Die konkreten Features je Sprache sind auf der offiziellen Website zu finden [Re22c]. ReSharper bietet alle vorgestellten Konzepte zur Refaktorierung an, sowie viele weitere. Besonders mehr als 1200 sogenannte *Quick-Fixes* an analysierten Fehlern kann *ReShaper* durchführen. Dazu gehörigen beispielsweise folgende zuvor nicht aufgelistete:

- (i) Hinzufügen eines `return`-Statements oder Umwandeln des Rückgabetypes in `void`, sofern dieses fehlt.
- (ii) Passendes *escape* von sensiblen Zeichen im String, beispielsweise Backslash in einem Dateipfad.
- (iii) Korrektur oder Import nicht aufgelöster Symbole, beispielsweise durch ein zugehöriges Paketsystem.
- (iv) Konvertierung von Interfaces in abstrakte Klassen und umgekehrt.
- (v) Ersetzen eines klassischen Konstruktors durch das Pattern der Factory Methode.
- (vi) Benutzerdefinierte Aktionen, welche bei Erkennung von definierten Mustern vorschlagen werden.
- (vii) Und viele mehr, siehe Dokumentation [Re22a].

Abbildung 5 zeigt einiger der Konzepte (siehe c, h, j, m) am Beispiel C#-Code in Visual Studio auf. Eine genaue Auflistung aller Features ist auf der Website zu finden [Re22b].

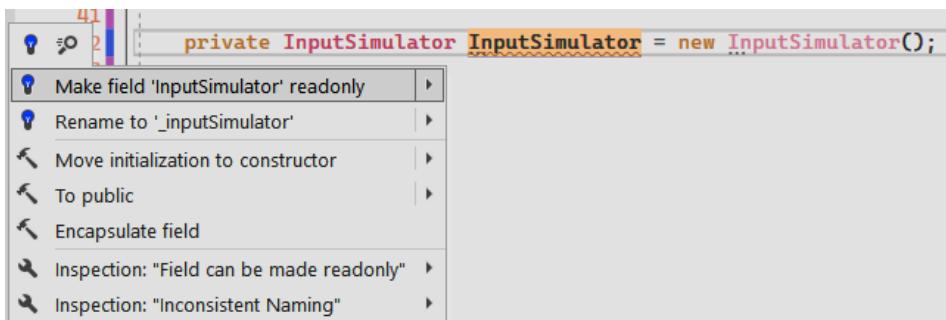


Abb. 5: Refactoring Vorschläge von ReSharper

Alternative Tools für eine derart große Anzahl an unterschiedlichen Sprachen und Funktionen sind aktuell keine auf dem Markt. Kostenlose Alternativen wie **CodeRush** beschränken sich zumeist auf wenige Sprachen [De22]. Allerdings bieten viele IDEs einige dieser Funktionen out of the box an und viele Entwickler arbeiten generell nur mit einer Programmiersprache. Daher können andere, für die jeweilige Programmiersprache geeignete, Tools verwendet werden.

6 Dokumentation

Um ein Projekt mit entwickeltem Code langfristig weiterentwickeln und warten zu können, ist die Dokumentation von diesem notwendig. Als *Softwaredokumentation* ist sämtliche Dokumentation einer Software im Entwicklungsprozess und darüber hinaus zu verstehen. Über den gesamten Entwicklungsprozess werden unterschiedlichste Dokumentationen erstellt. Sie dienen dazu, die Funktionalität, den Nutzen und die Entwicklung der Software für die verschiedenen Stakeholder nachvollziehbar zu machen. Dabei ist auch die Anzahl der Entwickler und die Größe des Projekts zu vernachlässigen. Eine gute Softwaredokumentation ist auch bei nur einem Entwickler essenziell und gewinnt bei größeren Kollaborationen nur noch mehr an Wichtigkeit. Die Softwaredokumentation muss alle wichtigen Fragen der mitwirkenden beantworten können und Aufschluss über die Software geben.

Documentation is highly valued, but often overlooked.

—opensourcesurvey

Eine großangelegte Umfrage von Open Source⁴ zeigt auf, dass eines der Hauptprobleme bei der Entwicklung freier Software eine unvollständige oder verwirrende Dokumentation ist. Vielen Entwicklern ist die Wichtigkeit der Dokumentation nicht bewusst. Doch Schlechte Dokumentation kostet Geld [Gi17]!

Die Dokumentation geht mit einer erfolgreichen Kollaboration in einem Entwicklerteam einher. Hierbei sind die verschiedenen Entwickler auf Informationen über Schnittstellen, Funktionen einzelner Module und den Änderungsverlauf im Projekt angewiesen. Für die automatische Dokumentation solcher Informationen stehen dem Entwickler verschiedene intelligente Werkzeuge zur Verfügung. Diese helfen bei der Erstellung der Dokumentationen und verbessern den Entwicklungsprozess. Neben der automatischen Generierung wird auch die Qualität und Konsistenz der Dokumente eingehalten und verbessert.

Entscheidend ist die Auswahl der Hilfsmittel. Es existiert eine Vielzahl an Hilfsmitteln für verschiedene Arten der Softwaredokumentationen. Bei der Auswahl der Hilfsmittel müssen verschiedene Kriterien, die der Entwicklungsprozess mit sich bringt, berücksichtigt werden.

6.1 Allgemeine Konzepte

In dem Prozess der Softwareentwicklung gibt es viele verschiedene Arten von Dokumentationen. Dabei sind nicht immer alle Arten von Dokumentationen notwendig. Dokumentationen können in verschiedene Bereiche und unterschiedliche Zielgruppen unterteilt werden. Je nach Vorgehen im Entwicklungsprozess und dem Vorhandensein der Zielgruppen, müssen die notwendigen Dokumentationen individuell festgelegt werden. Trotz einer Vielzahl

⁴ Softwaregruppierung zur Entwicklung freier Software. (Vgl. <https://opensource.com/>)

an unterschiedlichen Dokumentationsmöglichkeiten lassen sich zwei Hauptkategorien herausarbeiten.

Bei der **Projektdokumentation** wird der Fokus auf das Vorgehen, die Methodik, und die Werkzeuge gelegt. Sie ist für die verschiedenen Stakeholder und soll Aufschluss über den Verlauf des Projektes geben.

Die **Systemdokumentation** hingegen beschreibt das Produkt. Genauer beschreibt sie, aus was das Produkt besteht und wie es funktioniert und vorgeht. Diese Art der Dokumentation ist primär für den Entwickler. Gerade bei großen Teams oder späterer Weiterentwicklung ist diese Art der Dokumentation sehr wichtig.

Das Feld der **Projektdokumentation** bezieht sich wie beschrieben, auf den Verlauf und die Planung des Entwicklungsprozesses. Hierbei sind die Abläufe projektabhängig und werden Kunden-/Produktspezifisch angepasst. Intelligente Werkzeuge können aufgrund der Individualität von Projekten nur begrenzt eingesetzt werden. Die unterstützenden Werkzeuge, die hierbei verwendet werden, sind meist auf die allgemeinen Hilfsmittel einer Projektplanung zurückzuführen. Anders verhält sich dies jedoch im Bereich der **Systemdokumentation**. Dort unterstützen sie bei der Dokumentation des eigentlichen Umsetzungsprozess und bei der Beschreibung des Produkts.

Ein nützliches Werkzeug zur Dokumentation des Aufbaus einer Anwendung ist das **automatische Generieren von UML-Diagrammen**. UML-Diagramme sind ein wichtiger Bestandteil der Anwendungsdokumentation. Sie dienen in der Softwareentwicklung als ein Standard zur Visualisierung des Systementwurfs [Am04]. Die UML-Diagramme begleiten den ganzen Entwicklungsprozess der Software. Sie sind sowohl bei der Planung ein wichtiger Bestandteil, als auch bei der dynamischen Umsetzung der Software. Bei der Umsetzung kann der Aufbau in den verschiedenen Stadien visualisiert und so nachverfolgt werden. Die automatische Generierung der Diagramme an sich, ist schon ein wichtiges intelligentes Werkzeug für den Entwickler. Jedoch bieten UML-Diagramme noch mehr Möglichkeiten den Entwicklungsprozess zu erleichtern. So wie ein Diagramm aus dem Quelltext automatisch generiert werden kann, so ist auch Quelltext aus einem Diagramm generierbar. Das bietet dem Entwickler während der Umsetzung die Möglichkeit, Klassen und andere Typen grafisch zu erstellen, zu ändern und zu löschen [Te22].

Inline-Kommentare sind lesbare Kommentare innerhalb des Quelltextes. Sie werden dem Quelltext hinzugefügt, um ihn verständlicher und nachvollziehbarer zu machen. Dies ist wichtig, wenn mehrere Entwickler gemeinsam an einer Anwendung arbeiten. Es kann viele verschiedene Gründe geben, einen Quelltext-Abschnitt oder eine Zeile mit Kommentaren zu versehen. Folgend sind Gründe für Inline-Kommentare aufgelistet.

- (a) **Planen und Überprüfen:** Es kann vor dem Beginn der eigentlichen Programmierung anhand von Kommentaren die Umsetzung geplant werden. So kann an den konkreten Stellen, wie bspw. leeren Methoden, ein Kommentar mit der Beschreibung hinterlassen

werden. Anhand dieser Beschreibung übernimmt ein Entwickler dann die konkrete Implementierung (siehe Quellcode 5).

- (b) **Beschreibung des Quelltextes:** Wie im vorherigen Paragraphen beschrieben, ist es wichtig einen schnellen Überblick über den Quelltext zu erhalten. Gerade wenn mehrere Entwickler an dem selben Projekt arbeiten, ist eine Beschreibung am Beginn einer Passage und relevanten Stellen (Methoden, Klassen, ...) sehr hilfreich.
- (c) **Beschreibung des Algorithmus:** Noch wichtiger sind Kommentare bei unübersichtlichen Algorithmen. Auch wenn Quellcode eigentlich selbsterklärend sein soll, ist dies bei komplexen Algorithmen oftmals nicht machbar. Meist ist nicht auf den ersten Blick ersichtlich, was ein Algorithmus bewirkt. Daher kann eine vorhergehende Beschreibung und Kommentierung ausgewählter Stellen sinnvoll sein.
- (d) **Verwendung von Ressourcen:** Wird eine externe Ressource verwendet, ist es hilfreich Informationen über den Speicherort und die verwendete Version der Ressource zu hinterlassen. Auch die offiziellen Namen sind bei der Verwendung von Abkürzungen im Quelltext als Kommentare hilfreich.
- (e) **Debugging:** Beim Debugging ist es hilfreich, an bestimmten Stellen Markierungen zu setzen, um den Quelltext während des Debuggings übersichtlich zu halten.

Die Realisierung von Kommentaren ist in unterschiedlichen Programmiersprachen unterschiedlich gestaltet. Folgend ist eine Übersicht über einige der verwendeten Token zum Realisieren von Kommentaren dargestellt.

Symbol	Programmiersprache
REM	BASIC, Batch files
::	cmd.exe
#	Cobra, Perl, Python, Ruby, Make, Windows PowerShell, PHP
%	TeX, Prolog, MATLAB
//	C (C99), C++, C#, D, F#, Go, Java, JavaScript, Kotlin

Tab. 1: Tokens zu Beginn eines Kommentars in unterschiedlichen Programmiersprachen

Ein wichtiges Werkzeug, welches Inline-Kommentare ermöglicht und viele Entwicklungsumgebungen unterstützt, ist das verwenden von Tags. Hierbei handelt es sich um Begriffe, die einem Kommentar vorangestellt werden, sodass dieser kategorisierbar ist. Dieses Werkzeug erleichtert das Auffinden von bestimmten Stellen im Quelltext immens. Dabei gibt es verschiedene Tags für unterschiedliche Kategorien. Als Beispiele soll hier *TODO* genannt werden (Quellcode 5). Dieser Tag kann an Stellen in einem Kommentar erwähnt werden, an welchen es noch etwas zu implementieren gilt (siehe Quellcode 5). Es ist nun dem Entwickler die Möglichkeit gegeben, sich alle Kommentare mit diesem Tag aufzulisten zu lassen. Dadurch hat er eine schnelle Übersicht, an welchen Stellen Änderungen notwendig sind und kann automatisch zu diesen navigieren. Durch den Inhalt des Kommentars, ist auch auf einen Blick zu erkennen, was zu tun ist. Neben dem genannten *TODO* Schlüsselwort gibt

es noch weitere typische Tags, die folgend aufgelistet sind [Ji16]: *BUG*, *DEBUG*, *FIXME*, *TODO*, *UNDONE*.

```
1 //Konstruktor der Klasse
2 public Program(int a, int b){
3
4     //TODO: addiere a + b
5
6 }
```

Quellcode 5: Beispiel von Inline-Kommentaren

Eine weitere Möglichkeit welche Inline-Kommentare bieten, ist die automatische Generierung einer Dokumentation des Quelltextes. Durch Einhaltung einer bestimmten Syntax, kann im Nachgang eine vollständige Dokumentation mit Beschreibungen der Klassen, Methoden und Parameter erstellt werden. Diese wird in der Regel als HTML-Dokument erzeugt und kann somit einfach freigegeben werden. Dies kann mit der UML-Diagrammerzeugung verglichen werden, wobei es sich hier nur um textuelle Beschreibungen handelt. Es stehen mehr die einzelnen Aufgaben der Klassen und Methoden im Vordergrund, als deren Zusammenhang.

Als Syntax der Kommentare können unterschiedliche Formate vorgegeben sein. Je nach verwendetem Generator kann es auch eine individuelle Syntax und Semantik des Generators sein. Ein bekanntes Format was auch genutzt wird, ist das XML-Format. Mit diesen verschiedenen Standards lassen sich Beschreibungen zu Elementen und Kommentaren erstellen, mit denen anschließend automatisch eine Dokumentation generiert werden kann.

6.2 Marktanalyse

Bezüglich der **Erzeugung von UML-Diagrammen** sind viele verschiedene Werkzeuge auf dem Markt vertreten. In den meisten kommerziell genutzten Entwicklungsumgebungen ist ein UML-Generator standardmäßig vorhanden. Dies unterscheidet sich aber je nach Entwicklungsumgebung. In den bekannten Entwicklungsumgebungen Visual Studio von Microsoft (Abbildung 6) und IntelliJ von JetBrains sind UML-Generatoren vorhanden. Bei Visual Studio ist der UML-Generator auch in der kostenlosen Community Version enthalten.

Zudem unterstützt nur Visual Studio das Erstellen und Bearbeiten von Quelltext mit Hilfe der UML-Diagramme. Dies hat in eigens durchgeführten Versuchen sehr gut funktioniert. Es konnten Anwendungsstrukturen mit Hilfe der UML-Diagramme erstellt werden. Auch das nachträgliche Umbenennen von Klassen und Methoden hat hervorragend funktioniert [Te22][Je22].

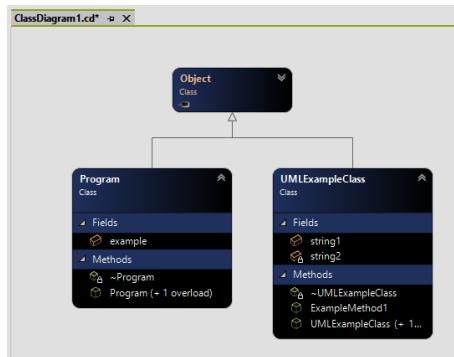


Abb. 6: Ein von Visual Studio generiertes UML-Diagramm

Ein Werkzeug, welches unabhängig von der Programmierumgebung arbeitet, ist Doxygen. Es handelt sich um das Standard Werkzeug für die automatische Dokumentationsgenerierung. Einige der unterstützten Programmiersprachen sind C++, C, Objective-C, C#, PHP, Java und Python [Do22].

Zur **Inline-Dokumentation** kann dagegen beispielhaft JavaDoc verwendet werden. Dieses ist in Intelij standardmäßig im Einsatz und verfügbar (siehe Quellcode 6).

```

1 /**
2 * Beschreibung der Methode
3 * @param integer1 Beschreibung Parameter 1
4 * @param integer2 Beschreibung Parameter 2
5 * @return Beschreibung des zu retunierenden Wert
6 */
7 public static String test(int integer1, int integer2){
8
9     return "foo";
10 }
```

Quellcode 6: Beispiel von Inline-Kommentaren

Die in Quellcode 6 verwendeten Kommentare erzeugen Abbildung 7.

```
test

public static String# test(int integer1,
                           int integer2)

Beschreibung der Methode

Parameters:
integer1 - Beschreibung Parameter 1
integer2 - Beschreibung Parameter 2

Returns:
Beschreibung des zu retunierenden Wert
```

Abb. 7: Automatisch generierte Übersicht einer Java-Methode Quellcode 6

Dennoch gibt es auch in diesem Feld weiter Anbieter, die eine Dokumentationserzeugung aus Inline-Kommentaren erlauben. Um den erwähnte XML-Standard aufzugreifen, ist hier Visual Studio zu erwähnen. In Visual Studio ist ein Bordmittel integriert, mit welchem sich die XML-Kommentare in eine HTML-Dokumentation wandeln lassen. Dazu gibt es verschiedene Schlüsselwörter, welche zur Beschreibung verwendet werden. So wird zum Beispiel das Schlüsselwort `<summary>` zum Definieren einer Beschreibung eines bestimmten Elements verwendet werden. Des Weiteren können Parameter mit Kommentaren und Anmerkungen versehen werden. Dies geschieht mit dem Schlüsselwort `<param name="nameDesParameters">Beschreibung des Parameters</param>`.

Beim Verwenden der zwei Werkzeuge ist schon zu erkennen, dass sich das Prinzip nur leicht unterscheidet. Die Syntax und Semantik, mit welchen die Kommentare anzugeben sind, unterscheiden sich zwar im Format, jedoch ähneln sich die erzeugten Dokumentationen im Stil und Aufbau sehr.

Doxygen, wie schon erwähnt (Unterabschnitt 6.2), begrenzt sich nicht nur auf die UML-Diagrammerzeugung, sondern bietet auch in diesem Feld eine Lösung. Es kombiniert die Felder und gibt in der erzeugten Dokumentation sowohl die UML-Diagramme, als auch die einzeln definierten Beschreibungen an. Hierbei setzt Doxygen auf eigens definierte Befehle. Es gibt Befehle der Form `\Befehl` welche die Art des Kommentars beschreiben. Aber auch die erzeugte Dokumentation von Doxygen ähnelt den erwähnten Werkzeugen sehr.

Eine Alternative, die für viele Programmiersprachen verwendet werden kann, ist Natural Docs. Natural Docs unterstützt 21 Programmiersprachen und ist folglich an keine Programmierumgebung gebunden. Dabei bindet sich Natural Docs wie Doxygen an keinen Standard, sondern definiert seine eigenen Befehle zum Kommentieren.

Als ein bekanntes Beispiel einer automatisch generierten Dokumentation, bietet Oracle eine

Dokumentation von Java⁵. Dabei sind die einzelne Methoden, Klassen und Schnittstellen von den verschiedenen Java-Versionen dokumentiert.

6.3 Nutzen und Risiken

Durch Werkzeuge zur Dokumentation werden Schritte des Prozesses automatisiert. Somit wird dieser weniger fehleranfällig und einfacher anzuwenden. Durch die nun einfache Dokumentation sind Entwickler eher geneigt, ihren Code tatsächlich zu dokumentieren. Somit wird beispielsweise bei der Verwendung von dokumentierten Methoden klar, was die jeweilige Methode im Detail macht und worauf ggf. geachtet werden muss. Dies ist besonders bei großen Projekten wichtig, in welchen aufgrund getrennter Entwicklung regelmäßig fremde Methoden verwendet werden. So wird direkt bei den Vervollständigungsvorschlägen (siehe Abschnitt 2) die Inline-Dokumentation angezeigt. Außerdem müssen Diagramme nicht weiterhin von Hand erzeugt werden, sondern sind einfach exportierbar, sowie importierbar.

Trotz aller Vorteile muss jedoch immer ein Mittelmaß gefunden werden. Gute Dokumentation ersetzt nicht *Clean Code*, also beispielsweise die Verwendung von sinnvollen Namen. Außerdem muss darauf geachtet werden, dass Dokumentation auch bei Änderungen im Code aktuell gehalten wird. Ansonsten unterscheidet sich die Beschreibung von der tatsächlichen Funktionalität. Nicht zuletzt sollten Kommentare kurz und verständlich anstelle von lang und unübersichtlich sein. Ansonsten wird unnötig viel Zeit darauf verwendet und es fehlt eine einfache, schnelle Verständlichkeit. Eine übertriebene Nutzung der Kommentarfunktion kann den Quellcode sogar weniger leserlich machen.

Bezüglich von Diagrammerzeugung muss außerdem erwähnt werden, dass gewisse Besonderheiten ggf. nicht automatisch in ein Diagramm übersetzt und somit manuell nachgebessert werden müssen. Erwähnenswert sind außerdem die möglicherweise anfallenden Kosten. Dies betrifft auch Werkzeuge zur Diagrammerzeugung aus Quelltexten wie in der Marktanalyse ersichtlich ist (Unterabschnitt 6.2).

7 Kollaboration

Ein weiteres wichtiges intelligentes Werkzeug, welches sich im Bereich der Dokumentation und Kollaboration befindet, ist die Versionsverwaltung. Diese ermöglicht während des gesamten Projektes die Zusammenarbeit mehrerer Personen, Versionierung, als auch Dokumentation der Änderungen. Große Projekte sind undenkbar ohne Versionsverwaltung.

⁵ Vgl.: <https://docs.oracle.com/javase/7/docs/api/overview-summary.html>

7.1 Allgemeines Konzept

Die Versionsverwaltung ist gerade bei Kollaborationen mehrerer Entwickler essenziell. Sie wird zum Erfassen von Änderungen an Dateien und Dokumenten verwendet. Eine Versionsverwaltung bietet einige Vorteile. Darunter zählen Transparenz, Übersicht, Nachverfolgbarkeit, Zusammenarbeit mehrerer Entwickler und Identifikation. All dies führt zu einer Effizienzsteigerung. Dabei muss es sich bei zu verwaltenden Daten nicht, wie weitläufig bekannt, nur um Quelltexte handeln. Eine Versionsverwaltung kann auch für Text- oder Tabellendokumente verwendet werden. Im Folgenden wird zwar überwiegend auf die Verwaltung von Quelltexten eingegangen, dennoch soll erwähnt werden, dass eine Versionsverwaltung in den verschiedensten Projektbereichen eingesetzt werden kann. Dies gibt den Entwicklern die Möglichkeit, Änderungen nachzuverfolgen und gegebenenfalls rückgängig zu machen. Des Weiteren lassen sich vielseitige Statistiken über die getätigten Änderungen erstellen. Weitere Hauptaufgaben einer Versionsverwaltung sind Protokollierung, Wiederherstellung, Archivierung, Koordinierung und das gleichzeitige entwickeln mehrere Zweige. Für die genannten Aufgaben verfügt jede Versionsverwaltung über fünf Basisaktionen, welche diese Aufgaben ermöglichen.

- (a) **Add:** Fügt Dateien zur Versionsverwaltung hinzu.
- (b) **Remove:** Entfernt Dateien aus der Versionsverwaltung.
- (c) **Commit:** Veröffentlicht vorgenommene Änderungen.
- (d) **Revert:** Setzt die aktuelle Version auf den letzten veröffentlichten Stand zurück.
- (e) **Branch:** Erzeugt einen neuen Zweig aus einem bestehend Zweig.
- (f) **Merge:** Fügt zwei Zweige zusammen und passt Differenzen an.

Darüber hinaus gibt es noch viele weitere Möglichkeiten, die ein Versionsverwaltungssystem besitzen kann. Die genannten Möglichkeiten sind die Grundlagen, die Versionsverwaltungssysteme beherrschen [DA20]. Die Möglichkeiten die dem Entwickler dadurch gegeben werden, sind im Entwicklungsprozess sehr wichtig und werden in den meisten Projekten eingesetzt. Auf Quelltexte bezogen, bietet die zentrale Lagerung der Quelltext-Dateien noch weitere Vorteile.

Wie in einem Verwaltungssystem üblich, können verschiedene Rollen vergeben werden. Diese ermöglichen die Vergabe von Freigaben und somit eine Rechteverwaltung. Dadurch ist die Möglichkeit gegeben, Änderungen erst nach einer Absprache und/oder Korrektur freizugeben. Eine weitere Hauptaufgabe, welche Versionsverwaltungssysteme ermöglichen ist die Integration spezieller Abläufe bei der Aktualisierung der Version. So können festgelegte Buildvorgänge, Sicherungen oder verschiedenen Tests der Anwendung durchgeführt werden.

7.2 Martkanalyse

Auf dem Markt gibt es unzählige Versionsverwaltungssysteme mit verschiedenen Schwerpunkten für verschiedensten Betriebssysteme. Darunter sind zum Beispiel *Git*, *Subversion (SVN)* und *Concurrent Version System (CVS)*. Das wohl bekannteste Versionsverwaltungssystem ist Git. Git unterscheidet sich insofern von anderen, als das es ein verteiltes System ist. Das bedeutet, dass jede Kopie des Verzeichnisses die gesamten Metadaten inklusive des Änderungsverlaufs enthält und so ein eigenständiges und abgekapseltes Verzeichnis bildet. Git ist kostenlos nutzbar und ein OpenSource-Projekt. Gleiches gilt für Subversion, welches wie git eine große Bekanntheit genießt und von der Apache Software Fundation entwickelt wird. Das Ziel von Subversion ist es, der Nachfolger des in der Vergangenheit sehr bekannten Concurrent Version System zu sein.

Mit 94 Millionen Nutzern ist GitHub⁶ ein sehr bekanntes, auf Git basierendes, Versionsverwaltungssystem im Internet. An diesem Beispiel ist sehr gut zu erkennen, welche weiteren Möglichkeiten eine Versionsverwaltung bietet. Neben Integration der oben genannten Standard-Werkzeugen einer Versionsverwaltung, bietet Github noch viele Erweiterungen rund um die Versionsverwaltung. So können Teams mit Dashboards und verschiedenen Statistiken zum Projekt arbeiten. Auch können verschiedenste Automationen eingepflegt werden, welche den Entwicklungsprozess erleichtern. Bei größerem Interesse kann bei einer Recherche das Stichwort CI/CD oder DevOps unterstützen.

7.3 Nutzen und Risiken

Durch die dadurch VCS entstehenden Möglichkeiten ist eine große Effektivitätssteigerung in Projekten möglich. Sie vereinfachen die Zusammenarbeit in großen Teams und Kollaborationen. Die Werkzeuge sind in den unterschiedlichsten Ausprägungen in vielen Firmen vertreten und werden von vielen Entwicklern schon als Standard angesehen. Zudem sorgen die Möglichkeiten zur Nutzung verschiedener Branches, das Mergen und die Dokumentation der Änderungen generell für eine höhere Sicherheit des Codes. Fehler können schnell identifiziert werden, eine lauffähige Version ist immer vorhanden und es kann unabhängig von anderen entwickelt werden.

Ein Risiko, welches die Werkzeuge dennoch bieten ist, dass sie über ihre Maße angewandt werden. Auch bei der Versionsverwaltung ist dies ein Thema. Viele Anbieter bieten über die Standardmöglichkeiten einer Versionsverwaltung hinaus Werkzeuge an. Diese sind jedoch nicht immer erforderlich. Ein weiterer Punkt ist, dass die meisten Werkzeuge für den kommerziellen Einsatz in Unternehmen kostenpflichtig sind. Die genannten Werkzeuge werden meist auf Unternehmensinterne Daten angewandt. Deshalb ist auch der Datenschutz ein sehr wichtiger Punkt, welcher nicht vernachlässigt werden sollte. Hierfür bieten die

⁶ <https://github.com/>

genannten Anbieter zwar Lösungen, wie zum Beispiel die Werkzeuge Unternehmensintern zu hosten, dennoch sollte der Datenschutz nicht vernachlässigt werden.

8 Schluss

Abschließend soll in einer kurzen Zusammenfassung die Arbeit zusammengefasst und die Erkenntnisse zentral gesammelt werden. Danach folgt mit den durch die Arbeit erlangten Erkenntnissen ein Ausblick in die Zukunft von intelligenten Werkzeugen zur Softwareentwicklung.

In der vorliegenden Arbeit wurden wichtige intelligente Werkzeuge zur Softwareentwicklung aufgezeigt. Dazu wurden die theoretischen Grundlagen der einzelnen Werkzeuge genauer aufgezeigt und die in einer Marktanalyse ausgewählte Werkzeuge, welche in der Praxis Anwendung finden, näher analysiert. Abschließend wurden der Nutzen und die möglichen Risiken der einzelnen Werkzeuge evaluiert. Die ausgewählten Werkzeuge, welche vorgestellt wurden, erleichtern den Entwicklungsprozess in essenzieller Weise und gestalten die Arbeit effizienter. Genauer wurden intelligente Werkzeuge zur Codevollständigung, Codegenerierung, Analyse, Refactoring, Dokumentation und Kollaboration vorgestellt. Bei der Marktanalyse ist aufgefallen, dass es eine Vielzahl von Anbietern intelligenter Werkzeuge gibt. Zwar gibt es teilweise sprachübergreifende Werkzeuge, dies ist allerdings nicht immer der Fall. Manchmal kann daher eine gesonderte Analyse bezüglich verwendetem Technologie-Stack sinnvoller sein. Des Weiteren ist ersichtlich, dass bestimmte Entwicklungsumgebungen die genannten Werkzeuge bereits implementieren. Daher muss in diesen Fällen nicht auf eine externe Lösung gesetzt werden. Dies spiegelt aber auch die Akzeptanz und Wichtigkeit dieser Werkzeuge bei der Entwicklung wieder. Die Analysen zeigen, dass jedes der vorgestellten intelligenten Werkzeuge den Entwicklungsprozess effizienter gestaltet. Die genannten Gefahren dabei sind meist, dass sich der Entwickler zu sehr auf die Werkzeuge verlässt und diese nicht hinterfragt. Daher sind die Werkzeuge aufgrund ihrer Effizienzsteigerung zu empfehlen, jedoch sollten Entwickler ihnen nicht blind vertrauen und dennoch die Ausgabe selbst kontrollieren.

Wie aus der Arbeit hervorgeht, sind viele intelligente Werkzeuge jetzt schon in der Softwareentwicklung etabliert. Auch an den Nutzerzahlen der einzelnen Werkzeuge, die in den Marktanalysen genannt wurden, lässt sich dies aufzeigen. Dennoch gibt es auch andere Meinungen. Gerade wie sich in der Marktanalyse zum Thema Refactoring (Unterabschnitt 5.4) herausgestellt hat, ist das Vertrauen zu gewissen Werkzeugen noch nicht vollständig hergestellt und bedarf noch gewisser Erfahrung. Dennoch betrifft dies nicht alle Werkzeuge. Das Themengebiet der vollständig auf künstliche Intelligenz basierten Werkzeuge bietet in Zukunft noch viel Entwicklungspotenzial. Wie zum Beispiel das Werkzeug der Codegenerierung Abschnitt 2 zeigt, bietet es immense Möglichkeiten den Entwicklungsprozess zu erleichtern oder gar zu automatisieren. Von einer vollständigen Automatisierung ist hier aber in absehbarer Zeit nicht zu reden. Dennoch entwickelt sich das allgemeine Feld der künstlichen Intelligenz rasant und die Prognosen sind hierzu nur bestätigend [St22]. Deshalb

ist davon auszugehen, dass auch die intelligente Werkzeuge immer mehr von dieser Technik profitieren und dadurch weiterentwickelt werden. Wie die Arbeit des Weiteren aufzeigt ist das Themengebiet hoch aktuell und sehr wichtig. So werden auch in Zukunft noch einige interessante Neuerungen und Weiterentwicklungen in diesem Bereich stattfinden.

Literatur

- [Am04] Ambler, S. W.: *The object primer: Agile Model-driven development with UML 2.0* / Scott W. Ambler. Cambridge University Press, Cambridge, 2004, ISBN: 0521540186.
- [BMM09] Bruch, M.; Monperrus, M.; Mezini, M.: *Learning from Examples to Improve Code Completion Systems*. In: *Proceedings of the 7th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on The Foundations of Software Engineering*. Association for Computing Machinery, New York, NY, USA, S. 213–222, 2009, ISBN: 9781605580012.
- [Ch13] Chemuturi, M.: *Requirements Engineering and Management for Software Development Projects*. Springer Verlag, 2013, ISBN: 978-1-4614-5376-5.
- [Ch21] Chen, M.; Tworek, J.; Jun, H.; Yuan, Q.; de Oliveira Pinto, H. P.; Kaplan, J.; Edwards, H.; Burda, Y.; Joseph, N.; Brockman, G.; Ray, A.; Puri, R.; Krueger, G.; Petrov, M.; Khlaaf, H.; Sastry, G.; Mishkin, P.; Chan, B.; Gray, S.; Ryder, N.; Pavlov, M.; Power, A.; Kaiser, L.; Bavarian, M.; Winter, C.; Tillett, P.; Such, F. P.; Cummings, D.; Plappert, M.; Chantzis, F.; Barnes, E.; Herbert-Voss, A.; Guss, W. H.; Nichol, A.; Paino, A.; Tezak, N.; Tang, J.; Babuschkin, I.; Balaji, S.; Jain, S.; Saunders, W.; Hesse, C.; Carr, A. N.; Leike, J.; Achiam, J.; Misra, V.; Morikawa, E.; Radford, A.; Knight, M.; Brundage, M.; Murati, M.; Mayer, K.; Welinder, P.; McGrew, B.; Amodei, D.; McCandlish, S.; Sutskever, I.; Zaremba, W.: *Evaluating Large Language Models Trained on Code*. CoRR abs/2107.03374/, 2021, arXiv: 2107.03374, URL: <https://arxiv.org/abs/2107.03374>.
- [DA20] Davis; Anglin, Hrsg.: *Modern Programming Made Easy*. Apress, [Place of publication not identified], 2020, ISBN: 978-1-4842-5568-1.
- [Da88] Darwin, I. F.: *Checking C Programs with Lint* -. Ö'Reilly Media, Inc.", Sebastopol, 1988, ISBN: 978-0-937-17530-9.
- [De22] DevExpress: *CodeRush for Visual Studio*, 2022, URL: <https://www.devexpress.com/Products/CodeRush/>, Stand: 05. 11. 2022.
- [Do08] Dollard, K.: *Code Generation in Microsoft .NET*. Apress, New York, 2008, ISBN: 978-1-430-20705-4.
- [Do22] Doxygen: Doxygen: Generate documentation from source code, 2022, URL: %5Curl%7B<https://www.doxygen.nl/index.html%7D>, Stand: 15. 11. 2022.

- [EM21] Eilertsen, A. M.; Murphy, G. C.: The Usability (or Not) of Refactoring Tools. In: 2021 IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER). S. 237–248, 2021.
- [Fo00] Fowler Martin mit Beck, K.; Brant, J.; Opdyke, W.; Roberts, D.: Refactoring: Wie Sie das Design vorhandener Software verbessern. Addison-Wesley, 2000, ISBN: 3827316308.
- [Fo08] Foundation, O.: Code Review Guide Book v. 2.0. 2008, ISBN: 978-1-304-58673-5.
- [Gi17] GitHub, Inc.: Open Source Survey, 2017, URL: %5Curl%7Bhttps:////opensourcesurvey.org/2017/%7D, Stand: 02.11.2022.
- [GKF06] How are java software developers using the eclipse ide?, IEEE, 2006, S. 76–83.
- [gm12] gmath: Static Analysis vs Style Checkers, 2012, URL: https://lwn.net/Articles/483972/.
- [Jea] Jetbrains: All Products, URL: https://www.jetbrains.com/all/.
- [Jeb] Jetbrains: Static Code Analysis, URL: https://www.jetbrains.com/de-de/teamcity/ci-ci-guide/concepts/static-code-analysis/.
- [Je20] Jetbrains: Why Does Jetbrains Separate Their Products Into Multiple IDEs, 2020, URL: https://intellij-support.jetbrains.com/hc/en-us/community/posts/360006942459-why-does-Jetbrains-separate-their-products-into-multiple-IDEs.
- [Je22] JetBrains s.r.o.: UML class diagrams, 2022, URL: %5Curl%7Bhttps://www.jetbrains.com/help/idea/class-diagram.html%7D, Stand: 08.11.2022.
- [Ji16] Jill Reinauer: Using the Task List, 2016, URL: %5Curl%7Bhttps://learn.microsoft.com/en-us/previous-versions/visualstudio/visual-studio-2015/ide/using-the-task-list?view=vs-2015&redirectedfrom=MSDN#tokenscomments%7D, Stand: 03.11.2022.
- [Ki22] with Kite, C. F.: Kite - Free AI Coding Assistant and Code Auto-Complete Plugin, 2022, URL: %5Curl%7Bhttps://www.kite.com/%7D, Stand: 12.11.2022.
- [Ma02] Martin, R. C.: Agile software development, principles, patterns, and practices. Pearson, Upper Saddle River, NJ, 2002.
- [Ma09] Martin Robert C. mit Feathers, M. C.
bibinitperiod E. R.: Clean Code. Mitp-Verlag, 2009, ISBN: 9783826696381.
- [MCB15] Marasoiu, M.; Church, L.; Blackwell, A.: An empirical investigation of code completion usage by professional software developers. In: Proceedings of PPIG 2015. S. 71–82, 2015.
- [No22] Nolle, T.: Statische und dynamische Code Analyse zusammen einsetzen, 2022, URL: https://www.computerweekly.com/de/tipp/Statische-und-dynamische-Code-Analyse-zusammen-einsetzen.

- [Op21] OpenAI: OpenAI Codex, 2021, URL: <https://openai.com/blog/openai-codex/>.
- [Op22] OpenAI: OpenAI Codex Sandbox, 2022, URL: <https://beta.openai.com/codex-javascript-sandbox>.
- [Or97] Oracle: Code Conventions, 1997, URL: <https://www.oracle.com/technetwork/java/codeconventions-150003.pdf>.
- [Pa19] Papenbrock, T.: Data Profiling – Effiziente Entdeckung Struktureller Abhängigkeiten. In (Grust, T.; Naumann, F.; Böhm, A.; Lehner, W.; Härder, T.; Rahm, E.; Heuer, A.; Klettke, M.; Meyer, H., Hrsg.): BTW 2019. Gesellschaft für Informatik, Bonn, S. 467–476, 2019.
- [Re22a] ReShaper, J.: Quick-Fixes, 2022, URL: https://www.jetbrains.com/de-de/resharper/features/quick_fixes.html, Stand: 15.11.2022.
- [Re22b] ReShaper, J.: Refactorings, 2022, URL: https://www.jetbrains.com/help/resharper/Refactorings__Index.html, Stand: 05.11.2022.
- [Re22c] ReShaper, J.: ReSharper features in different languages, 2022, URL: https://www.jetbrains.com/help/resharper/Introduction_Feature_Map.html, Stand: 02.11.2022.
- [RL08] How Program History Can Improve Code Completion, IEEE, 2008, S. 317–326.
- [Ro00] Rosenfeld, R.: Two decades of statistical language modeling: where do we go from here? Proceedings of the IEEE 88/8, S. 1270–1278, 2000.
- [Sc19a] Schmidt, M.: Statische und dynamische Codeanalyse in einem kontinuierlichen Testprozess, 2019, URL: <https://www.embedded-software-engineering.de/statische-und-dynamische-codeanalyse-in-einem-kontinuierlichen-testprozess-a-846419/>.
- [Sc19b] van Scharrenburg, E.: Code Completion with Recurrent Neural Networks. In. 2019.
- [St07] Stahl, T.; Völter, M.; Efftinge, S.; Haase, A.: Modellgetriebene Softwareentwicklung: Techniken, Engineering, Management. dpunkt, Heidelberg, 2007, ISBN: 978-3-89864-448-8.
- [St22] Statista: Artificial Intelligence (AI) market size/revenue comparisons 2018–2030, Juni 2022, URL: <https://www.statista.com/statistics/941835/artificial-intelligence-market-size-revenue-comparisons/>.
- [Te22] Terry G. Lee: Design and view classes and types with Class Designer, 2022, URL: %5Curl%7B<https://learn.microsoft.com/en-us/visualstudio/ide/class-designer/designing-and-viewing-classes-and-types?view=vs-2022%7D>, Stand: 08.11.2022.
- [Th21] TheCodeBytes, 2021, URL: <https://thecodebytes.com/wp-content/uploads/2021/12/low-level-programming-languages.jpg>.

Vergleich von Frameworks zur Entwicklung hybrider Applikationen

Jannik Dürr¹, Dominic Joas², Ruben Kalmbach³

Abstract: Hybride Applikationen sind ein moderner Ansatz zur Vereinfachung der Softwareentwicklung. Basierend auf einer gemeinsamen Codebasis wird die Anwendung durch native Wrapper zu verschiedenen Betriebssystemen kompatibel gemacht. Aufgrund der großen Anzahl an Frameworks für die Entwicklung hybrider Applikationen werden in diesem Beitrag verschiedene Frameworks gegenübergestellt. Zunächst werden der Aufbau und die Eigenschaften hybrider Anwendungen beschrieben, sowie Vergleichskriterien und relevante Merkmale hybrider Frameworks zusammengefasst. Dann werden drei Anwendungsgebiete definiert: rechenintensiv, formularlastig und sensorlastig. Die Vergleichskriterien werden in einem paarweisen Vergleich für jedes Anwendungsgebiet separat gewichtet. Anschließend werden die einzelnen Frameworks mithilfe von Literatur und eigenen Untersuchungen betrachtet und anhand der Kriterien bewertet. Durch die Berechnung der Punktzahlen für jedes Framework können Empfehlungen für jedes Anwendungsgebiet gegeben werden.

Keywords: Hybride Applikation; Hybride Frameworks; Mobile Applikation; Native Applikation

1 Einleitung

In diesem Kapitel wird eine Einleitung in den Beitrag in Form einer Motivation gegeben. Anschließend werden die verwendeten Methoden erläutert, gefolgt von einem Überblick über das Vorgehen und die Ziele.

1.1 Motivation

Applikationen auf mobilen Geräten nehmen im Alltag vieler Menschen eine große Rolle ein. In den App-Stores von Google und Apple stehen mehrere Millionen Apps zur Verfügung, die jeden Bereich des täglichen Lebens abdecken. Bei vielen dieser Apps handelt es sich um native Applikationen, seit einigen Jahren gewinnen jedoch hybride Applikationen ebenfalls immer mehr an Popularität und kommen z. B. bei Social-Media-Apps wie Twitter erfolgreich zum Einsatz.

¹ DHBW Stuttgart Campus Horb, Studiengang Informatik, Florianstraße 15, 72160 Horb am Neckar, Deutschland,
i20007@hb.dhbw-stuttgart.de

² DHBW Stuttgart Campus Horb, Studiengang Informatik, Florianstraße 15, 72160 Horb am Neckar, Deutschland,
i20015@hb.dhbw-stuttgart.de

³ DHBW Stuttgart Campus Horb, Studiengang Informatik, Florianstraße 15, 72160 Horb am Neckar, Deutschland,
i20018@hb.dhbw-stuttgart.de

Hybride Applikationen versuchen, die Vorteile von nativen Applikationen und Web-Applikationen zu kombinieren. Native Applikationen sind Anwendungen, die speziell für ein konkretes Betriebssystem entwickelt und optimiert wurden. Web-Applikationen sind hingegen vollständig browserbasiert und unabhängig von konkreten Betriebssystemen, allerdings ist der Zugriff auf die Sensoren des Geräts stark eingeschränkt. Einen guten Kompromiss bilden die bereits angesprochenen hybriden Applikationen, die wie Web-Anwendungen mit Webtechnologien entwickelt werden, allerdings anschließend in native Container, auch Wrapper genannt, verpackt werden. So sind sie kompatibel zu mehreren Betriebssystemen und können dabei auf native Schnittstellen zugreifen. Der Arbeitsaufwand, der bei der Entwicklung von Apps ein zentraler Faktor ist, kann so stark reduziert werden, da die Entwicklung nicht für jedes Betriebssystem separat erfolgen muss.

Für die Entwicklung von hybriden Applikationen stehen eine Vielzahl an Frameworks zur Verfügung. Abbildung 1 stellt diesbezüglich dar, welche hybriden Frameworks weltweit von Entwicklern bevorzugt verwendet werden. Allerdings ist es für Entwickler, die sich nicht intensiv mit der Entwicklung hybrider Applikationen beschäftigt haben, oft schwierig zu entscheiden, welches Framework für bestimmte Anwendungsfälle verwendet werden sollte. Daher werden in dieser Arbeit die wichtigsten hybriden Frameworks genauer betrachtet und anhand definierter Kriterien miteinander verglichen. So können Empfehlungen abgegeben werden, welches Framework abhängig von den Anforderungen eingesetzt werden sollte.

1.2 Methoden

Folgende Methoden werden neben der Literaturarbeit verwendet:

- **Paarweiser Vergleich**

Durch einen paarweisen Vergleich werden die Vergleichskriterien systematisch paarweise gegenübergestellt und so ermittelt, welches der Kriterien wichtiger ist bzw. ob sie gleich wichtig sind. Aus der numerischen Interpretation der Gegenüberstellungen ergeben sich die Rangfolge und die relative Gewichtung der Vergleichskriterien.

- **Prototyping**

Durch die Entwicklung von Prototypen mit möglichst identischem Funktionsumfang werden Erfahrungen mit den Frameworks gesammelt und der Entwicklungsaufwand bewertet.

- **Benchmarking**

Durch die Durchführung eines Software-Benchmarks mit identischen Algorithmen werden Ausführungszeiten und CPU-Auslastung der Prototypen ermittelt.

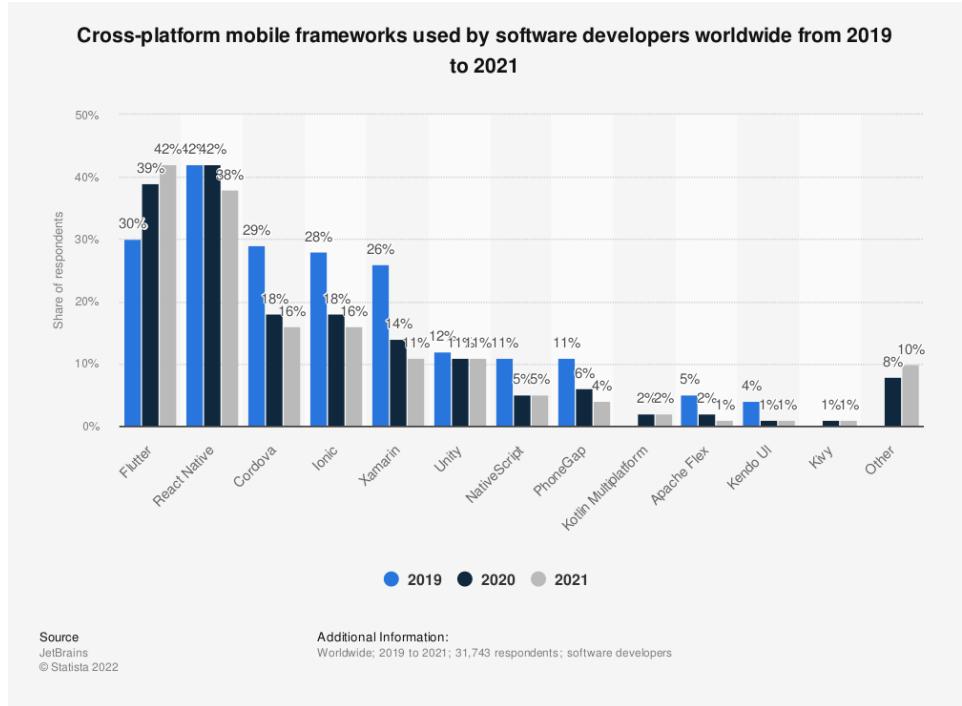


Abb. 1: Umfrage, welche hybriden Frameworks von Entwicklern verwendet werden [Statista 2021]

1.3 Vorgehen und Ziele

Das Ziel des Beitrags ist der Vergleich von Frameworks zur Entwicklung hybrider Applikationen. Zu diesem Zweck werden hybride Applikationen zunächst definiert und gegen mobile und native Applikationen abgegrenzt. Anschließend werden Vergleichskriterien ermittelt und Anwendungsgebiete für hybride Applikationen bestimmt, wobei den Kriterien jeweils eine Gewichtung zugeordnet wird. Dann werden sechs relevante, hybride Frameworks basierend auf Literatur und eigenen Erkenntnissen anhand der ermittelten Kriterien untersucht. Anschließend findet der Vergleich der Frameworks statt, wobei die hybriden Frameworks zunächst paarweise anhand einzelner Kriterien verglichen werden. Mithilfe von Prototypen und Benchmarks wird außerdem die Performance der Frameworks gemessen und ausgewertet. Anschließend werden die Ergebnisse zusammengefasst und es werden Empfehlungen gegeben, welches Framework sich für welches Anwendungsgebiet eignet.

2 Hybride Applikationen

In diesem Kapitel werden hybride Applikationen zunächst definiert und gegen mobile und native Applikationen abgegrenzt. Dabei wird außerdem der Aufbau behandelt. Anschließend werden Eigenschaften sowie Vor- und Nachteile von hybriden Applikationen genauer betrachtet.

2.1 Definition und Aufbau

Um eine Definition für hybride Applikationen zu finden, ist es notwendig, ein einheitliches Verständnis über native Applikationen, Web-Applikationen und Cross-Plattform-Applikationen zu schaffen. Daher werden diese Begriffe im Folgenden erläutert und gegeneinander abgegrenzt.

Native Applikation

Eine native Applikation ist eine App, die für genau ein Betriebssystem von mobilen Endgeräten entwickelt wird und in der Regel auch nur dort ausführbar ist. Für die Entwicklung muss die Programmiersprache des jeweiligen Betriebssystems (z.B. Java/Kotlin für Android) verwendet werden. Da native Applikationen speziell für das Betriebssystem entwickelt werden, fällt es auch leicht, Sensoren zu nutzen, die das jeweilige Endgerät anbietet. Durch die native Ausführung wird die App außerdem so optimiert, dass sie ressourcenschonend und leistungsstark ist.

Web-Applikation

Web-Applikationen werden in der Regel nicht lokal auf einem mobilen Endgerät installiert, sondern werden über eine Cloud oder einen Server bereitgestellt und sind über eine URL abrufbar. Dies bringt den Vorteil mit sich, dass man Web-Apps plattformübergreifend nutzen kann. Die App muss daher nur einmal entwickelt werden und nicht für jede Plattform neu. Durch die Ausführung auf einem Web-Server sind die Berechtigungen und Funktionalitäten der App aber eingeschränkt. Daher können die Sensoren des jeweiligen Endgerätes nicht genutzt werden und man kann nur eingeschränkt auf den Speicher des Geräts zugreifen. Hat ein Smartphone keinen Internet-Zugriff, so kann nicht auf den Web-Server zugegriffen und somit auch die App nicht genutzt werden.

Cross-Plattform Applikation

Eine Cross-Plattform Applikation ist eine Applikation, welche über eine gemeinsame Codebasis entwickelt wurde und auf verschiedene Betriebssysteme übertragen werden kann. Dies kann entweder durch die Entwicklung einer Web-Applikation (vgl. oben) oder die Entwicklung einer hybriden Applikation erreicht werden.

Hybride Applikation

Im Gegensatz zu einer Web-Applikation basiert eine hybride App nicht zwangsläufig auf Web-Technologien. Bei der Entwicklung hybrider Apps können stattdessen auch andere Technologien und Programmiersprachen (z. B. Dart oder C#) zum Einsatz kommen. Hybride Applikationen basieren auf einer gemeinsamen Code-Basis, die in nativen Code für verschiedene Betriebssysteme kompiliert wird. So kann im Gegensatz zu Web-Applikationen auch auf native Funktionen der jeweiligen Betriebssysteme wie z. B. Sensoren zugegriffen werden.

„Generally speaking, hybrid app is programmed in browser-supported language and wrapped as native app. It is called hybrid because it combines the features of both native and web applications.“ [Denko et al. 2021]

Die einzelnen Applikationsarten sind in Abbildung 2 entsprechend ihrer Flexibilität und ihres Funktionsumfangs angeordnet. Es ist erkennbar, dass native Apps den größten Funktionsumfang, aber gleichzeitig auch die geringste Flexibilität bieten. Web-Applikationen hingegen haben eine enorme Flexibilität, unterstützen allerdings nur einen geringen Funktionsumfang. Hybride Applikationen sind in der Mitte beider Applikationsarten angeordnet und bieten einen Mittelweg zur Entwicklung mit sowohl relativ hohem Funktionsumfang als auch relativ hoher Flexibilität.

Aufbau

Eine hybride App besteht in der Regel aus mehreren Teilen. Es gibt einen Kern, in dem sich der Quellcode für die eigentliche Applikation befindet und für Android und iOS jeweils einen Teil, in dem der native Code generiert wird. Der Kern der App wird häufig mit Web-Technologien erstellt. Dabei kann es sich z. B. um HTML, CSS und JavaScript handeln. Es können allerdings auch abweichende Sprachen (z. B. Dart oder C#) zum Einsatz kommen.

Frameworks für hybride Apps liefern eine JavaScript-Bridge mit, welche den eigentlichen Quellcode in nativen Code umwandelt. Dieser wird dann zur Erstellung von Android- bzw. iOS-Apps genutzt. Wurde der native Code erstellt, kann man diesen über die jeweilige Entwicklungsumgebung öffnen und wie den Quellcode einer nativen App verwenden.

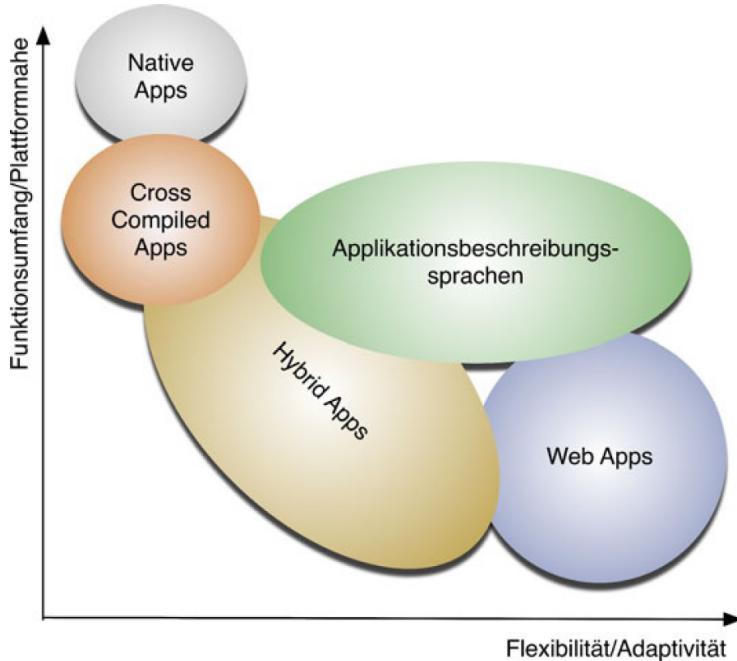


Abb. 2: Übersicht über Arten von Applikationen nach Flexibilität und Funktionsumfang [Willnecker et al. 2012, S.405]

Da eine hybride App oft unter dem Slogan „write once, run everywhere“ beworben wird, ist es notwendig, die Applikation unabhängig vom konkreten Betriebssystem auf gleiche Art und Weise anzuzeigen. Um dies zu erreichen, wird häufig in der App eine Webview angezeigt, welche den JavaScript-Code wiedergibt.

„Hybrid app looks and performs in native way but works like web app, so hybrid app possesses the great user experience of native app and cross-platform characteristic of web app at the same time.“

[Que et al. 2016]

2.2 Vor- und Nachteile

Hybride Applikationen bringen einige Vor- und Nachteile mit sich, die in diesem Abschnitt genauer betrachtet werden. Abschließend wird ein Fazit getroffen und die wichtigsten Aussagen zusammengefasst.

Vorteile

Der zentrale Vorteil hybrider Applikationen besteht in der Kompatibilität zu verschiedenen Betriebssystemen. Gemäß dem „Write once, run anywhere“-Paradigma ist die gemeinsame Codebasis mit den meistverwendeten, mobilen Betriebssystemen kompatibel [Denko et al. 2021]. Dabei handelt es sich Stand 2022 zu über 99% um die Betriebssysteme Android und iOS [Statista 2022]. Somit sind auch die Entwicklungskosten deutlich geringer als bei nativen Applikationen, da ein Entwicklungsteam für alle Betriebssysteme zeitgleich entwickeln kann [Kleinschrod 2020]. Dies bietet sich besonders für Start-Up-Unternehmen an, die mit wenigen Mitarbeitern schnell eine funktionsfähige Applikation entwickeln wollen. Da hybride Applikationen somit auch in den beiden App Stores von Google und Apple angeboten werden können, kann eine höhere Anzahl potentieller Nutzer erreicht werden als bei nativen Applikationen, die auf eine konkrete Plattform beschränkt sind [Tyshchenko 2020].

Da die Codebasis hybrider Applikationen mit Webtechnologien entwickelt wird, kommen im Entwicklungsprozess vergleichsweise einfach zu erlernende Programmiersprachen wie HTML, CSS und JavaScript zum Einsatz. Native Applikationen werden mit Java oder Kotlin für Android und Swift oder Objective-C für iOS entwickelt, wobei es sich jeweils um deutlich komplexere Programmiersprachen handelt [Tyshchenko 2020]. Der Zugriff auf die Sensoren des Endgeräts (z. B. GPS, Kamera oder Beschleunigungssensor) ist bei hybriden Applikationen durch Verwendung einer Bridge zu den Schnittstellen des Betriebssystems möglich [Que et al. 2016], wobei es sich um einen klaren Vorteil gegenüber Web-Applikationen handelt. Außerdem können hybride Applikationen genau wie native Applikationen Daten in lokalen Datenbanken abspeichern [Denko et al. 2021]. Sie sind somit auch offline verfügbar, was bei Web-Applikationen nicht der Fall ist.

Nachteile

Hybride Applikationen haben grundsätzlich eine geringere Performance als native Applikationen und nutzen die Leistungsfähigkeit des Endgeräts nicht optimal aus [Kleinschrod 2020]. Dies hat die Ursache, dass es durch die notwendige Kommunikation zwischen Applikation und nativen Komponenten zu Geschwindigkeitseinbußen kommt. Zudem ist es bei nativen Applikationen einfacher, die UX (User Experience) besser für jedes Betriebssystem zu optimieren, da direkt mit nativen Widgets gearbeitet werden kann [Cowart 2012]. Bei hybriden Applikationen ist dies nicht der Fall, sodass die Entwicklung einer Benutzeroberfläche, die auf allen Plattformen gut aussieht, anspruchsvoll sein kann. Außerdem sind hybride Applikationen tendenziell anfälliger für Fehler und Ausfälle als native Applikationen [Tyshchenko 2020].

Ein weiterer Aspekt der Entwicklung hybrider Applikationen ist der Testprozess. Dieser ist grundsätzlich aufwändiger als bei nativen Applikationen, da die Applikation für alle

Plattformen getestet werden muss. Dies erschwert auch die Testautomatisierung, da die Skripte einen größeren Funktionsumfang abdecken müssen [Tyshchenko 2020]. Neben den Funktionstests (z. B. Ressourcenzugriff und Verhalten), Oberflächentests (z. B. korrekte Darstellung) und Leistungstests (z. B. Lasttest und Stresstest) gibt es zwei weitere Testarten, die besonders im Testprozess hybrider Applikationen relevant sind. Dabei handelt es sich um Kompatibilitätstests und Verbindungstests [Tyshchenko 2020]. Kompatibilitätstests sind sinnvoll, um die korrekte Ausführung der hybriden Applikation auf mehreren Betriebssystemen sicherzustellen. Durch Verbindungstests wird das korrekte Verhalten der Applikationen online und offline sichergestellt.

Fazit

Hybride Applikationen sind ein guter Kompromiss zwischen nativen Applikationen und Web-Applikationen. Sie kombinieren die wichtigsten Vorteile der beiden Applikationsarten und versuchen, daraus entstehende Nachteile bestmöglich zu reduzieren. Zusammengefasst kann man sagen, dass hybride Applikationen die schnelle, einfache und kostengünstige Entwicklung einer Applikation für mehrere Betriebssysteme ermöglichen und dafür eine leicht reduzierte Performance, eine etwas erschwerete Optimierung der Benutzeroberfläche und einen etwas aufwändigeren Testprozess in Kauf nehmen.

2.3 Potential

Trotz der einzugehenden Kompromisse gegenüber nativen Applikationen sind beliebte und vielverwendete hybride Applikationen bereits Bestandteil unseres Alltags und im Fokus von großen Unternehmen. Bekannte Beispiele für hybride Applikationen sind Twitter, Instagram, Evernote, Uber und Remote POS.

Diese Apps demonstrieren, dass die Leistungsfähigkeit von hybriden Applikationen durchaus den derzeitigen Anforderungen des Markts gewachsen sind und ein Interesse in der Verwendung und Weiterentwicklung von hybriden Frameworks besteht [Dharmwan 2021].

Die Möglichkeit, eine Applikation in den App Stores gängiger Anbieter zu veröffentlichen, ohne dass die Entwicklungskosten proportional zu der Anzahl der gewünschten Zielplattformen wachsen, ist ein Potential, das Firmen bei der Entwicklung ihrer Applikationen nutzen können und möchten.

3 Vergleichskriterien

In diesem Kapitel werden Vergleichskriterien ermittelt, anhand derer die Frameworks untersucht werden. Anschließend werden Anwendungsgebiete bestimmt und die Relevanz der Vergleichskriterien je Anwendungsgebiet gewichtet.

3.1 Bestimmung

Folgende Vergleichskriterien werden bei der Untersuchung der Frameworks berücksichtigt.

Plattformspezifische Funktionen

Die Zugriffsmöglichkeiten des Frameworks auf plattformspezifische Funktionen wird untersucht. Dies umfasst z. B. die Speicherverwaltung, die sich je nach Betriebssystem unterscheidet. Außerdem wird der Zugriff auf verschiedene Sensoren betrachtet. Dazu gehört u. A. die Standortermittlung durch GPS, der Umgebungslichtsensor oder das Magnetometer.

Performance

Die Performance des Frameworks wird u. A. mithilfe von Literatur bewertet. Dabei werden Geschwindigkeitseinbußen bei der Kommunikation zwischen Codebasis und dem nativen Wrapper, der plattformspezifische Funktionen realisiert, berücksichtigt. Zur detaillierten Bestimmung der Leistungswerte für die definierten Anwendungsgebiete sind Prototyping und Benchmarking notwendig. Ein weiterer Aspekt ist u. A. die Dateigröße des Builds für die jeweiligen Zielplattformen.

Benutzeroberfläche

Die Vorgehensweise bei der Erstellung der Benutzeroberfläche wird untersucht. Dabei wird betrachtet, in welchem Umfang grafische Elemente zur Verfügung stehen und durch welche strukturellen Elemente die responsive Anordnung der Elemente umgesetzt ist. Außerdem wird untersucht, welche Technologien (z. B. HTML & CSS oder native Elemente) zur Entwicklung der Benutzeroberfläche eingesetzt werden.

Erste Schritte

Die Einrichtung des Frameworks wird betrachtet. Dabei wird untersucht, wie aufwändig und schwierig die Installation ist und ob weitere Abhängigkeiten zu dritten Programmen bestehen, die ebenfalls installiert werden müssen. Außerdem wird die Komplexität der Einarbeitung in das Framework bewertet, wobei z. B. Einstiegsschwierigkeiten hervorgehoben werden.

Entwicklungsunterstützung

Die Unterstützung bei der Softwareentwicklung mit dem Framework wird bewertet. Dabei wird untersucht, welche IDEs verwendet werden können und ob dabei für die kommerzielle Verwendung kostenpflichtige Lizenzen benötigt werden. Außerdem wird betrachtet, ob es IDE-Erweiterungen für das Framework gibt und ob Templates für die Entwicklung existieren.

Dokumentation

Die Dokumentation des Frameworks wird bewertet. Dabei wird Wert auf den Umfang und die Nutzerfreundlichkeit der Dokumentation gelegt. Außerdem wird die Übersichtlichkeit und die Verfügbarkeit betrachtet.

Ökosystem

Das Ökosystem des Frameworks wird untersucht. Dabei wird die Vielzahl der zur Verfügung stehenden Klassenbibliotheken betrachtet, die den Funktionsumfang des Frameworks erweitern. Außerdem wird die Kompatibilität zu externen Diensten untersucht.

Lebenszyklus & öffentliches Interesse

Der Lebenszyklus des Frameworks wird betrachtet. Es wird bewertet, wie populär und modern das Framework ist und an welchem Punkt des Lebenszyklus es sich befindet. So kann bestimmt werden, wie wahrscheinlich eine zeitnahe Einstellung der Weiterentwicklung des Frameworks ist und ob sich ein Umstieg auf die langfristige Softwareentwicklung mithilfe des Frameworks empfiehlt.

Außerdem werden folgende Aspekte betrachtet, die jedoch nicht Teil des anschließenden Vergleichs sind.

Programmiersprachen Programmiersprachen, in denen die Softwareentwicklung erfolgt

Zielplattformen Zielplattformen, für die Software entwickelt werden kann

Wartbarkeit Architektur-Patterns, die mithilfe des Frameworks umgesetzt werden können

Datenhaltung DBMS, die vom Framework unterstützt werden

Lizenzen & Kosten Lizenz, unter der das Framework steht und eventuell anfallende Kosten

3.2 Anwendungsgebiete

Um die Frameworks bestmöglich zu testen, müssen Anwendungsgebiete definiert werden, für die in den jeweiligen Frameworks beispielhaft hybride Applikationen erstellt werden. Diese Anwendungsgebiete werden im Folgenden näher beschrieben.

Formularlastige Anwendung

Eine formularlastige Anwendung ist eine Anwendung, in welche Benutzer*innen Informationen über Formulare eingeben können. Diese Daten sollen beispielhaft in einer lokalen Datenbank abgespeichert und wieder daraus geladen werden. Außerdem wird bei der Entwicklung der Funktionsumfang und der Aufwand zur Erstellung der Formulare betrachtet. Auch die Usability soll ermittelt werden, indem notiert wird, was beim Ausfüllen der jeweiligen Formulare auffällt und was positiv bzw. negativ hervorzuheben ist.

Sensorlastige Anwendung

Damit ein Smartphone mit der Umwelt kommunizieren kann, haben aktuelle Geräte eine große Anzahl an Sensoren eingebaut. Ein aktuelles Smartphone der Mittelklasse hat üblicherweise mindestens folgende Sensoren eingebaut:

Gyroskop Misst die Rotation des Geräts.

Beschleunigungssensor Misst, ob das Gerät in Bewegung ist.

Lichtsensor Misst die Helligkeit der Umgebung.

GPS-Sensor Bestimmt die Position des Geräts.

Kamera Aufnahme von Bildern und Videos.

Fingerabdrucksensor Registriert den Fingerabdruck der Benutzer*in.

Um zu ermitteln, welche Sensoren das jeweilige Framework nutzen kann, soll eine Applikation entwickelt werden, welche diese Sensoren im Einsatz zeigt. Hierbei soll auch berücksichtigt werden, wie einfach es ist, die Messungen der Sensoren in der Applikation zu berücksichtigen.

Rechenintensive Anwendung

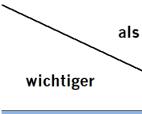
Um die Leistung der Frameworks zu messen, sollen rechenintensive Funktionen implementiert werden. Hierbei soll ermittelt werden, wie lange das Framework für die Berechnungen benötigt und wie stark die CPU ausgelastet wird. Außerdem soll ermittelt werden, wie das Framework mit Extremsituationen wie einem hohen Leistungsbedarf zurechtkommt und ob es vielleicht sogar ab einem gewissen Grad abstürzt.

3.3 Gewichtung

Die Anforderungen an eine Applikation variieren von Anwendungsgebiet zu Anwendungsgebiet. Um dies im Vergleich zu berücksichtigen, erfolgt eine individuelle Gewichtung der Vergleichskriterien je Anwendungsgebiet. Im Rahmen eines paarweisen Vergleichs werden die Vergleichskriterien systematisch gegenübergestellt und entschieden, welches der betrachteten Kriterien wichtiger ist. Die Summe der gewonnenen Vergleiche wird durch die Anzahl der Vergleichskriterien geteilt, um eine prozentuale Gewichtung des betrachteten Vergleichskriteriums zu erhalten.

Im Folgenden werden die paarweise Vergleiche und die daraus resultierende Gewichtung der Vergleichskriterien für die Anwendungsgebiete dargestellt und Auffälligkeiten aufgezeigt.

3.3.1 Formularlastige Anwendung

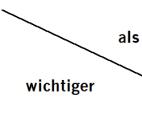


	Plattformspezifische Funktionen	Performance	Benutzeroberfläche	Erste Schritte	Entwicklungsunterstützung	Dokumentation	Ökosystem	Lebenszyklus & öffentliches Interesse	Summe	%
Plattformspezifische Funktionen	0	0	1	0	0	1	0	0	2	7%
Performance	1	0	1	0	1	1	0	0	4	14%
Benutzeroberfläche	1	1	0	1	1	1	1	1	7	25%
Erste Schritte	0	0	0	1	0	0	0	0	1	4%
Entwicklungsunterstützung	1	1	0	0	1	1	1	0	4	14%
Dokumentation	1	0	0	1	0	1	0	0	3	11%
Ökosystem	0	0	0	1	0	0	0	0	1	4%
Lebenszyklus & öffentliches Interesse	1	1	0	1	1	1	1	1	6	21%
Prüfsumme									100,00%	

Abb. 3: Gewichtung der Kriterien in formularlastigen Anwendungen

In Abbildung 3 ist der paarweise Vergleich und die daraus resultierende Gewichtung der Vergleichskriterien für das Anwendungsgebiet der formularlastigen Anwendungen dargestellt. Die Benutzeroberfläche wurde wichtiger als alle sieben verbleibenden Vergleichskriterien eingestuft und erhält dadurch eine relative Gewichtung von 25%. Der Lebenszyklus und das öffentliche Interesse erhält eine relative Gewichtung von 21%. Performance und Entwicklungsunterstützung erhalten jeweils eine relative Gewichtung von 14%. Erste Schritte und Ökosystem erhalten jeweils eine relative Gewichtung von 4%.

3.3.2 Sensorlastige Anwendung



	Plattformspezifische Funktionen	Performance	Benutzeroberfläche	Erste Schritte	Entwicklungsunterstützung	Dokumentation	Ökosystem	Lebenszyklus & öffentliches Interesse	Summe	%
Plattformspezifische Funktionen	1	1	1	1	1	1	1	1	7	25%
Performance	0	1	1	1	0	0	1	0	3	11%
Benutzeroberfläche	0	0	1	0	0	0	0	0	0	0%
Erste Schritte	0	0	1	1	0	0	0	0	1	4%
Entwicklungsunterstützung	0	1	1	1	1	0	0	1	4	14%
Dokumentation	0	1	1	1	1	1	0	0	4	14%
Ökosystem	0	0	1	1	1	1	1	0	4	14%
Lebenszyklus & öffentliches Interesse	0	1	1	1	0	1	1	1	5	18%
Prüfsumme									100,00%	

Abb. 4: Gewichtung der Kriterien in sensorlastigen Anwendungen

In Abbildung 4 ist der paarweise Vergleich und die daraus resultierende Gewichtung der Vergleichskriterien für das Anwendungsgebiet der sensorlastigen Anwendungen darge-

stellt. Die plattformspezifische Funktionen wurde wichtiger als alle sieben verbleibenden Vergleichskriterien eingestuft und erhalten dadurch eine relative Gewichtung von 25%. Der Lebenszyklus und das öffentliche Interesse erhält eine relative Gewichtung von 18%. Entwicklungsunterstützung, Dokumentation und Ökosystem erhalten jeweils eine relative Gewichtung von 14%. Alle Vergleichskriterien wurden wichtiger als die Benutzeroberfläche eingestuft, diese erhält dadurch eine relative Gewichtung von 0%.

3.3.3 Rechenintensive Anwendung

	als wichtiger	Plattformspezifische Funktionen	Performance	Benutzeroberfläche	Erste Schritte	Entwicklungsunter- stützung	Dokumentation	Ökosystem	Lebenszyklus & öffentliches Interesse	Summe	%
Plattformspezifische Funktionen		0	1	1	0	1	0	0	0	3	11%
Performance		1		1	1	1	1	1	1	7	25%
Benutzeroberfläche		0	0		0	0	0	0	0	0	0%
Erste Schritte		0	0	1		0	0	0	0	1	4%
Entwicklungsunterstützung		1	0	1	1		1	0	0	4	14%
Dokumentation		0	0	1	1	0		1	0	3	11%
Ökosystem		1	0	1	1	1	0		0	4	14%
Lebenszyklus & öffentliches Interesse		1	0	1	1	1	1	1		6	21%
										Prüfsumme	100,00%

Abb. 5: Gewichtung der Kriterien in rechenintensiven Anwendungen

In Abbildung 5 ist der paarweise Vergleich und die daraus resultierende Gewichtung der Vergleichskriterien für das Anwendungsgebiet der rechenintensiven Anwendungen dargestellt. Die Performance wurde wichtiger als alle sieben verbleibenden Vergleichskriterien eingestuft und erhält dadurch eine relative Gewichtung von 25%. Der Lebenszyklus und das öffentliche Interesse erhält eine relative Gewichtung von 21%. Entwicklungsunterstützung, sowie Ökosystem, erhalten jeweils eine relative Gewichtung von 14%. Alle Vergleichskriterien wurden wichtiger als die Benutzeroberfläche eingestuft, diese erhält dadurch eine relative Gewichtung von 0%.

3.3.4 Auffälligkeiten

Bei der Betrachtung der Ergebnisse fällt auf, dass Lebenszyklus und öffentliches Interesse bei jedem Anwendungsgebiet als zweitwichtigstes Kriterium gewichtet ist. Außerdem ist erkannbar, dass die Benutzeroberfläche hauptsächlich bei formularlastigen Anwendungen relevant und bei rechenintensiven und sensorlastigen Anwendungen zu vernachlässigen ist. In jedem Anwendungsgebiet dominiert ein Vergleichskriterium mit 25%, welches repräsentativ für die Anforderungen des Anwendungsgebiets ist.

4 Hybride Frameworks

In diesem Kapitel werden Frameworks zur Entwicklung hybrider Applikationen untersucht. Dabei werden die Kriterien aus dem vorherigen Kapitel berücksichtigt.

4.1 Flutter

Flutter ist ein modernes, hybrides Framework, das von Google entwickelt und 2018 veröffentlicht wurde. Als Open-Source-Framework steht unter der BSD-Lizenz und ist somit in kommerziellen Projekten nutzbar. Die Entwicklung erfolgt in der objektorientierten Programmiersprache Dart, die als Nachfolger von JavaScript entwickelt wurde und viele der charakteristischen JavaScript-Funktionalitäten implementiert, sich dabei allerdings eher an der Syntax von Java orientiert [Wu 2018]. Außerdem können Bibliotheken in den Programmiersprachen C und C++ eingebunden werden. Mit Flutter können Applikationen für Android, iOS, Windows, Linux, macOS und Browser wie Firefox, Chrome oder Edge entwickelt werden.

Obwohl Flutter ein vergleichsweise neues und modernes Framework ist, zeichnet es sich durch eine detaillierte und umfangreiche Dokumentation aus. Diese deckt den gesamten Entwicklungsprozess ab und leitet den Entwickler von der Installation bis zum Deployment der Flutter-Applikationen. Außerdem sind Beispielprojekte und Videoanleitungen auf dem Youtube-Kanal von Flutter zu finden. Die Dokumentation ist online⁴ einsehbar und steht bei Bedarf ebenfalls als Download⁵ zur Verfügung. Zum aktuellen Zeitpunkt ist sie ausschließlich auf Englisch verfasst.

Flutter ist für die Plattformen Windows, macOS, Linux und ChromeOS verfügbar. Zur Installation muss das FlutterSDK inklusive der enthaltenen Dart Virtual Machine heruntergeladen, extrahiert und im gewünschten Verzeichnis abgelegt werden. Flutter benötigt mehrere Abhängigkeiten zu anderen Programmen. Diese können mithilfe des Befehls *flutter doctor* in der Konsole abgefragt werden. Konkret werden Android SDK, Android Studio, Visual Studio, Visual Studio Code, IntelliJ IDEA und Chrome benötigt. Außerdem werden sogenannte Host Devices benötigt, auf denen das Debugging der Applikationen stattfinden kann. Für Android können diese im Android Emulator von Android Studio konfiguriert werden. Dazu ist es jedoch notwendig, die VM Hardware Acceleration zu aktivieren, was einen gewissen Aufwand mit sich bringt. Genaue Anweisungen sind in der Flutter Dokumentation zu finden.

Die Entwicklung von Flutter-Applikationen kann in verschiedenen IDEs (Integrierte Entwicklungsumgebung) erfolgen. Visual Studio Code ist die populärste IDE, da sie auch in kommerziellen Projekten kostenlos nutzbar ist und mit der *Flutter Extension* weitere Funktionalitäten zur Bearbeitung, Refactoring und Ausführung der Applikationen hinzugefügt

⁴ <https://docs.flutter.dev/>

⁵ <https://api.flutter.dev/offline/flutter.docset.tar.gz>

werden können. Weitere IDEs sind z. B. Android Studio oder IntelliJ IDEA. Besonders hervorgehoben werden muss die Hot-Reload-Funktion von Flutter, die ein schnelles Nachladen bei Änderungen während der Ausführung der Applikation ermöglicht und auch bei größeren Änderungen noch zuverlässig und schnell funktioniert. Dies geschieht durch Injection der modifizierten Code-Dateien in die laufende Dart Virtual Machine [Zammetti 2019]. Außerdem stehen für Flutter hunderte Templates auf diversen Websites⁶ zur Verfügung, die die Entwicklung deutlich beschleunigen können und viele Anwendungsgebiete bereits abdecken.

Flutter ermöglicht einen einfachen Zugriff auf spezifische Funktionen der jeweiligen Zielplattform. Mithilfe von Paketen (z. B. *sensors_plus* oder *flutter_sensors*) kann auf die Sensoren des Endgeräts wie z.B. Beschleunigungssensor, Gyroskop, Magnetometer, Näherungssensor und viele mehr zugegriffen werden. Außerdem ist es möglich, den Zugriff auf plattformspezifische Schnittstellen selbst in Dart zu implementieren, um z. B. den Akkustand abzufragen. Die Speicher verwaltung der Applikationen liegt nicht in der Hand des Entwicklers, sondern wird von der Dart Virtual Machine übernommen, die u. A. einen Garbage Collector für die Freigabe von nicht mehr benötigtem Speicherplatz umfasst.

Die Benutzeroberfläche wird in Flutter modular, d. h. in Form von Komponenten, entwickelt. Dabei verwendet Flutter nicht wie die meisten Frameworks OEM (Original Equipment Manufacturer)-Widgets, die vom Betriebssystem des Endgeräts zur Verfügung gestellt werden, sondern generiert eigene Widgets. Dies ermöglicht ein einzigartiges Design der Widgets und erhöht die Erweiterbarkeit und Flexibilität [xster 2017]. Für die Umsetzung der Benutzeroberfläche wird die Komponente *MaterialApp* als Top-Level-Widget verwendet, da sie z. B. Kopfzeile und Navigator bereits enthält. Die Oberfläche ist üblicherweise als Grid-Struktur organisiert, sodass die Anordnung der Widgets in Reihen (Row) und Spalten (Column) möglich ist. Für das Styling ist kein CSS (Cascading Style Sheets) notwendig, da die Gestaltung mithilfe von Decoration- und Style-Elementen direkt in Dart erfolgen kann. Zu diesem Zweck stehen auch weitere Elemente wie Center(), Positioned() oder Transform() zur Verfügung, um die Widgets innerhalb der Grid-Zelle auszurichten.

Da Flutter nicht die OEM-Widgets des Betriebssystems verwendet, kann in der Regel eine höhere Performance als bei anderen Frameworks erreicht werden. Dies hat den Hintergrund, dass Geschwindigkeitseinbußen, die durch die Kommunikation zwischen Applikation und nativen Komponenten entstehen würden, vermieden werden können [Helios Blog 2020]. Stattdessen nutzt Flutter seine eigene Hochleistungsengine Skia zum Rendern der benötigten Widgets [Wu 2018]. Man spricht daher davon, dass das Rendering von der Systemebene in die Applikationsebene verlagert wird. Dies führt allerdings dazu, dass die Build-Dateien größer als bei anderen Frameworks sind, da die eigenen Widgets und Renderer ebenfalls in der releasesten Applikation enthalten sein müssen [Wu 2018]. Grundsätzlich ist Dart als eine Programmiersprache mit hoher Performance zu betrachten, da sowohl JIT (Just-in-Time)-Kompilierung für die Entwicklung als auch AOT (Ahead-of-Time)-Kompilierung

⁶ z. B. <https://flutterawesome.com/tag/templates/>

für den Release der Applikation eingesetzt werden [Dart Documentation 2022]. AOT-kompilierter Code ermöglicht einen schnellen Start des Programms und zuverlässig geringe Ausführungszeiten während des gesamten Programmablaufs [Obinna 2020].

Im Flutter-Ökosystem steht eine große Anzahl an Paketen zur Verfügung. Dabei ist nahezu jeder Bereich von Datenbanken über WebSockets und Audio bis hin zu Animationen abgedeckt. Zur Organisation wird das offizielle Paket-Repository pub.dev verwendet. Von dort können die Pakete über den Pub Package Manager einfach in eigene Projekte integriert werden, indem die Dependency mit der benötigten Version in einer Projektdatei hinterlegt wird und dann beim nächsten Build aufgelöst wird. Anschließend kann das Paket mithilfe des import-Befehls verwendet werden.

Für die Datenhaltung steht eine recht begrenzte, aber wachsende Auswahl an Datenbanksystemen zur Verfügung. Die Pakete sqflite und moor erlauben den Zugriff auf das relationale Datenbanksystem SQLite, indem sie als Wrapper speziell für Flutter fungieren und SQL-Queries sowohl in SQL als auch direkt in Dart ermöglichen. Der Zugriff auf NoSQL Datenbanksysteme ist ebenfalls möglich. Zu diesem Zweck können die Pakete hive und firebase verwendet werden, die Unterstützung für die gleichnamige Key-Value-Datenbank und JSON-Datenbank bieten [Greenrobot 2021].

Flutter steht seit seinem Release in Konkurrenz mit dem Framework React Native. Es ist nicht eindeutig absehbar, welches Framework sich langfristig durchsetzen wird. Allerdings gewinnt Flutter in den letzten Jahren immer weiter Marktanteile und ist seit 2021 das meistverwendete, hybride Framework am Markt [Statista 2021]. Der Umstieg von Unternehmen auf die Softwareentwicklung mit Flutter scheint daher langfristig sinnvoll, da es sich voraussichtlich weiterhin als eines der meistverwendeten Frameworks etablieren wird.

4.2 React Native

React Native wurde im Jahr 2015 von Meta entwickelt und bis zum Jahr 2020 so weiterentwickelt, dass es ausgereift und für den Einsatz im Produktivumfeld geeignet ist. React Native begann als internes Hackathon-Projekt, dabei war das eigentliche Ziel des Projekts, den Entwicklungsprozess von Android und iOS zu vereinheitlichen [Niemeier 2022]. Das Framework ist Open-Source und steht unter der MIT-Lizenz. Aufgrund dieser Lizenz kann das Framework auch kommerziell verwendet werden. Es unterstützt die wichtigsten mobilen Plattformen Android und iOS genauso wie Windows, macOS und AndroidTV. Nicht zuletzt unterstützt es natürlich auch das Entwickeln von Webanwendungen. Viele namhafte Applikationen setzen bereits React Native ein, darunter sind Anwendungen wie Instagram, Skype oder auch Uber Eats [Chandratre 2020].

Die Entwicklung einer App in React Native erfolgt auf Basis von Webtechnologien wie HTML, CSS und JavaScript [Zammetti 2019]. Das Framework setzt unter Anderem auch die verwandte JavaScript-Bibliothek ReactJS ein, welche auch von Meta entwickelt wurde.

Der Unterschied zwischen React und React Native ist, dass React auf die Entwicklung von Web-Applikationen spezialisiert ist, während React Native für die Entwicklung von nativen Apps optimiert ist [Lestal 2020]. Entwickler, die bereits in React Anwendungen entwickelt haben, können diesen Code auch weiterhin nutzen [Krypczyk et al. 2021].

React Native kann schnell und einfach eingerichtet werden. Für die Entwicklungsumgebungen von Jetbrains gibt es Plugins, welche ganz bequem über das Plugin-Portal installiert werden können. Genauso gibt es auch für Visual Studio Code aus dem Hause Microsoft Unterstützung durch Plugins. Um React Native zu installieren, muss zuerst NodeJS und NPM installiert werden. NodeJS ist ein Framework, welches die Entwicklung von JavaScript-Anwendungen erleichtert. Über NPM können andere Bibliotheken per Kommandozeilenbefehl installiert werden. Für React Native gibt es auch ein sogenanntes Command Line Interface. Über dieses Interface kann man das System auf einfache Weise aktualisieren oder Erweiterungen installieren. Ist NPM installiert, so kann durch Aufrufen bestimmter Befehle in der Kommandozeile das Framework installiert und eine Anwendung eingerichtet werden. Da der Code von React Native in nativen Code umgewandelt wird, müssen für Android und iOS ebenso auch die nativen Entwicklungsumgebungen installiert werden. Dies wären Android Studio für Android und AppCode für iOS. Über diese nativen Entwicklungsumgebungen kann man auch Emulatoren der jeweiligen Betriebssysteme starten und so die App auf den Geräten testen⁷.

Projekte in React Native bestehen aus Code, welcher in JavaScript erstellt wird. Dieser ist in Views unterteilt, welche die Oberfläche widerspiegeln und Services, welche die Fachlogik enthalten. Oberflächen werden in React entwickelt, die Geschäftslogik wird in JavaScript verfasst. Nach dem Build der Anwendung werden Views in native Views umgewandelt. Services nutzen über eine Bridge die Schnittstellen der nativen Plattformen [Krypczyk et al. 2021]. React Native unterstützt den Hot-Reloading Mechanismus, sodass Änderungen im Programmcode zur Laufzeit durchgeführt werden können, ohne dass die Anwendung neu gestartet werden muss. Jede React Native Anwendung besteht hauptsächlich aus zwei unterschiedlichen Arten von Threads. Dabei kümmert sich einer der Threads als Haupt-Thread um die Anzeige der Elemente in der Benutzeroberfläche, während der andere Thread für die Ausführung des JavaScript-Codes verantwortlich ist. Er definiert außerdem die Funktionalitäten der Elemente auf der Benutzeroberfläche [Niemeier 2022].

Da React Native Open-Source ist, gibt es viele Bibliotheken und Erweiterungen von anderen Entwicklern. Daher gibt es auch Bibliotheken, welche React Native um die Nutzung von Sensoren erweitern [Schmidt 2018]. So werden z. B. Geschwindigkeitssensor, Gyroskop, Magnetometer und Barometer unterstützt. Über eine gut dokumentierte, native Schnittstelle werden auch native Elemente für die Oberfläche unterstützt. Dabei kann es sich z. B. um diverse Views in Android handeln.

In einer Datei können verschiedene Views hinterlegt werden, welche die Oberflächen der Applikation definieren. Views können sich auch je nach Plattform unterscheiden, sodass

⁷ https://www.tutorialspoint.com/react_native/react_native_environment_setup.htm

eine App auf Android später anders aussehen kann als auf iOS. Die einzelnen Views können modular aufgebaut werden und aus verschiedenen Dateien bestehen.

React Native unterstützt sehr viele Technologien und Schnittstellen, welche häufig bei der App-Entwicklung benötigt werden. So kann grundsätzlich auf native Oberflächenelemente zugegriffen werden. Der grundlegende Funktionsumfang kann beliebig durch eine der zahlreichen Erweiterungen ergänzt werden. Zum Beispiel kann durch Erweiterungen die Unterstützung für folgende Datenbank-Systemen ermöglicht werden: Firebase, SQLite, Realm, PouchDB, AsyncStorage und Weitere.

Zusammenfassend kann festgehalten werden, dass React Native voraussichtlich noch länger unterstützt wird, da hinter dem Framework eine namhafte Firma wie Meta steht. Das Projekt wird auf Open-Source-Basis entwickelt und kann so potenziell von jedem Entwickler weiterentwickelt werden. React Native hat einen großen Funktionsumfang und kann durch zusätzliche Erweiterungen dynamisch ausgebaut werden. Applikationen werden einmal gebaut und können für eine Vielzahl von Plattformen entwickelt werden. Um native Apps zu testen, müssen allerdings auch Entwicklungsumgebungen für die jeweiligen Plattformen installiert sein.

4.3 .NET MAUI

.NET MAUI (Multi-Platform App UI) ist ein Framework von Microsoft, mit dem seit 2022 plattformunabhängige Applikationen entwickelt werden können. MAUI ist der offizielle Nachfolger von Xamarin und verwendet bei linuxbasierten Zielplattformen ebenfalls die Laufzeitumgebung Mono [Davidbritch 2022d].

Mit MAUI können Applikationen für Android, iOS, Windows, macOS und Tizen entwickelt werden. Die gemeinsame Codebasis wird in der objektorientierten Programmiersprache C# geschrieben. Die Benutzeroberfläche kann wahlweise über die XML-basierte Beschreibungssprache XAML (Extensible Application Markup Language) oder unter Verwendung des Single-Page Web-Frameworks .NET Blazor erstellt werden. Bei letzterem spricht man von MAUI Blazor oder auch Blazor Hybrid, welches die Zielplattform um die gängigen Browser wie Chrome, Firefox und Edge erweitert.

Die Benutzeroberfläche wird in MAUI Blazor modular, d. h. in Form von wiederverwenbaren, dynamischen Komponenten entwickelt. Bei einer Razor-Komponente handelt es sich um einen eigenständigen Teil der Benutzeroberfläche und die dazugehörige Verarbeitungslogik. Die Komponenten werden in einer Kombination aus C#, HTML-Markup und optionaler JavaScript implementiert. C# ersetzt hierbei das bei Web-Frameworks üblicherweise benötigte JavaScript, sodass keine Kenntnisse über JavaScript benötigt werden. Die Darstellungsvorgaben können über CSS beliebig definiert werden. Sowohl einzelne Seiten als auch die komplette Blazor Applikation lassen sich über ein BlazorWebView-Steuerelement in der MAUI Applikation hosten. Die Web-UI wird anschließend in dem eingebetteten

Steuerelement gerendert und die Komponenten werden im systemeigenen Prozess ausgeführt [Davidbritch 2022a].

MAUI unterliegt der MIT-Lizenz und Blazor unterliegt der Apache-2.0-Lizenz, somit handelt es sich um Open-Source Software.

Obwohl MAUI und Blazor vergleichsweise neue Frameworks sind, zeichnen sie sich durch eine detaillierte und umfangreiche Dokumentation aus. Diese deckt den gesamten Entwicklungsprozess ab und leitet Entwickler*innen von der Installation bis zum Deployment der hybriden Applikation. Außerdem sind Beispielprojekte und Videoanleitungen auf der Lernplattform⁸ und dem Youtube-Kanal von Microsoft und dotnet zu finden. Die Dokumentation ist online⁹ einsehbar und verweist auf weitere nützliche Ressourcen. Zum aktuellen Zeitpunkt ist sie auf Englisch und auf Deutsch verfasst.

Die Entwicklung von MAUI Applikationen kann in verschiedenen IDEs erfolgen. Visual Studio 2022, Visual Studio 2022 für Mac, Visual Studio Code und Rider sind namhafte Vertreter. Mit wenigen Schritten kann die bestehende Visual Studio Installation um den .NET MAUI Entwicklungsworkload erweitert werden. Hierzu werden weder Konsolenbefehle noch die manuelle Installation von weiteren Programmen benötigt. Der in Visual Studio enthaltene Android Emulator ermöglicht das Verwalten und Debuggen der simulierten Geräte, ohne die Entwicklungsumgebung zu verlassen. Um die Zielplattformen macOS und iOS zu Debuggen, wird ein Mac benötigt, welcher remote in die Entwicklungsumgebung eingebunden werden kann. Visual Studio ermöglicht ebenfalls Hot Reloading. Durch die Verwendung von Templates kann der Entwicklungsprozess beschleunigt werden [Davidbritch 2022b].

Für MAUI und Blazor als Teile des .NET-Ökosystems gibt es viele Bibliotheken und Erweiterungen von anderen Entwicklern. Diese können unkompliziert über die Paketverwaltung NuGet eingebunden werden. Durch Blazors Interoperabilität zu JavaScript ist es ebenfalls möglich, JavaScript-Bibliotheken einzubinden. Für Sensoren, die weder über die MAUI-API noch über Bibliotheken angesprochen werden können, besteht die Möglichkeit, plattformspezifischen Code aus der gemeinsamen Codebasis aufzurufen. Für diese plattformbedingte Kompilierung werden entsprechende Kenntnisse über die Schnittstellen der Zielplattform benötigt [Davidbritch 2022c].

MAUI vereint plattformspezifische Frameworks in einer einzigen API. Die geteilte Codebasis interagiert überwiegend mit der zielplattformunabhängigen MAUI-API, welche anschließend die spezifischen APIs der Zielplattformen konsumiert. Ein direkter Aufruf von Schnittstellen der Zielplattformen ist ebenfalls möglich. Abhängig von der Zielplattform wird wahlweise Ahead-of-Time (AOT) und Just-in-Time (JIT) Kompilierung ermöglicht [Davidbritch 2022d]. In Abbildung 6 ist die Architektur einer MAUI Applikation dargestellt.

Zusammenfassend kann festgehalten werden, dass die Entwicklung des Frameworks noch

⁸ <https://dotnet.microsoft.com/en-us/learn/maui>

⁹ <https://learn.microsoft.com/de-de/aspnet/core/blazor/hybrid/?view=aspnetcore-6.0>

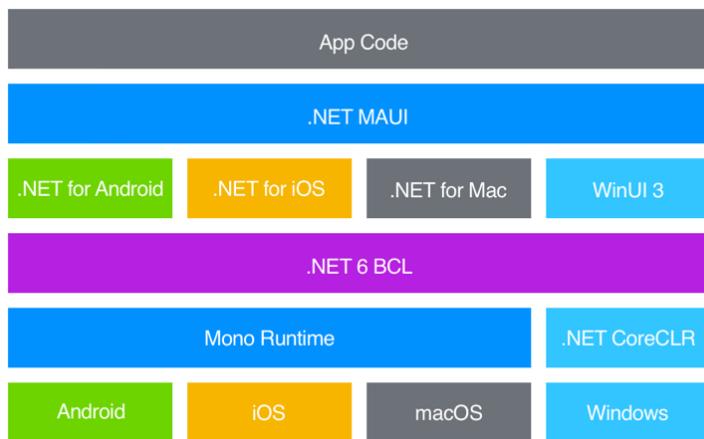


Abb. 6: Übersicht über die Architektur einer MAUI Applikation nach [Davidbritch 2022d]

weiter voran geht und als Nachfolger von Xamarin auch entsprechend lange unterstützt wird. MAUI hat einen großen Funktionsumfang und kann durch zusätzliche Erweiterungen dynamisch ausgebaut werden. Durch die Möglichkeit, XAML oder Blazor einzusetzen, richtet sich MAUI sowohl an Webentwickler*innen, als auch an Entwickler*innen mit Erfahrung in Desktop-Applikationen.

5 Vergleich der Frameworks

Im folgenden Kapitel wird der Vergleich der Frameworks durchgeführt. Dabei wird zunächst jedes Framework anhand der ermittelten Vergleichskriterien bewertet. Zur Bestimmung der Performance findet ein Benchmarking statt, bei dem Berechnungszeiten und CPU-Auslastung für rechenintensive Funktionen dokumentiert werden. Anschließend kann mit den Bewertungen und den zuvor bestimmten Gewichtungen für die Kriterien in jedem definierten Anwendungsgebiet eine Punktzahl errechnet werden, anhand derer die Frameworks sortiert werden können.

5.1 Prototypen

Im Rahmen des Vergleichs werden für jedes Framework drei unterschiedliche Prototypen umgesetzt (vgl. Abschnitt 3.2). Zur Demonstration einer formularlastigen App wird eine To-Do-Listen Applikation umgesetzt. Diese nimmt Aufgaben per Formular auf und speichert sie in einer lokalen Datenbank ab. Außerdem können Aufgaben gelöscht oder als „erledigt“ markiert werden.

Die zweite Applikation ist eine sensorlastige App. Hierbei werden die gängigsten Sensoren über das Framework angesprochen und die Daten ausgelesen. Wichtig sind hierbei Sensoren wie z. B. der Beschleunigungssensor, das Gyroskop oder das Magnetometer.

Um die Performance zu vergleichen, wurden bei der rechenintensiven App eine rekursive und eine iterative Funktion implementiert. Bei der rekursiven Funktion handelt es sich um die Ackermann-Funktion. Diese ist dafür bekannt, sehr stark anzusteigen und die Rechenkapazität schnell zu erschöpfen. Die Ackermann-Funktion ist im Folgenden definiert [Tutego 2015].

$$\begin{aligned} a(0, m) &= m + 1 \\ a(n, 0) &= a(n - 1, 1) \\ a(n, m) &= a(n - 1, a(n, m - 1)) \end{aligned}$$

Als iterative Funktion dient eine Implementierung der Potenzierung, bei der die Berechnung der potenzierten Zahl iterativ erfolgt. Hierbei werden die Basis und der Exponent als Parameter übergeben.

5.2 Benchmarks

Um die rechenintensive App zu testen, werden die implementierten Funktionen (vgl. Kapitel 5.1) ausgeführt und dabei die Zeit gemessen, die die Applikation für die Ausführung bzw. Berechnung benötigt. Außerdem wurde die CPU-Auslastung dokumentiert. Zur Ausführung und Messung wird das Samsung Galaxy S20 FE verwendet.

Bei der Ackermann-Funktion wurden für die Parameter n und m folgende Wertebelegungen gemessen: (n=3, m=8) und (n=3, m=12). Die Messergebnisse sind in Abbildung 7 dargestellt. Bei der Durchführung der Messungen wurde festgestellt, dass Flutter und MAUI nur geringe

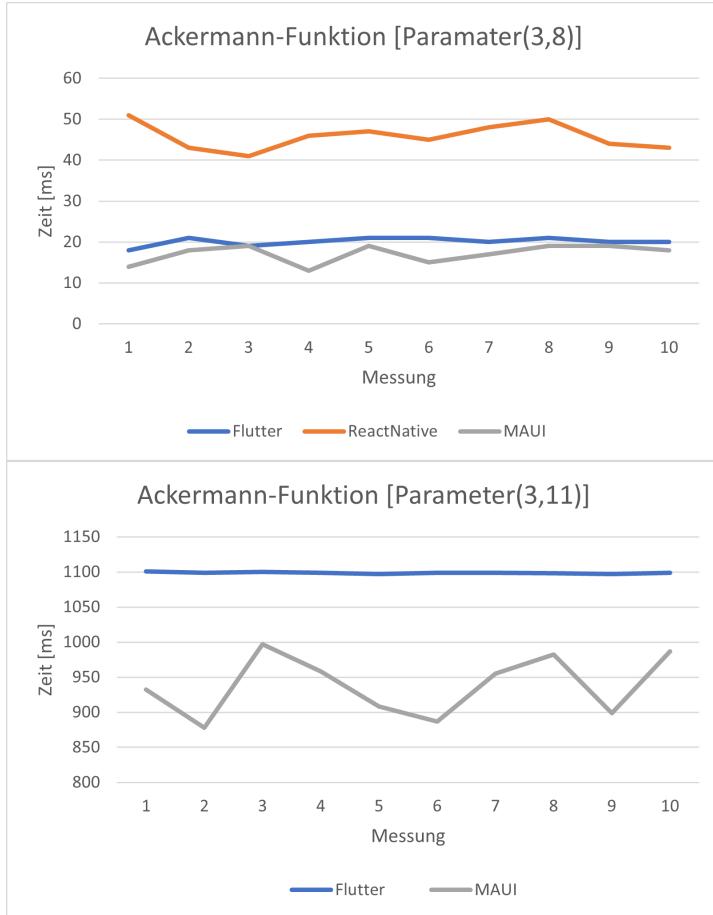


Abb. 7: Analyse der Frameworks anhand der Ackermann-Funktion

Unterschiede bei der Ausführungszeit aufweisen. MAUI berechnet die Ackermann-Funktion bei beiden Varianten etwas schneller als Flutter. Konkret benötigt MAUI jeweils rund 15% weniger Rechenzeit. Deutlich abgeslagen ist in beiden Fällen das dritte Framework React Native. Bei den Parametern (n=3, m=8) benötigte es 2,5x länger als Flutter, um das Ergebnis der Ackermann-Funktion zu berechnen. Bei (n=3, m=12) konnte React Native kein Ergebnis errechnen und brach schon vorher mit einem StackOverflow-Fehler ab.

Bei der iterativen Potenzierung wurde mit der Basis 12 und dem Exponenten 9 gemessen. Die Messergebnisse sind in Abbildung 8 dargestellt. Dabei ist derselbe Trend wie bei der

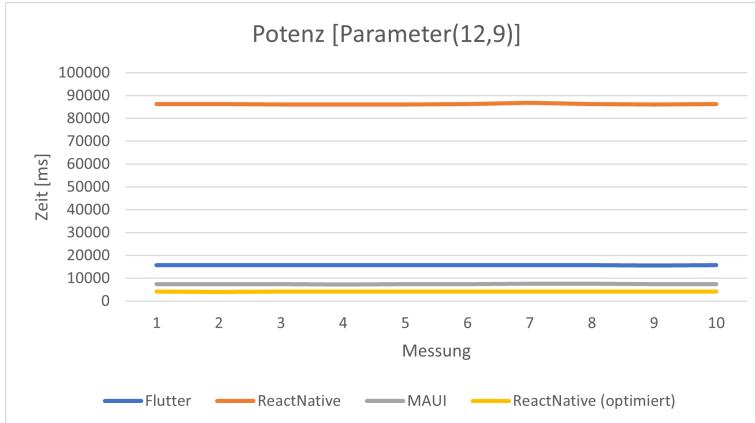


Abb. 8: Analyse der Frameworks anhand der Iterativen Potenzierung

Ackermann-Funktion erkennbar. MAUI hat erneut die höchste Performance und ist konkret 50% schneller als Flutter. Außerdem ist React Native zeitlich erneut weit von den anderen Frameworks entfernt. Es benötigt rund 5,5x länger als Flutter zur Berechnung der Potenz. Durch Ausführung eines nativen Moduls war es möglich, ReactNative zu optimieren und ebenfalls eine hohe Performance zu erreichen.

Die Auslastung der CPU wurde bei einer rechenintensiven Berechnung über einen Zeitraum von 30 Sekunden gemessen. Die Messergebnisse sind in Abbildung 9 dargestellt. Es ist

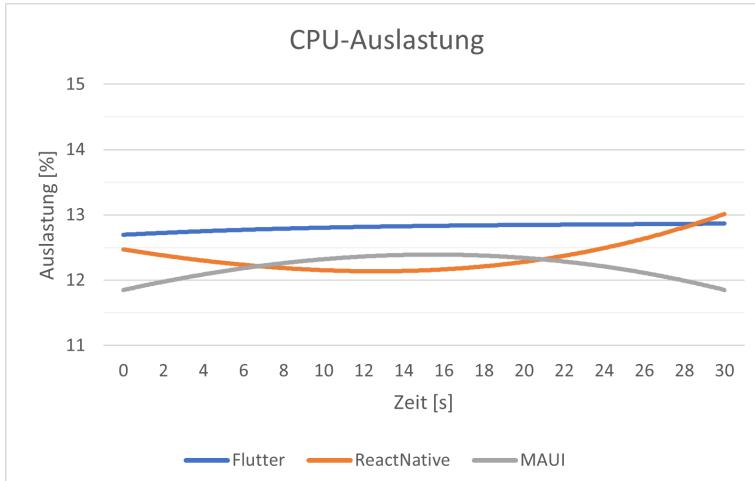


Abb. 9: CPU-Auslastung der Frameworks

sichtbar, dass alle Frameworks annähernd dieselben Auslastung generieren. Konkret lastet

MAUI die CPU mit geringem Abstand am wenigsten aus, während Flutter tendenziell eine leicht höhere Auslastung herbeiführt.

5.3 Bewertung der Frameworks

In diesem Abschnitt werden die Frameworks für die ermittelten Vergleichskriterien bewertet. Jedes Kriterium kann mit minimal 0 Punkten und maximal 5 Punkten bewertet werden. Dabei werden die ermittelten Eigenschaften der Frameworks aus der Literaturarbeit in Kapitel 4 verwendet. Außerdem werden die Ergebnisse des Benchmarkings in Abschnitt 5.2 und die subjektiven Eindrücke bei der Erstellung der Prototypen miteinbezogen.

5.3.1 Flutter

Flutter erhält aufgrund der angegebenen Faktoren die folgenden Bewertungen.

Plattformspezifische Funktionen (5/5) Es ist ein einfacher Zugriff auf Sensoren des Endgeräts gegeben. Außerdem ist eine Möglichkeit zur eigenständigen Implementierung beim Zugriff auf plattformspezifische Schnittstellen vorhanden.

Performance (4/5) Es liegt eine hohe Performance durch u. A. AOT-Kompilierung vor. Da Flutter laut Benchmarking minimal langsamer als MAUI ist, ist ein kleiner Punktabzug notwendig.

Benutzeroberfläche (4/5) Flutter hat ein einzigartiges Design, da keine OEM-Widgets verwendet werden, sondern mit einer eigenen Renderengine gearbeitet wird. Es ist ein kleiner Punktabzug notwendig, da im Gegensatz zu den anderen Frameworks keine Gestaltung per CSS möglich ist.

Erste Schritte (2/5) Es ist aufgrund der vergleichsweise aufwändigen Installation ein starker Punktabzug notwendig, da viele Abhängigkeiten zu anderen Programmen vorliegen. Die Einarbeitung in Dart kann Zeit benötigen, da der Aufbau der Anwendung teilweise nicht intuitiv ist.

Entwicklungsunterstützung (4/5) Es ist eine Unterstützung für viele verschiedene IDEs inklusive Erweiterungen vorhanden. Durch Hot Reload wird eine schnelle Entwicklung ermöglicht. Es ist ein kleiner Punktabzug notwendig, da die Unterstützung beim Anlegen von Widgets besser sein könnte.

Dokumentation (5/5) Die Dokumentation ist online und offline verfügbar und sehr umfangreich. Außerdem sind Videoanleitungen auf Youtube vorhanden.

Ökosystem (4/5) Pakete können einfach über den Pub Package Manager eingebunden werden. Es ist eine umfangreiche Paketauswahl vorhanden. Ein kleiner Punkt abzug ist notwendig, da kein vergleichbar großes Ökosystem wie z. B. bei .NET vorliegt.

Lebenszyklus & öffentliches Interesse (5/5) Flutter ist das meistverwendete, hybrides Framework im Jahr 2021 mit steigenden Tendenzen. Es ist sehr modern und noch am Beginn des Lebenszyklus, da der Release erst 2018 war.

5.3.2 React Native

React Native erhält aufgrund der angegebenen Faktoren die folgenden Bewertungen.

Plattformspezifische Funktionen (5/5) Es ist ein einfacher Zugriff auf Sensoren des Endgeräts gegeben. Außerdem sind Bibliotheken vorhanden, die Dateizugriffe übernehmen und einfach gestalten. Es gibt die Möglichkeit, über Android z. B. eigene Module zu implementieren.

Performance (2/5) Es ist ein starker Punkt abzug notwendig, da React Native eindeutig die geringste Performance der drei Frameworks hat. Die Optimierung durch native Module wiegt dies nicht auf.

Benutzeroberfläche (4/5) Native Widgets des jeweiligen Systems werden verwendet. Das Design erfolgt über spezielles CSS, somit ist die Gestaltung sehr variabel. Ein kleiner Punkt abzug ist notwendig, da keine eigenen, einzigartigen Widgets benutzt werden.

Erste Schritte (3/5) Es ist ein kleiner Punkt abzug notwendig, da die Installation etwas aufwändiger ist. Nach der Installation genügen allerdings einfache Befehle zur Erstellung von Apps.

Entwicklungsunterstützung (4/5) Viele verschiedene IDEs inklusive Erweiterungen werden unterstützt. React Native ist z. B. in der IDE Webstorm von JetBrains bereits bei Installation vorhanden. Es ist ein kleiner Punkt abzug notwendig, da die Unterstützung beim Anlegen von Widgets besser sein könnte.

Dokumentation (5/5) Die Dokumentation ist online umfangreich verfügbar. Außerdem sind zahlreiche Bücher erhältlich.

Ökosystem (4/5) Über NPM können Module einfach per Terminal hinzugefügt werden. Eine umfangreiche Paketauswahl ist ebenfalls vorhanden. Ein kleiner Punkt abzug ist notwendig, da kein vergleichbar großes Ökosystem wie z. B. bei .NET vorliegt.

Lebenszyklus & öffentliches Interesse (5/5) React Native wird für viele bekannte Apps verwendet und durch Meta weiterentwickelt. Das Framework ist ausgereift, da es bereits seit 2015 auf dem Markt ist.

5.3.3 MAUI

MAUI erhält aufgrund der angegebenen Faktoren die folgenden Bewertungen.

Plattformspezifische Funktionen (4/5) Der Zugriff auf Sensoren des Endgeräts ist durch Pakete gegeben. Eine Möglichkeit zur eigenständigen Implementierung in C# beim Zugriff auf plattformspezifische Schnittstellen ist vorhanden. Es ist ein kleiner Punkt abzugrenzen, da der Zugriff auf Sensoren aufwändiger als in den anderen Frameworks ist.

Performance (5/5) MAUI hat u. A. durch die AOT-Kompilierung eine sehr hohe Performance. Es ist minimal schneller als Flutter und deutlich schneller als React Native.

Benutzeroberfläche (4/5) Die Gestaltung der Oberfläche erfolgt über HTML und CSS. Bereits erstellte Webanwendungen können wiederverwendet werden. Ein kleiner Punkt abzugrenzen ist notwendig, da keine eigenen, einzigartigen Widgets benutzt werden.

Erste Schritte (5/5) Die Installation und Einrichtung ist sehr einfach, da Visual Studio den Großteil der Arbeit abnimmt und Abhängigkeiten nachinstalliert. Der Aufbau der Anwendungen ist intuitiv und orientiert sich an anderen Microsoft-Produkten.

Entwicklungsunterstützung (4/5) Unterstützung für viele verschiedene IDEs ist vorhanden. Die Hot Reload-Funktion ermöglicht eine schnelle Entwicklung und funktioniert zuverlässig. Der Android Emulator ist direkt in Visual Studio integriert. Ein kleiner Punkt abzugrenzen ist notwendig, da die Unterstützung beim Anlegen von Widgets besser sein könnte.

Dokumentation (4/5) Die Dokumentation ist online umfangreich verfügbar. Außerdem sind Videoanleitungen und Lernplattformen vorhanden. Es gibt einen kleinen Punkt abzugrenzen, da Verwechslungsgefahr zwischen klassischem MAUI (via XAML) und MAUI Blazor besteht.

Ökosystem (5/5) Der NuGet Package Manager kann zum Einbinden von Paketen entweder im Terminal oder auf einer grafischen Oberfläche verwendet werden. Das .NET-Ökosystem ist sehr umfangreich und es werden viele Anwendungsbereiche unterstützt.

Lebenszyklus & öffentliches Interesse (3/5) MAUI ist ein sehr junges Framework mit großem Interesse im .NET-Umfeld. Es steht am Beginn des Lebenszyklus, da der Release erst 2022 war. Es ist ein kleiner Punkt abzugrenzen, da noch kleinere Fehler bzw. „Kinderkrankheiten“ im Framework vorhanden sind.

5.4 Vergleich und Ergebnisse

Im Folgenden werden die in Abschnitt 5.3 ermittelten Bewertungen der Frameworks unter Berücksichtigung der in Abschnitt 3.3 definierten Gewichtungen der Vergleichskriterien je Anwendungsgebiet miteinander verglichen.

5.4.1 Formularlastige Anwendung

	Gewichtung	React		Flutter		MAUI	
		Bewertung	Wert	Bewertung	Wert	Bewertung	Wert
Plattformspezifische Funktionen	7%	5	0,36	5	0,36	4	0,29
Performance	14%	2	0,29	4	0,57	5	0,71
Benutzeroberfläche	25%	4	1,00	4	1,00	4	1,00
Erste Schritte	4%	3	0,11	2	0,07	5	0,18
Entwicklungsunterstützung	14%	4	0,57	4	0,57	4	0,57
Dokumentation	11%	5	0,54	5	0,54	4	0,43
Ökosystem	4%	4	0,14	4	0,14	5	0,18
Lebenszyklus & öffentliches Interesse	21%	5	1,07	5	1,07	3	0,64
Summe		4,07		4,32		4,00	

Abb. 10: Bewertung für formularlastige Anwendungen

Im Anwendungsgebiet der formularlastigen Anwendungen setzt sich Flutter mit einer Gesamtbewertung von 4,32 gegen React Native mit 4,07 und MAUI mit 4,00 durch. Alle Frameworks erreichen eine Wertung von größer oder gleich 4 Punkten von den maximal erreichbaren 5 Punkten. In dem für dieses Anwendungsgebiet am höchsten gewichteten Vergleichskriterium schneiden alle Frameworks gleich gut ab. Das Bewertungsschema der einzelnen Vergleichskriterien je Anwendungsgebiet und Framework, sowie die Gesamtbewertung eines jeden Frameworks ist in Abbildung 10 dargestellt.

5.4.2 Rechenintensive Anwendung

Im Anwendungsgebiet der rechenintensiven Anwendungen setzt sich Flutter mit einer Gesamtbewertung von 4,36 gegen MAUI mit 4,21 und React Native mit 3,89 durch. Lediglich Flutter und MAUI erreichen eine Wertung von größer oder gleich 4 Punkten von den maximal erreichbaren 5 Punkten. In dem für dieses Anwendungsgebiet am höchsten gewichteten Vergleichskriterium schneidet MAUI am besten ab, dennoch erreicht Flutter eine höhere Gesamtwertung. React Native schneidet bei diesem Vergleichskriterium am schlechtesten ab. Das Bewertungsschema der einzelnen Vergleichskriterien je Anwendungsgebiet und

	Gewichtung	React		Flutter		MAUI	
		Bewertung	Wert	Bewertung	Wert	Bewertung	Wert
Plattformspezifische Funktionen	11%	5	0,54	5	0,54	4	0,43
Performance	25%	2	0,50	4	1,00	5	1,25
Benutzeroberfläche	0%	4	-	4	-	4	-
Erste Schritte	4%	3	0,11	2	0,07	5	0,18
Entwicklungsunterstützung	14%	4	0,57	4	0,57	4	0,57
Dokumentation	11%	5	0,54	5	0,54	4	0,43
Ökosystem	14%	4	0,57	4	0,57	5	0,71
Lebenszyklus & öffentliches Interesse	21%	5	1,07	5	1,07	3	0,64
Summe			3,89		4,36		4,21

Abb. 11: Bewertung für rechenintensive Anwendungen

Framework, sowie die Gesamtbewertung eines jeden Frameworks ist in Abbildung 11 dargestellt.

5.4.3 Sensorlastige Anwendung

	Gewichtung	React		Flutter		MAUI	
		Bewertung	Wert	Bewertung	Wert	Bewertung	Wert
Plattformspezifische Funktionen	25%	5	1,25	5	1,25	4	1,00
Performance	11%	2	0,21	4	0,43	5	0,54
Benutzeroberfläche	0%	4	-	4	-	4	-
Erste Schritte	4%	3	0,11	2	0,07	5	0,18
Entwicklungsunterstützung	14%	4	0,57	4	0,57	4	0,57
Dokumentation	14%	5	0,71	5	0,71	4	0,57
Ökosystem	14%	4	0,57	4	0,57	5	0,71
Lebenszyklus & öffentliches Interesse	18%	5	0,89	5	0,89	3	0,54
Summe			4,32		4,50		4,11

Abb. 12: Bewertung für sensorlastige Anwendungen

Im Anwendungsgebiet der sensorlastigen Anwendungen setzt sich Flutter mit einer Gesamtbewertung von 4,50 gegen React Native mit 4,32 und MAUI mit 4,11 durch. Alle Frameworks erreichen eine Wertung von größer oder gleich 4 Punkten von den maximal erreichbaren 5 Punkten. In dem für dieses Anwendungsgebiet am höchsten gewichteten Vergleichskriterium erhalten Flutter und React Native die maximal erreichbare Punktzahl.

MAUI schneidet beim bei diesem Vergleichskriterium etwas schlechter ab. Das Bewertungsschema der einzelnen Vergleichskriterien je Anwendungsgebiet und Framework, sowie die Gesamtbewertung eines jeden Frameworks ist in Abbildung 12 dargestellt.

5.4.4 Ergebnis

In Abbildung 13 ist das Ergebnis des Vergleichs übersichtlich zusammengefasst. Es ist erkennbar, dass Flutter insgesamt die beste Bewertung erreicht hat. Dahinter folgen mit einem Abstand MAUI und ReactNative.

Anwendungsgebiet \ Framework	Flutter	MAUI	ReactNative
Formularlastig	4,32	4,00	4,07
Rechenintensiv	4,36	4,21	3,89
Sensorlastig	4,50	4,11	4,32
	13,18	12,32	12,28

Abb. 13: Übersicht über Punktzahlen je Framework und Anwendungsgebiet

6 Fazit

Alle drei Frameworks erleichtern die Entwicklung von mobilen Applikationen, indem der Kern der Applikation nur einmal entwickelt werden muss. So kann viel Zeit gespart werden, die sonst in die Entwicklung separater Anwendungen in unterschiedlichen Programmiersprachen für jedes einzelne Betriebssystem investiert werden müsste. Bei allen getesteten Frameworks wird die Erstellung einer Web-App, einer Android-App und einer iOS-App unterstützt. Außerdem ist abhängig vom individuellen Framework die Entwicklung für weitere Systeme möglich. So kann man z. B. mit Maui und React Native Windows-Store-Apps erstellen, während Flutter die Erstellung von Windows Executables ermöglicht.

Alle Frameworks überzeugen durch eine gute, ausführliche Online-Dokumentation. Bei Flutter stellte sich die Einrichtung jedoch als etwas komplizierter heraus, da viele Abhängigkeiten zu anderen Programmen bestehen. Außerdem erlauben alle Frameworks eine große Vielfalt an Gestaltungsmöglichkeiten, da z. B. die Gestaltung durch eine eingeschränkte Form von CSS möglich ist. Sowohl React Native als auch Maui bringen zudem den Vorteil mit sich, dass der Code von bereits existierenden Web-Apps übernommen werden kann. Da hinter allen Frameworks namhafte Firmen wie Microsoft, Google und Meta stehen, kann man davon ausgehen, dass die Frameworks auch in Zukunft weiterentwickelt werden und weiter unterstützt werden.

Bei der Durchführung des Benchmarkings konnte ermittelt werden, dass die Prototypen der Frameworks Flutter und Maui sehr performant sind. Das Framework React Native ist hingegen deutlich weniger performant. Die CPU-Auslastung ist bei allen Frameworks nahezu identisch.

Mit einer formularlastigen Anwendung, einer sensorlastigen Anwendung und einer rechenintensiven Anwendung wurden drei Anwendungsgebiete definiert. Durch die Ermittlung von Vergleichskriterien, einer individuellen Gewichtung je Anwendungsgebiet und einer anschließenden Bewertung konnten je Framework und Anwendungsgebiet Punktzahlen berechnet werden. Dabei konnte sich Flutter in jedem Anwendungsgebiet vor MAUI und React Native durchsetzen. Das abschließende Ergebnis des Vergleichs ist in Abbildung 13 dargestellt. Es ist ersichtlich, dass Flutter insgesamt die beste Bewertung erreicht hat. Auf dem zweiten Platz befindet sich MAUI, auf dem dritten Platz folgt React Native.

Literatur

- [Chandratre 2020] Ruchir Chandratre. „Liste der Dinge, die Sie beachten sollten, bevor Sie mit der Entwicklung mobiler Apps mit React Native beginnen“. In: *Cisin* (28. Jan. 2020). URL: <https://www.cisin.com/coffee-break/de/enterprise/list-of-things-you-should-keep-in-mind-before-you-start-developing-mobile-apps-with-react-native.html> (besucht am 04. 11. 2022).
- [Cowart 2012] Jim Cowart. *What is a Hybrid Mobile App?* Hrsg. von Telerik Blogs. 2012. URL: <https://www.telerik.com/blogs/what-is-a-hybrid-mobile-app-> (besucht am 18. 10. 2022).
- [Dart Documentation 2022] Dart Documentation. *Dart Overview*. 2.11.2022. URL: <https://dart.dev/overview#platform> (besucht am 02. 11. 2022).
- [Davidbritch 2022a] Davidbritch. *Hosten einer Blazor-Web-App in einer .NET MAUI-App mit BlazorWebView - .NET MAUI*. 4.12.2022. URL: <https://learn.microsoft.com/de-de/dotnet/maui/user-interface/controls/blazorwebview?view=net-maui-7.0> (besucht am 04. 12. 2022).
- [Davidbritch 2022b] Davidbritch. *Installieren von Visual Studio 2022 zum Entwickeln plattformübergreifender Apps mit .NET MAUI - .NET MAUI*. 4.12.2022. (Besucht am 04. 12. 2022).
- [Davidbritch 2022c] Davidbritch. *.NET MAUI invoking platform code - .NET MAUI*. 6.11.2022. URL: <https://learn.microsoft.com/en-us/dotnet/maui/platform-integration/invoke-platform-code> (besucht am 06. 11. 2022).
- [Davidbritch 2022d] Davidbritch. *Was ist .NET MAUI? - .NET MAUI*. 4.12.2022. URL: <https://learn.microsoft.com/de-de/dotnet/maui/what-is-maui?view=net-maui-7.0> (besucht am 04. 12. 2022).
- [Denko et al. 2021] Blaž Denko, Špela Pečnik und Iztok Fister Jr. „A Comprehensive Comparison of Hybrid Mobile Application Development Frameworks“. In: *International Journal of Security and Privacy in Pervasive Computing* 13.1 (2021), S. 78–90. ISSN: 2643-7937. DOI: [10.4018/IJSPPC.2021010105](https://doi.org/10.4018/IJSPPC.2021010105).
- [Dharmwan 2021] Subodh Dharmwan. „7 examples of hybrid apps that have taken businesses to the next level“. In: *Cynoteck Technology Solutions* (27. Apr. 2021). URL: <https://cynoteck.com/de/blog-post/hybrid-apps-that-have-taken-businesses-to-the-next-level/> (besucht am 04. 12. 2022).

- [Greenrobot 2021] Greenrobot. *What is the best Flutter Database?* 2021. URL: <https://greenrobot.org/news/flutter-databases-a-comprehensive-comparison/> (besucht am 04. 11. 2022).
- [Helios Blog 2020] Helios Blog. *Flutter vs. React Native: Which one should you opt for in 2020?* 2020. URL: <https://www.heliossolutions.co/blog/flutter-vs-react-native-which-one-should-you-opt-for-in-2020/> (besucht am 02. 11. 2022).
- [Kleinschrod 2020] Bernd Kleinschrod. *Native App vs Web App vs Hybrid App.* Hrsg. von Webraketen. 2020. URL: <https://webraketen.io/app-native-web-hybrid/> (besucht am 19. 10. 2022).
- [Krypczyk et al. 2021] Veikko Krypczyk und Dirk Mittmann. *React Native im Überblick.* 2021. URL: <https://entwickler.de/react/react-native-im-ueberblick> (besucht am 04. 11. 2022).
- [Lestal 2020] Justin Lestal. „React vs React Native: What's the difference?“ In: *Devskiller* (12. Aug. 2020). URL: <https://devskiller.com/react-vs-react-native-whats-the-difference/> (besucht am 04. 11. 2022).
- [Niemeier 2022] Susan Niemeier. *React Native.* 4.11.2022. URL: <https://www.tenmedia.de/de/glossar/react-native> (besucht am 04. 11. 2022).
- [Obinna 2020] Onuoha Obinna. „How does JIT and AOT work in Dart? - Onuoha Obinna - Medium“. In: *Medium* (7. Apr. 2020). URL: <https://onuoha.medium.com/how-does-jit-and-aot-work-in-dart-cab2f31d9cb5> (besucht am 02. 11. 2022).
- [Que et al. 2016] Peixin Que, Xiao Guo und Maokun Zhu. „A Comprehensive Comparison between Hybrid and Native App Paradigms“. In: *2016 8th International Conference on Computational Intelligence and Communication Networks. CICN 2016 : 23-25 December 2016, THDC Institute of Hydropower Engg and Technology, Bhagirathipuram, Tehri, India : proceedings.* 2016 8th International Conference on Computational Intelligence and Communication Networks (CICN) (Tehri, India). Hrsg. von G. S. Tomar. Piscataway, NJ: IEEE, 2016, S. 611–614. ISBN: 978-1-5090-1144-5. DOI: [10.1109/CICN.2016.125](https://doi.org/10.1109/CICN.2016.125).
- [Schmidt 2018] Daniel Schmidt. „Using Sensors in React Native - React Native Training - Medium“. In: *React Native Training* (14. Mai 2018). URL: <https://medium.com/react-native-training/using-sensors-in-react-native-b194d0ad9167> (besucht am 04. 11. 2022).

- [Statista 2021] Statista. *Cross-platform mobile frameworks used by global developers 2021. The State of Developer Ecosystem 2021*. Hrsg. von JetBrains. 2021. URL: <https://www.statista.com/statistics/869224/worldwide-software-developer-working-hours/> (besucht am 21.10.2022).
- [Statista 2022] Statista. *Mobile Betriebssysteme - Marktanteile Internetnutzung weltweit bis September 2022*. Hrsg. von StatCounter. 2022. URL: <https://de.statista.com/statistik/daten/studie/184335/umfrage/marktanteil-der-mobilen-betriebssysteme-weltweit-seit-2009/> (besucht am 06.11.2022).
- [Tutego 2015] Tutego. *Die Ackermann-Funktion*. de. Copyright: Christian Ullenoobom, www.tutego.de. 10.02.2015. URL: <http://www.tutego.de/java/articles/Ackermann-Funktion.html> (besucht am 29.11.2022).
- [Tyshchenko 2020] Andrew Tyshchenko. „Testing Native and Hybrid Mobile Apps. Whats the Difference?“ In: *Medium* (19. Juni 2020). URL: <https://medium.com/@andrew.tishchenko/testing-native-and-hybrid-mobile-apps-whats-the-difference-4ca98a71afc1> (besucht am 05.11.2022).
- [Willnecker et al. 2012] Felix Willnecker, Damir Ismailović und Wolfgang Maison. „Architekturen mobiler Multiplattform-Apps“. In: *Smart mobile apps. Mit Business-Aps ins Zeitalter mobiler Geschäftsprozesse*. Hrsg. von Stephan Verclas und Claudia Linnhoff-Popien. Xpert.press. Dordrecht: Springer, 2012, S. 403–417. ISBN: 978-3-642-22258-0. doi: [10.1007/978-3-642-22259-7_26](https://doi.org/10.1007/978-3-642-22259-7_26).
- [Wu 2018] Wenhao Wu. *React Native vs Flutter, Cross-platforms mobile application frameworks*. 2018. URL: <https://www.thesesus.fi/bitstream/handle/10024/146232/thesis.pdf?sequence=1>.
- [xster 2017] xster. „Why Flutter doesn't use OEM widgets“. In: *Medium* (16. Nov. 2017). URL: <https://medium.com/flutter/why-flutter-doesnt-use-oem-widgets-94746e812510> (besucht am 02.11.2022).
- [Zammetti 2019] Frank W. Zammetti. *Practical flutter. Improve your mobile development with google's latest Open-Source SDK*. eng. Springer eBook Collection. Zammetti, Frank W. (VerfasserIn). Berkeley, CA: Apress, 2019. 396 S. ISBN: 978-1-4842-4971-0. doi: [10.1007/978-1-4842-4972-7](https://doi.org/10.1007/978-1-4842-4972-7).

Untersuchung von Web-Usability Prinzipien und Evaluation zweier Webshops

Alicia Dietrich¹, Nick Dürr², Jan Perthe³

Abstract: Aufgrund der stetig steigenden Anzahl von Webshops gewinnt Web-Usability immer mehr an Bedeutung. Die vorliegende Seminararbeit beschäftigt sich aus diesem Grund mit der Betrachtung von Prinzipien der Web-Usability nach Nielsen und nach DIN EN ISO 9241-151. Diese Ansätze werden anschließend gegenübergestellt. Um die Einhaltung der Prinzipien sicherzustellen, existieren unterschiedliche Vorgehensweisen. Im Rahmen der Arbeit werden die Usability Inspection, der Usability Engineering Lifecycle und die Menschzentrierte Gestaltung vorgestellt und miteinander verglichen. Danach wird die Usability Inspection am Beispiel von zwei verschiedenen Webshops praktisch umgesetzt. Schlussendlich wird die Anwendbarkeit dieser Vorgehensweise im Bezug auf die Webshops verglichen und evaluiert.

Keywords: Web-Usability, E-Commerce, Heuristiken, Usability Engineering, Usability Inspection

1 Einleitung

1.1 Motivation

In der heutigen Informationsgesellschaft stellt das Internet für viele Unternehmen einen der wichtigsten Verkaufskanäle dar. Mithilfe von Webshops können Unternehmen einfach Produkte oder Dienstleistungen bewerben, an Kunden verkaufen und eine breite Menschenmenge erreichen. Maßgeblich für den Erfolg dieser Verkaufsstrategie ist die Anzahl der Kunden im Verhältnis zur Anzahl der Interessenten. Dieses Verhältnis hängt neben der Beliebtheit der Produkte und des Unternehmens davon ab, wie der Webshop gestaltet ist. Ein ansprechend gestalteter Webshop zieht deutlich mehr Interessenten an. Wichtig ist dabei letztlich nicht nur das Aussehen, sondern auch die Bedienbarkeit (Usability) der Oberfläche. In einer Umfrage von Statista zum Besuch von Online-Fashion-Shops gaben 44 Prozent der Nutzer an, dass Usability ein Kaufs entscheidender Faktor ist. Weitere 41 Prozent gaben sogar an, dass Usability wesentlich ist für die Entscheidung den Webshop zu besuchen oder nicht

¹ DHBW Stuttgart Campus Horb, Informatik, Florianstr. 15, 72160 Horb am Neckar, Deutschland, i20005@hb-dhbw-stuttgart.de

² DHBW Stuttgart Campus Horb, Informatik, Florianstr. 15, 72160 Horb am Neckar, Deutschland, i20008@hb-dhbw-stuttgart.de

³ DHBW Stuttgart Campus Horb, Informatik, Florianstr. 15, 72160 Horb am Neckar, Deutschland, i20027@hb-dhbw-stuttgart.de

In einer Umfrage von Statista zum Besuch von Online-Fashion-Shops gaben 44 Prozent der Nutzer an, dass Usability ein Kaufs entscheidender Faktor ist. Weitere 41 Prozent gaben sogar an, dass Usability wesentlich ist für die Entscheidung den Webshop zu besuchen oder nicht. Lediglich 15 Prozent gaben an Usability sei nicht relevant.[St22] Beide Faktoren, Design und Usability, sind somit wichtig, um sich gegenüber der breiten Menge an Webshops abzuheben und den Interessenten zu einem erfolgreichen Abschluss des Kaufes oder gar zum Besuch des Webshops zu überzeugen. Doch welche Faktoren beinhalten eine „gute“ Usability und wie kann diese am eigenen Webshop überprüft werden? Diese Frage soll unter anderem im Rahmen der Arbeit geklärt werden.

1.2 Zielsetzung

Diese Arbeit soll die Frage klären, welche Usability Heuristiken es gibt und wie diese praktisch umgesetzt werden können. Diese Fragen werden beispielhaft an einem Webshop beantwortet.

Als Ergebnis dieser Arbeit wird ein Katalog an Heuristiken erarbeitet. Dabei werden verschiedene Heuristiken aus der Literatur gegenübergestellt und analysiert welche Heuristiken besonders relevant für das Web sind.

Daraufhin wird beschrieben, wie die Heuristiken in diesem Katalog während der Entwicklung einer Webseite praktisch umgesetzt werden können. Dabei wird vor allem auf den Usability Engineering Lifecycle, Usability Inspections und die menschenzentrierte Gestaltung eingegangen. Auch werden konkrete Methoden und Hilfsmittel zur Umsetzung des Lifecycles betrachtet.

Schlussendlich wird dieses Vorgehen an zwei verschiedenen Webshops beispielhaft aufgezeigt. Dabei wird überprüft, wie gut das Vorgehen und die Heuristiken auf die verschiedene Arten von Webshops anwendbar sind.

2 Grundlagen der Web-Usability

2.1 Was ist Web-Usability?

Um den Begriff der Web-Usability einzuführen, ist es notwendig den Begriff Usability zu betrachten. Usability wird von [Ja22] wie folgt definiert: Usability beschreibt ein Attribut, das sich darauf bezieht, wie "leichtöder einfach etwas zu verwenden ist. Es bezieht sich also darauf, wie schnell ein Nutzer lernen kann eine Software zu verwenden, wie effizient die Nutzung dieser ist und wie gut es im Gedächtnis bleibt. Außerdem wird betrachtet, wie fehleranfällig sie ist und wie viele Nutzer sie gerne verwenden.[Ja22]

Bei der Web-Usability steht die Funktion bzw. die Funktionalität der Website im Fokus. Es wird definiert, welchem Zweck die Website erfüllen soll. Die Web-Usability sorgt dafür,

dass ein Benutzer eine Website optimal bedienen kann und zudem noch ansprechend und sinnvoll gestaltet ist. Eine Website soll für den Benutzer bezüglich der Bedienung und der Funktionalität möglichst attraktiv sein.

Nach [DId] wird Usability (zu Deutsch: Gebrauchstauglichkeit) wie folgt definiert: „Ausmaß, in dem ein System, ein Produkt oder eine Dienstleistung durch bestimmte Benutzer in einem bestimmten Nutzungskontext genutzt werden kann, um bestimmte Ziele effektiv, effizient und zufriedenstellend zu erreichen“.

Um die Usability einer Website einzuhalten, wurden nach ISO und Nielsen verschiedene Prinzipien definiert, die erfüllt werden müssen, um ein hohes Maß an Usability zu erreichen. Diese Prinzipien werden im Folgenden definiert.

2.2 Heuristiken der Web-Usability nach Nielsen

Nielsen definiert in [Ni94a] verschiedene Prinzipien, nach denen eine Website oder Weboberfläche gestaltet werden sollte.

2.2.1 Ästhetisches und minimalistisches Design

Das erste Prinzip welches Nielsen beschreibt, sagt aus, dass eine Oberfläche möglichst einfach und simpel gehalten werden sollte. Jedes zusätzliche Feature muss vom Nutzer erlernt werden und jedes weitere Feature hat die Möglichkeit missverstanden zu werden. Die Oberfläche sollte dabei möglichst dem natürlichen Verhalten des Nutzer beim Lösen einer Aufgabe entsprechen. Das bedeutet es wäre ideal, wenn zu jedem Zeitpunkt die für den Nutzer benötigten Informationen am passenden Ort angezeigt werden und die Folge der Aktionen und Objekte der Abfolge des Nutzer entspricht.

Informationen die zusammen verwendet werden, sollten auch nah beieinander platziert werden. Durch graphische Elemente lassen sich Informationen gruppieren bzw. voneinander trennen. Hierfür können z.B. Linien oder farbliche Abhebungen verwendet werden.

Beim Design einer Oberfläche sollte auch der natürliche Lesefluss des Nutzers beachtet werden. Informationen, die weiter oben und links platziert werden, bekommen am meisten und meistens als erstes die Aufmerksamkeit des Nutzers.

Für das Design einer Website werden drei wichtige Punkte festgelegt:

- Don ‘t over do it: Ein Website sollte nicht aus vielen kontrastreichen und sättigungsreichen Farben bestehen, sondern lieber aus schlichten Farben und einer geringeren Anzahl. Es sollten insgesamt fünf bis sieben verschiedene Farben verwendet werden. Durch eine geringere Anzahl an Farben kann der Nutzer sich besser erinnern.
- Die Oberfläche sollte auch ohne Farbe genutzt werden können. Dies ist vor allem für farbenblinde Menschen wichtig. Jede Farbinformation sollte zusätzlich durch Icons

symbolisiert werden, sodass eine Oberfläche auch ohne Farben interpretiert werden kann.

- Als letzten Punkt sollte beachtet werden, dass eine Farbe nicht zur Informationsbereitstellung genutzt werden sollte, sondern lediglich, um Informationen zu kategorisieren, zu differenzieren und um zu highlighten

Das Design der Oberfläche soll nach dem Motto „weniger ist mehr“ erstellt werden. Es sollte differenziert werden, welche Informationen wirklich essenziell notwendig sind und welche eventuell in zusätzliche Szenen, wie Hilfsdialoge usw., ausgelagert werden können. Am besten wäre es alle wichtige Informationen innerhalb eines Single Screens darzustellen und bei Bedarf weiterer Informationen weitere Elemente zur Verfügung zu stellen. Das Prinzip von weniger ist mehr gilt nicht nur für das Design der Oberfläche, sondern auch für die Funktionen und Mechanismen, die eine Website zur Verfügung stellt. Dem Nutzer sollen nicht unnötig viele Optionen angeboten werden, dies führt zu Fehlern und zur Verwirrung. Außerdem bedeutet jedes (unnötige) Feature, dass zu einer Software hinzugefügt wird, eine weitere Sache die der Nutzer lernen muss. Dadurch wird die Komplexität einer Software unnötig „aufgeblättert“.

2.2.2 Übereinstimmung von System und realer Welt

In diesem Prinzip wird auf Terminologie der Anwendung eingegangen. Die Terminologie der Nutzeroberfläche soll der Sprache des Nutzer entsprechen. Dialoge sollten, wenn möglich, in der nativen Sprache des Nutzer formuliert sein, damit der Nutzer sie gut verstehen kann. Wie beim Design sollen zur Symbolisierung der Nachricht zusätzlich Icons verwendet werden. Innerhalb der Wortwahl sollte darauf geachtet werden, mehrdeutige Wörter möglichst zu vermeiden, um Missverständnisse des Nutzers zu verhindern. Eine Oberfläche sollte trotzdem nicht nur auf ein paar häufig verwendete Wörter beschränkt sein, hat die Community ihre eigene Fachterminologie, so sollten lieber diese verwendet werden. Alle Interaktionen die den Nutzer beinhalten, sollten aus Sicht des Nutzer formuliert sein (z.B. „Sie haben 100 Stück von xyz erworben.“).

Insgesamt führt eine besser Terminologie der Oberfläche zu einem besseren Verständnis beim Nutzer und es fällt ihm einfacher die Oberfläche zu bedienen. Dies steigert die Effizienz im Umgang mit der Anwendung. Um dies weiter zu unterstützen, sollte ein Nutzer für Begriffe Aliase definieren können, die vom System definierte Begriffe ersetzt. Ein System sollte außerdem mitlernen können und sich z.B. Falscheingaben des Nutzers merken und trotzdem passende Ergebnisse zum eigentlich gemeinten Befehl anzeigen.

2.2.3 Erkennen vor Erinnern

In diesem Prinzip geht es um die Unterstützung des Systems gegenüber dem Nutzer. Computer können sich, im Gegensatz zu uns Menschen, sehr genau an Dinge erinnern, während wir Menschen eher gut darin sind Dinge wiederzuerkennen. Deshalb sollten Menüs eingeführt werden, die dem Nutzer helfen sich innerhalb einer komplexen Anwendung zurechtzufinden.

Die Anwendung sollte den Nutzer darin unterstützen, sich an verschiedene Informationen zu erinnern. Dies kann umgesetzt werden, indem z.B. beim Bearbeiten eines Eintrags alle Felder mit den aktuell bestehenden Informationen zum Eintrag vorausgefüllt werden. Der Nutzer muss sich so nicht an alle Informationen zurückerinnern. Um Fehler zu vermeiden, sollte die Software bei der Eingabe von Informationen immer auf die gewünschte Form hinweisen, um später auftretende Fehler zu verhindern.

Ein System sollte außerdem auf einer kleinen Menge an Regeln basieren, die der Nutzer lernen muss. Je mehr Regeln der Nutzer lernen muss, desto schwieriger ist es für ihn, sich an alle zu erinnern. Es muss ein gutes Mittelmaß gefunden werden, wie viele Regeln ein System besitzt, denn ganz ohne Regeln kommt ein System auch nicht aus. Dann ist es schwierig für den Nutzer das Verhalten verschiedener Elemente zu verstehen bzw. vorherzusagen.

2.2.4 Konsistenz und Standardisierung

Konsistenz innerhalb der Usability bedeutet, dass gleich ausgeführten Aktionen zu identischen Ergebnissen führen sollten. Die Software soll immer das gleiche Verhalten und die gleiche Reaktion auf identisch ausgeführte Aktionen zeigen. Durch Konsistenz erhält der Nutzer mehr Sicherheit im Umgang mit der Software und neue Dinge können schneller erlernt werden.

Zudem sollten gleiche Informationen immer an gleichen Stellen zu finden sein und mit gleicher Formatierung. Informationen die sich aufeinander beziehen sollten auch in der gleichen Reihenfolge vorzufinden sein. Als Beispiel sei eine Tabelle gegeben mit Daten aus dem Jahr und Vorjahr. Die Zusatzinformation, die zu dieser Tabelle formuliert wird, soll sich dann ebenfalls in der Reihenfolge Vorjahr und Jahr befinden und nicht umgekehrt. Konsistenz sollte auch bei den Aufgaben eines Systems beachtet werden.

2.2.5 Sichtbarkeit des Systemstatus

Nach diesem Prinzip soll ein System den Nutzer periodisch darüber informieren, was es macht und wie es den Input des Nutzers interpretiert. Das System sollte nicht nur negatives Feedback geben, sondern auch positives. Dies soll verhindern, dass der Nutzer erst Feedback erhält, wenn ein Error aufgetreten ist. Das System soll dem Nutzer außerdem ein Feedback geben, was es mit dem Input des Nutzers machen wird. Es gibt drei verschiedene

Typen von Feedback: kurzfristige Nachrichten, die nur kurz informieren und danach wieder verschwinden, mittelange Nachrichten, die so lange auf dem Bildschirm bleiben sollen, bis der Nutzer sie liest und dauerhafte Nachrichten, die so wichtig sind, dass sie dauerhaft angezeigt werden.

Feedback wird besonders dann benötigt, wenn Aktionen eine lange Antwortzeit haben. Antwortzeiten sollten so schnell wie möglich sein, denn schon ab einer Sekunde Reaktionszeit bemerkt der Nutzer die Unterbrechungen des Systems. Maximal zehn Sekunden kann die Aufmerksamkeit des Nutzers auf dem Bildschirm behalten werden. Bei langen Ladezeiten sollte der Nutzer deshalb Feedback erhalten in Form von Ladebalken mit Prozentangaben, bei kurzen Ladezeiten in Form von Spinning Wheels oder Ähnlichem. Sie zeigen dem Nutzer, dass das System noch arbeitet und noch nicht gesrasht ist.

Bei Systemfehlern sollte ebenfalls ein informatives Feedback über den Fehler gemacht werden. Das schlimmste ist, wenn der Nutzer keine Nachricht erhält, wenn das System gesrasht ist oder eine, bei der er den Fehler erraten muss. Die Fehlermeldung sollte genaue Informationen enthalten, wo das System unterbrochen wurde und warum und ob der Fehler nur lokal auftritt oder das ganze System betrifft.

2.2.6 Benutzerkontrolle und Freiheit

Der Nutzer sollte das Gefühl bekommen, die Kontrolle über den PC bzw. die Anwendung zu haben und nicht umgekehrt. Das System sollte hierzu einfache Wege anbieten, um aus verschiedenen Situationen zu entkommen. Alle Dialoge und Systemstatus sollten über einen Exit-Button verfügen. In manchen Situationen sollte auch ein undo möglich sein. Dies ist vor allem sinnvoll, wenn der Nutzer nur eine bestimmte Aktion rückgängig machen will. Während Ladezeiten sollte der Nutzer die Möglichkeit haben den Prozess abzubrechen bzw. zu unterbrechen. Dadurch erhält der Benutzer mehr Kontrolle und er traut sich mehr Funktionen auszuprobieren, wenn diese abgebrochen oder rückgängig gemacht werden können.

2.2.7 Vermeiden von Fehlern

Falls während der Nutzung ein Fehler auftritt, sollte der Nutzer eine passende Error-Nachricht erhalten. Nielsen gibt vier Regeln an, nach denen diese Nachrichten erstellt werden sollten:

1. Sie sollten klar und verständlich formuliert sein. Der Nutzer sollte die Nachricht selbst verstehen können.
2. Sie sollten präzise formuliert sein, passend zur Situation und nicht generalisiert.
3. Die Nachricht sollte dem Nutzer helfen, das Problem zu lösen.
4. Sie sollte den Nutzer nicht anschuldigen und Wörter wie fatal oder illegal vermeiden.

Besser als nur gute Error-Nachrichten ist es Fehler zu vermeiden und im System abzufangen. Die Errors können dabei durch ihre Häufigkeit oder Frequenz, durch Testen oder Logging Errors gefunden werden. Bei gefährlichen Aktionen ist es hilfreich, wenn vor Ausführung der Aktion eine zusätzliche Bestätigungsanfrage angezeigt wird.

2.2.8 Hilfe und Dokumentation

Die Heuristik "Hilfe und Dokumentation" ist eine der wichtigsten Heuristiken zur Benutzer-freundlichkeit, da sie ein breites Spektrum potenzieller Probleme abdeckt, die auftreten können, wenn Benutzer versuchen, ein Produkt oder eine Dienstleistung zu nutzen. Indem sichergestellt wird, dass die Dokumentation klar und hilfreich ist, können Designer einen großen Beitrag dazu leisten, ihre Produkte benutzerfreundlicher zu gestalten.

Einige der spezifischen Aspekte, die unter die Heuristik "Hilfe und Dokumentation" fallen, sind:

- sicherstellen, dass alle relevanten Informationen in der Dokumentation enthalten sind und dass diese so aufgebaut ist, dass sie leicht zu verstehen und zu benutzen ist.
- sicherstellen, dass die Dokumentation für die Benutzer zugänglich ist, wenn sie sie benötigen, unabhängig davon, ob sie online oder an einem physischen Ort bereitgestellt wird.
- sicherstellen, dass die Dokumentation auf dem neuesten Stand gehalten wird, wenn sich das Produkt oder die Dienstleistung weiterentwickelt.
- Bereitstellung von Kontaktinformationen für Benutzer, die mehr Hilfe benötigen, als die Dokumentation bieten kann.

2.3 Heuristiken der Web-Usability nach ISO

Im Folgenden werden die Usability-Heuristiken von [DIB] betrachtet.

2.3.1 Allgemeines

Im Gegensatz zu konventionellen Anwendungen, umfassen die Webanwendungen eine breite Zielgruppe. Diese breite Menge an Nutzern hat viele verschiedene Informationsbedürfnisse und Interessen. Deshalb sollte der Verwendungszweck, sowie die Interessenvertreter einer Webanwendung klar festgelegt werden. Für den Nutzer selbst sollte der Verwendungszweck klar erkennbar sein.

Die Prinzipien nach ISO unterteilen sich grob in drei Bereiche:

- Gestaltung des Inhalts
- Navigation und Suche
- Darstellung des Inhalts
- Allgemeine Gestaltungsaspekte

2.3.2 Gestaltung des Inhalts

Für die Erstellung einer Web-Benutzerschnittstelle sollte zuvor ein geeignetes konzeptuelles Modell erstellt werden, gemäß den Aufgaben und mentalen Modellen des Nutzers. Dafür können eine Vielzahl an verschiedenen Techniken verwendet werden, wie z.B. UML-Diagramme, Topic Maps usw. Sie verhelfen dazu dem Informationsbedarf des Nutzers gemäß seiner Aufgabe gerecht zu werden. Die Struktur der Seite und die Anordnung der Informationen sollte dem mentalen Modell des Nutzers entsprechen. Zum Beispiel auf einer Nachrichtenwebsite werden auf der Startseite alle Nachrichten im Überblick angezeigt und mittels Verlinkungen auf die zugehörigen detaillierten Berichte verlinkt. Der Nutzer kann sich schnell einen Überblick verschaffen über alle relevanten Themen und bei Interesse genauere Details erhalten.

Auswahl der Medienformen

Inhaltliche Objekte werden basierend auf dem konzeptuellen Inhaltsmodell in Form von Text, Graphiken, Animationen oder anderer Medienformen entwickelt. Objekte können nicht-interaktiv sein, lediglich der Übertragung von Informationen zum Benutzer dienen, oder interaktiv sein, indem sie es den Benutzern erlauben, Eingaben zu tätigen und die Funktionalität der Web-Anwendung zu nutzen. Inhalte werden so aufbereitet, dass Darstellung und Strukturierung auf einfache Weise den wechselnden und dynamischen Nutzeranforderungen angepasst werden können und die Bereitstellung in verschiedenen Kontexten ermöglicht werden können.

Für die Übermittlung der Informationen an den Nutzer sollten geeignete Medienobjekte ausgewählt werden. Für fortlaufende Informationen bieten sich besonders Videos an, während sich in anderen Fällen Schaubilder oder informative Texte besser eignen. Eine Kombination verschiedener Medien ist von besonderer Bedeutung für das Verständnis des Nutzers. Wenn innerhalb eines Mediums Animationen oder bewegter Text dargestellt werden, sollte es für den Nutzer die Möglichkeit geben, diese zu pausieren. Es sollte außerdem ein Online-Feedback Mechanismus zur Verfügung gestellt werden, um Fragen, Kommentare oder Bewertungen zu den angebotenen Inhalten oder Produkten mitzuteilen.

Verarbeitung personenbezogener Daten

Werden innerhalb einer Website personenbezogene Daten bezogen, sollte eine klare und leicht verständliche Datenschutzerklärung bereitgestellt werden, Folgende Informationen sind üblicherweise in einer Erklärung zum Datenschutz enthalten: 1. welche Daten erhoben

bzw. ermittelt werden; 2. in welcher Form die Daten genutzt werden; 3. wem die Daten zur Nutzung überlassen werden.

Anpassungen des Nutzers

Ein weiteres Prinzip, dass in ISO angesprochen wird, ist die individuelle Gestaltung und Anpassung des Benutzers. Ein Nutzer sollte bei der Angabe persönlicher Daten aktiv zustimmen, ob und in welcher Form die Daten verwendet werden. Ein weiterer wichtiger Punkt ist, dass der Nutzer Inhalte und Navigation nach seinen Wünschen anpassen kann und die Website somit effizienter benutzen kann. Es sollte die Möglichkeit geben, verschiedenen Nutzergruppen verschiedene Rollen zu verteilen. Je nach Rolle werden Inhalte unterschiedlich angezeigt. Wenn eine individuelle Gestaltung oder eine Ansicht in einer bestimmten Rolle oder einem Profil erfolgen, sollte dies dem Nutzer kenntlich gemacht werden. Nutzer sollten außerdem Profile ändern oder löschen können. Bei automatischer Benutzerprofilerstellung sollte dem Nutzer kenntlich gemacht werden, welche Art von Daten verwendet werden und wie diese die Nutzung der App beeinflussen.

2.3.3 Navigation und Suche

Navigation

Die Navigation einer Website soll dem Nutzer helfen, sich zurechtzufinden und festzustellen, wo er sich befindet, wo er bereits war und was er noch besuchen kann. Die Navigation sollte übersichtlich sein und hinreichend deutlich machen, wo sich der Nutzer in der Navigationsstruktur befindet und welche Position der aktuelle Abschnitt in der Struktur einnimmt. Der Abschnitt sollte eine klare Position innerhalb der Gesamtstruktur haben. Dies ist besonders wichtig, wenn der Nutzer über eine Suche auf die Seite gelangt, da ihm dann der Kontext fehlt, in dem er sich befindet. Bei der Erstellung einer Navigationsstruktur sollten die verschiedenen Verhaltensweisen berücksichtigt werden (einmal nur irgendwie navigieren oder gezielte Navigation). Gezielte Navigation bedeutet, dass der Nutzer rational entscheidet, welchen Weg er wählt, sich den gewählten Weg merkt und bei Bedarf zurückgeht. Heuristische Navigation hingegen ist ein eher intuitiver Ansatz, bei dem der Benutzer einfach weitergeht, ohne zu planen. Daher ist es wichtig, alternative Navigationspfade anzubieten, um verschiedene Strategien zu unterstützen. Darüber hinaus sollte die Navigation so einfach wie möglich gehalten werden, um den Aufwand für den Benutzer zu minimieren.

Die Navigationsstruktur bestimmt alle möglichen Pfade, auf denen sich ein Benutzer bewegen kann. Die Navigation einer Website sollte auf der Grundlage des Inhalts und der Aufgaben des Nutzers ausgewählt werden. Komplexere Navigationsstrukturen sollten breit gefächert sein, mit mehreren Links auf einer Seite. Die einzelnen Verknüpfungen sollten logisch gruppiert und sinnvoll benannt sein. Eine breit angelegte Navigationsstruktur ist gegenüber einer tief verzweigten zu bevorzugen. Die Navigationsstruktur sollte die Aufgaben des Nutzers unterstützen, z. B. den Kauf eines Produkts, und den Nutzer bei

mehrstufigen Aufgaben durch eine Abfolge von Seiten führen. Häufige Aufgaben können durch Quicklinks (Schnellzugriffe) unterstützt werden.

Die Navigationsstruktur sollte mit dem Inhalt konsistent sein und im Frameset (im Rahmen der Seite) platziert werden. Es sollten mehrere Navigationsebenen angezeigt werden und wenn die Struktur zu tief verzweigt ist, können mehrere Navigationsübersichten angeboten werden (häufig bei Online-Shops mit vielen verschiedenen Kategorien der Fall). Die Navigationsstruktur sollte immer sichtbar oder leicht einsehbar sein und eine Sitemap (Übersicht über die gesamte Struktur), Querverweise und eine „Schritt zurück“-Funktion bieten. Große Seiten sollten in sinnvolle Abschnitte unterteilt werden. Fehlerhafte oder tote Verknüpfungen sollten vermieden werden.

Suche

Die Suche ist wichtig, um den Zugang zu relevanten Informationen effektiv zu gestalten. Wenn die Suchbegriffe bekannt sind, könnte die Suche der Navigation vorgezogen werden. Sie sollte immer dann angeboten werden, wenn die Website nicht mit vertretbarem Aufwand erkundet werden kann. Dann kann die Suche effizienter sein als die Navigation. Die Suche sollte den Zielen und Erfahrungen des Nutzers angepasst sein. Die Suche sollte von allen Seiten der Website aus global zugänglich sein. Das Feld sollte groß genug auch für lange Eingaben sein. Es sollte auch ein Shortcut zur Aktivierung angeboten werden. Die Suche soll fehlertolerant sein und trotz ungenauer oder falsch eingegebener Suchbegriffe brauchbare Ergebnisse liefern.

Die Suchergebnisse sollten in einer für den Nutzer nachvollziehbaren Form angeordnet werden, nach Abhängigkeit seiner Informationsbedürfnisse. Die Ergebnisse sollten nach Relevanz klassifiziert werden und falls es ein vorausgewähltes Klassifikationsschema gibt, sollte dies dem Nutzer angezeigt werden. Ergebnisse der Suche sollten außerdem in ausreichender Form beschrieben werden, damit ein Nutzer dessen Relevanz einschätzen kann. Es sollten außerdem verschiedene Möglichkeiten angeboten werden die Ergebnisse zu sortieren. Die Anzahl der Treffer sollte dem Nutzer ebenfalls angezeigt werden.

Bei der Verwendung von Suchfunktionen ist es wichtig, anzugeben, welche Bereiche durchsucht wurden (z.B. nur die Website oder das ganze Internet). Es sollte möglich sein, den Suchbereich auszuwählen, um die Ergebnisse einzuschränken. Außerdem sollte ein einheitliches Verfahren für den Umgang mit großen Treffermengen angeboten werden, um die Suche effizienter zu gestalten (z.B. das Aufteilen auf mehrere Seiten). Wenn die Treffermenge sehr groß ist, sollte der Benutzer die Möglichkeit haben, die Suche zu verfeinern. Bei der Ergebnisanzeige der Suche sollte auch immer der zuvor eingegebene Suchbegriff aufgelistet sein. Im Falle einer erfolglosen Suche sollten dem Nutzer Hinweise zur besseren Eingabe von Suchbegriffen gegeben werden. Bei der Anzeige der Suchergebnisse sollte dem Benutzer die Möglichkeit gegeben werden, eine neue Suche mit einer geänderten Abfrage durchzuführen. Die Bereitstellung eines Suchverlaufes kann auch hilfreich sein.

2.3.4 Darstellung des Inhalts

Aspekte der Seitengestaltung

Jede Seite einer Website sollte einen erklärenden Titel enthalten, ggf. auch Eigentümer und Datum. Jede Seite innerhalb der Website soll möglichst nach einem einheitlichen Layout gestaltet sein, was dem Nutzer dabei hilft ähnliche oder gleiche Informationen an gleicher Stelle wiederzufinden. Bei unterschiedlichen Kategorien können sich die Layouts innerhalb der Kategorie unterscheiden.

Mit Farben sollte sorgfältig umgegangen werden. Es sollten maximal zehn Farben verwendet werden. Schwierige Farbkombinationen sollten dabei vermieden werden. Unter Berücksichtigung von menschlichen Fähigkeiten und Einschränkungen von Farbwahrnehmung sollten Farben mit Sorgfalt eingesetzt werden und nicht als einzige Form der Informationsvermittlung.

Wenn neue Inhalte hinzukommen oder alte Inhalte geändert werden, sollte dies deutlich gemacht werden. Die Länge der Seite sollte angemessen gewählt werden, und das vertikale und horizontale Scrollen sollte auf ein Minimum reduziert werden. Vor allem für Start-, Navigations- und Übersichtsseiten sollte eine entsprechend kürzere Länge gewählt werden, um eine Übersichtlichkeit zu gewährleisten. Längere Seiten sollte für Inhalte gewählt werden, bei denen es relevant ist, länger ohne Unterbrechungen zu lesen.

Das Seitenlayout einer Website sollte an verschiedene Größen anpassbar sein. Die Zugehörigkeit zu einer Website sollte auf den verschiedenen Unterseiten einer Website deutlich erkennbar sein. Ist ein Dokument sehr lang, verteilt und auf dem Bildschirm schwer zu lesen, sollte eine Druckversion des Dokuments bereitgestellt werden.

Verknüpfungen

Verknüpfungen sind ein wichtiger Bestandteil jeder Website oder Anwendung, da sie den Benutzern die Möglichkeit bieten, zu verschiedenen Seiten zu navigieren oder verschiedene Aktionen durchzuführen. Daher ist es wichtig, dass sie so gestaltet sind, dass sie leicht zu verstehen und zu benutzen sind.

Einige Tipps für die Gestaltung von Verknüpfungen sind:

- Sie können auf unterschiedliche Weise dargestellt werden (als Schaltflächen oder als Text)
- Verknüpfungen sollten angemessen gekennzeichnet werden (durch Hervorhebung des Textes oder Platzierung von Verknüpfungen)
- Benachbarte Verknüpfungen, Navigations- und Transaktionsflächen und Navigations- und Steuerungsflächen sollten voneinander unterschieden werden
- Verknüpfungen sollten selbsterklärend sein
- Unterscheidung von externen Verknüpfungen(die von der Webseite zu einer anderen führen) und internen Verknüpfungen (die innerhalb der Webseite navigieren)

- Es sollten passende Bezeichnungen für Verknüpfungen gewählt werden, die die Nutzer aufgrund ihres Allgemeinwissens, aufgrund früherer Erfahrungen in der Anwendungsdomäne oder aufgrund von Erfahrungen mit anderen Systemen kennt.
- Hervorhebung von Verknüpfungen, die bereits besucht wurden

Die Vermeidung von Redundanzen (doppelte Verknüpfungen innerhalb einer Seite) ist ebenfalls wichtig, um ein gutes Nutzererlebnis zu gewährleisten.

Interaktionsobjekte

Die Interaktionsobjekte sollten entsprechend den erwarteten Eingaben ausgewählt werden und werden durch die folgenden Eigenschaften bestimmt: der Art der Eingabe, die möglichen Eingabewerte (vorgegeben oder frei wählbar), der Art des Eingabewertes und der Anzahl der zu wählenden Elemente. Interaktionsobjekte sollten leicht zu erkennen und zu verstehen sein. Für wichtige Interaktionen sollten zudem Tastaturkürzel bereitgestellt werden. Der angezeigte Text sollte gut lesbar sein (in räumlicher Anordnung und der Anzeigeeinheit). Um das Überfliegen des Textes zu unterstützen, sollten Schlüsselwörter, Links, Überschriften usw. hervorgehoben werden. Eine Zusammenfassung des Inhalts zu Beginn ist sinnvoll. Die Textgröße der Schrift sollte veränderbar sein.

2.3.5 Allgemeine Gestaltungsaspekte

Sprache

Wenn Benutzergruppen unterschiedlicher Herkunft unterstützt werden sollen, sollten lokale Versionen der Website bereitgestellt werden, die sich an Sprache und Kultur des entsprechenden Landes anpassen. Alle Sprachen, die unterstützt werden, sollten angezeigt werden. Die Sprache sollte global einstellbar sein, d.h. auf jeder Seite auf der man sich befindet. Formate, Maßeinheiten und Währungen sollten dabei der Lokalität des Nutzers entsprechen.

Hilfe anbieten

Die Benutzeroberfläche einer Website sollte so gestaltet sein, dass sie leicht zu verstehen und zu nutzen ist. Wenn Inhalte oder Funktionalitäten nicht für jeden verständlich sind, sollte dem Nutzer entsprechende Hilfe angeboten werden. Dies kann durch Hilfeseiten, FAQ-Seiten oder Dokumentation geschehen.

Fehlertolerante Gestaltung

Fehler, die bei der Nutzung auftreten, sollten minimiert werden. Bei auftretenden Fehlern sollten eindeutige Fehlermeldungen angezeigt werden, die auch die Ursache des Fehlers angeben und falls möglich die resultierende Maßnahme.

Sonstiges

Eine sinnvolle Benennung von URL-Adressen sollte beachtet werden.

Das Herunterladen von Dateien sollte sich in einer annehmbaren Zeitspanne bewegen.

Bei der Gestaltung der Oberfläche sind anerkannte Standards in der Webtechnologie und die Ausführung auf verschiedenen Eingabegeräten ebenfalls wichtige Faktoren, die zu berücksichtigen sind.

2.4 Gegenüberstellung der Definitionen

Die beiden Ansätze der Usability überschneiden sich stark, wobei der Ansatz von ISO spezifischer ist für Web-Shops. Einen Überblick über Gemeinsamkeiten und Unterschiede befindet sich in Tabelle 1.

Nielsen	Gemeinsamkeiten	ISO
Zusätzliche Aussagen über die Menge an Features	Minimalistisches Design bzw. Darstellung und Gestaltung des Inhalts: beide gehen auf Aspekte der Gestaltung ein wie: möglichst einfach und simpel, wenig Farben, Trennung von Inhalten mittels Linien, kurze Seitenlänge	ISO geht bei diesem Aspekt noch genauer auf die Wahl der Medienform ein und die individuelle Gestaltung, Nielsen hält sich hier allgemeiner
Übereinstimmung von System und Realwelt: Nielsen geht auf die Terminologie und dessen Bedeutung für besseres Verständnis und effizientere Nutzung ein.		
Erkennen vor Erinnern: nach dem Prinzip, der Mensch ist gut darin Dinge wiederzuerkennen, der PC ist gut sich an alle Details zu erinnern		
	Konsistenz und Standardisierung: beide beschreiben die Konsistenz: gleiche oder ähnliche Inhalte sollten sich an gleicher Stelle mit ähnlicher bzw gleicher Formatierung befinden	

Sichtbarkeit des Systemstatus: System soll den Nutzer informieren, was es macht (ihm passendes Feedback geben)		
Nielsen beschreibt diese Heuristik in Form von rückgängig machen von Aktionen	Benutzerkontrolle und Freiheit/ individuelle Anpassungen: Beide beschreiben die Notwendigkeit für individuelle Anpassungen.	ISO geht hier zusätzlich auf individuelle Anpassungen im Bereich von Navigation und Suche ein.
	Vermeiden von Fehlern-/ Fehlertolerante Gestaltung: Beide gehen auf das Prinzip ein, dass Fehler mit entsprechender Fehlermeldung abgefangen werden sollten oder durch Mechanismen ganz vermieden werden sollten	
	Hilfe und Dokumentation: Beide beschreiben die Notwendigkeit der Hilfe bzw. Dokumentation, um den Nutzer weiter zu unterstützen und zusätzliche Hilfe anzubieten, falls der Nutzer Probleme hat.	
		Navigation und Suche: Nielsen macht über diese beiden Punkte keine Aussage. ISO beschreibt genau wie eine sinnvolle Navigationsstruktur und eine hilfreiche Suche aufgebaut werden sollte. Dies ist sehr essenziell für eine Webseite.
		Sprache: ISO geht auf Internationalisierung ein.

Tab. 1: Eine Vergleich der in ISO und Nielsen dargestellten Heuristiken.

3 Sicherstellen von Web-Usability

Um sicherzustellen, dass die vorgestellten Prinzipien aus Kapitel 2 eingehalten werden, existieren verschiedene Ansätze. Die Usability Inspection, der Usability Engineering Lifecycle und die Menschzentrierte Gestaltung sollen als Auswahl dieser Vorgehensweisen näher betrachtet werden.

3.1 Usability Inspection Methods

Usability Inspection Methods ist ein Werk von Jacob Nielsen und Robert Mack, in dem diese fünf verschiedene Methoden von unterschiedlichen Autoren zur Sicherstellung bzw. zur Evaluation von Usability vorstellen. Im Folgenden soll auf die Methoden Heuristic Evaluation, Cognitive Walkthrough und Pluralistic Walkthrough eingegangen werden.

Heuristic Evaluation

Die heuristische Evaluation ist ein expertenbasiertes Verfahren. Dabei wird das Produkt auf die Erfüllung der in Kapitel 2.2 beschriebenen Heuristiken untersucht und bewertet. Diese Inspektion sollte von mehreren Usability-Experten (in der Regel fünf) unabhängig voneinander durchgeführt werden. Durch die unterschiedlichen Blickwinkel sollen möglichst viele Usability-Probleme identifiziert werden.

Nachdem die Evaluatoren mögliche Usability-Probleme ausgemacht haben stellen sie sich die Ergebnisse der Analyse gegenseitig vor und priorisieren diese auf einer Skala von null bis vier [Ni94b, S. 49]:

- 0 – kein Usability-Problem
- 1 – kosmetisches Problem
- 2 – kleines Usability-Problem
- 3 – großes Usability-Problem
- 4 – Usability-Katastrophe

Auf Basis dieser Priorisierung können die Probleme im Anschluss behoben werden, dabei werden Probleme mit einer höheren Priorisierung zuerst bearbeitet.

Die heuristische Evaluation ist allerdings nur eine Vorhersagemethode, d.h. die Experten bewerten anhand der Heuristiken, was in der Praxis ein Usability-Problem werden könnte und verfügen über wenig bis kein Wissen über den tatsächlichen Anwendungskontext. Bei einer Evaluation mit Benutzern können meistens qualitativ bessere Ergebnisse erzielt werden.

Allerdings ist diese Vorgehensweise verhältnismäßig kostengünstig und effizient, da man keine Benutzer einladen und befragen muss. Außerdem ist die Hürde eine heuristische Evaluation in der Praxis anzuwenden aufgrund der vergleichsweise geringen Komplexität relativ niedrig.

Cognitive Walkthrough

Der kognitive Durchgang ist ebenfalls ein expertenbasiertes Verfahren. Hier ist das Ziel der Experten aber sich in den Benutzer hineinversetzen und in diesem Kontext Aufgaben auszuführen.

Zu Beginn müssen Nutzer und die Aufgaben definiert werden. Anschließend werden Experten einberufen, die für jede Aufgabe gedanklich schrittweise durch die Anwendung gehen. Bei jedem dieser Schritte werden benutzerdefinierte Fragen aus Sicht des Benutzers beantwortet. Der Vorschlag der Autoren für diese Fragen ist in der folgenden Aufzählung zu sehen [Ni94b, S. 112].

- Wird der Benutzer versuchen, den richtigen Effekt zu erzielen?
- Wird der Benutzer erkennen, dass der korrekte Handlungsschritt verfügbar ist?
- Wird der Benutzer den korrekten Handlungsschritt mit dem zu erzielenden Effekt assoziieren?
- Wird der Benutzer nach dem korrekten Handlungsschritt erkennen, dass er seinem Ziel näher gekommen ist?

Sollte eine Frage nicht mit „ja“ beantwortet werden oder während des Durchgangs andere Probleme erkannt werden, werden diese mit Verbesserungsvorschlägen festgehalten und nach Abschluss des kognitiven Durchgangs angegangen [vgl. Ni94b, S. 106].

Problematisch ist hierbei allerdings, ähnlich wie bei anderen expertenbasierten Verfahren, dass möglicherweise einige Probleme nur durch Benutzer identifiziert werden können und somit beim kognitiven Durchgang nicht erkannt werden. Dennoch ist es ein kostengünstiges, effizientes Verfahren und hilft nicht nur zu erkennen, dass Usability-Probleme vorliegen, sondern auch von welcher Art diese Probleme sind.

Pluralistic Walkthrough

Der pluralistische Durchgang ähnelt dem kognitiven Durchgang, bringt aber fünf Schritte bzw. Charakteristika mit sich. Zum einen werden neben Usability-Experten auch die Benutzer und die Entwickler des Produkts miteinbezogen. Diese bilden das sogenannte „Walkthrough-Team“. Zum anderen wird, wie beim Cognitive Walkthrough, schrittweise durch das Produkt gegangen. Die dritte Eigenschaft ist, dass alle Mitglieder des Walkthrough-Teams einmal für sich die Rolle des Benutzers einnehmen. Dies soll dazu führen, dass sich auch Entwickler und Experten in die Benutzer hineinversetzen können.

Anschließend sollen die Mitglieder des Walkthrough-Teams im vierten Schritt ihre Beobachtungen so detailliert wie möglich notieren. Zum Abschluss treffen sich alle Mitglieder zu einer gemeinsamen Diskussion und diskutieren über jeden Schritt, der während des Durchgangs vollzogen wurde. Die Ergebnisse dieser Diskussion dienen anschließend der Verbesserung des Produkts.

3.2 Usability Engineering Lifecycle

Der Usability Engineering Lifecycle ist ein zyklisches Modell zur Gewährleistung von Usability bzw. Web-Usability während des Entwicklungsprozesses. Das Modell wurde von Deborah J. Mayhew ausgearbeitet [Ma10].

3.2.1 Requirements analysis

Im Rahmen der Anforderungsanalyse gilt es anfangs herauszufinden, welche Art von Nutzern das Produkt verwenden sollen. Dafür wird ein Nutzerprofil (**User Profile**) erstellt, das nutzerspezifische Charakteristika enthält. Dazu gehören Wissen und Erfahrung der Nutzer, aufgabenspezifische Merkmale (z.B. Häufigkeit der Verwendung), körperliche Eigenschaften (z.B. Farbenblindheit) und psychologische Merkmale wie die persönliche Motivation [Ma10, S. 36].

Im nächsten Schritt sollen die Aufgaben der Nutzer analysiert werden (**Contextual Task Analysis**). Dafür werden u.a. die aktuellen Tätigkeiten, typische Arbeitsabläufe und das Arbeitsumfeld betrachtet und ausgewertet. Dies soll vor allem durch Interviews der Nutzer im Arbeitsumfeld erfolgen [Ma10, S. 69]. Basierend auf den Anforderungen und des Nutzungskontexts werden anschließend Ziele definiert (**Usability Goal Setting**), die mithilfe des Produkts erreicht werden sollen. Dabei wird zwischen qualitativen (z.B. Design soll unerfahrene Nutzer bei komplexer Aufgabe unterstützen) und quantitativen Zielen (z.B. erfahrener Nutzer soll höchstens zwei Minuten für bestimmte Aufgabe brauchen) unterschieden [Ma10, S. 126 f.].

Als nächstes sollen die Eigenschaften und Beschränkungen der Plattform (d.h. Hard- und Software), auf der das Produkt betrieben werden soll, beschrieben werden (**Platform Capabilities and Constraints**). Dazu gehören zum Beispiel die Displaygröße, die Geschwindigkeit des Systems oder die möglichen Eingabegeräte [Ma10, S. 147]. Im letzten Schritt der Anforderungsanalyse gilt es allgemeine Design-Richtlinien (**General Design Principles**) zu identifizieren, die das Produkt betreffen.

Die Erkenntnisse der gesamten Anforderungsanalyse werden abschließend in einem Style Guide erfasst, der während der Umsetzung zur Überprüfung der Ergebnisse dient.

3.2.2 Design, testing and development

Die Phase der Entwicklung des Designs ist beim Usability Engineering Lifecycle in drei Ebenen aufgeteilt.

Level 1

Auf der ersten Ebene wird ein konzeptionelles Design erstellt, das die grundlegende Funktionalität, die Arbeitsabläufe und die definierten Regeln umsetzt. Davor muss allerdings die Beschreibung des Nutzungskontext und der Aufgaben überarbeitet werden (**Work Reengineering**). Anschließend werden vom Entwickler-Team Designregeln für die Präsentation der Informationen festgelegt (**Conceptual Model Design**). Auf Basis dieser Regeln wird im nächsten Schritt ein Mock-up entwickelt (**Conceptual Model Mock-ups**). Dieses Mockup wird anschließend in einem iterativen Prozess innerhalb des Entwickler-Teams evaluiert (**Iterative Conceptual Model Evaluation**): Solange das Mockup oder Teile davon die Entwickler nicht grundsätzlich zufrieden stellen und die grundlegenden Anforderungen erfüllt, wird weiter daran gearbeitet.

Level 2

Sobald dieses Mockup vorliegt, wird mit der zweiten Ebene fortgefahrene. Hier gilt es zunächst Design Standards (**Screen Design Standards**) festzuhalten. Diese Standards können schon allgemein vorhanden sein oder individuell für das Produkt definiert werden. Anschließend werden die Standards in Form von Prototypen getestet (**Screen Design Standards Prototyping**). Schlussendlich werden die Standards vom Entwickler-Team anhand der erstellten Prototypen evaluiert und bei Bedarf angepasst (**Iterative Screen Design Standards Evaluation**).

Level 3

Nachdem auf den beiden vorherigen Ebenen alle Vorkehrungen getroffen wurden, wird auf der dritten Ebene endgültig ein Design für das Produkt entworfen (**Detailed User Interface Design**). Dafür werden das auf Ebene 1 erstellte Mockup sowie der auf Ebene 2 erstellte Style-Guide verwendet. Das Design, das hierbei entsteht, ist eine erste Version des Produkts und dient als Basis für das Feedback der Nutzer. Davor werden an diesem Design allerdings Usability-Tests durchgeführt, die validieren sollen, ob das Design die in der Anforderungsanalyse definierten Ziele erfüllt (**Iterative Detailed User Interface Design Evaluation**). Sollte dies nicht der Fall sein werden entsprechende Änderungen am Design vorgenommen.

3.2.3 Installation

Nach der Umsetzung des Produkts folgt der Einsatz in der Praxis. Hierbei wird noch einmal Rücksprache mit den Benutzern gehalten (**User Feedback**). Dies kann auf verschiedenste

Art und Weise (z.B. E-Mail, Telefon, Umfragen, ...) passieren. Dabei ist aber wichtig, dass Probleme, die durch das Feedback aufgefallen sind, anschließend behoben werden.

3.3 Menschzentrierte Gestaltung

In der Norm *DIN EN ISO 9241-210: Menschzentrierte Gestaltung interaktiver Systeme* ist eine weitere Vorgehensweise zum Sicherstellen von Usability beschrieben, die grundsätzlich dem Usability Engineering Lifecycle ähnelt. Die menschzentrierte Gestaltung ist ein iterativer Prozess und hat zum Ziel, interaktive Systeme gebrauchstauglicher zu machen. Um dies zu realisieren wird der Fokus besonders auf den Austausch mit den Anwendern des Produkts gelegt. Dafür werden die Zwischenergebnisse frühzeitig mit diesen abgestimmt, damit auf etwaige Anforderungsänderungen reagiert werden kann.

Neben der Durchführung des Prozesses in Iterationen beinhaltet die Norm fünf weitere Grundprinzipien [DlC]. Zum einen soll die Anpassung der Gestaltung, wie bereits beschrieben, dauerhaft auf Basis der benutzerzentrierten Evaluierung erfolgen. Zum anderen soll das Gestaltungsteam aus Personen bestehen, die fachübergreifende Kompetenzen und Gesichtspunkte vereinen.

Die anderen drei Grundsätze beziehen sich auf den Anwender. Dazu gehört, dass die Gestaltung auf einem umfassenden Verständnis der Benutzer, deren Aufgaben und deren Arbeitsumgebung basiert. Außerdem müssen die Nutzer während der Gestaltung und Entwicklung zwingend miteinbezogen werden, da diese nach Abschluss des Projekts mit dem Produkt arbeiten. Zu guter Letzt ist es wichtig, dass bei der Gestaltung die User Experience (UX) berücksichtigt wird. UX beschreibt die Wahrnehmungen und Erfahrungen des Benutzers während der Interaktion mit der Anwendung. Dies beinhaltet die Zeit vor, während und nach der Benutzung.

Außer den zentralen Grundsätzen der menschenzentrierten Gestaltung sind die in der Norm definierten Projektphasen (siehe Abbildung 1) essenziell. Nach der anfänglichen Planung des menschzentrierten Gestaltungsprozesses folgt eine Analyse. Deren Ziel besteht darin, den Nutzungskontext („Kombination von Benutzern, Zielen, Aufgaben, Ressourcen und Umgebung“) zu verstehen und in einer Nutzungskontextbeschreibung zu sichern.

Die Anforderungen der Nutzenden werden in der nächsten Phase genau spezifiziert und festgehalten. Im Anschluss daran werden daraus Gestaltungslösungen, bspw. in Form eines sogenannten Mockups, entwickelt. Diese werden in der letzten Phase gegen die Nutzungsanforderungen evaluiert, d.h. der Nutzende wird aufgefordert, mit diesem Mockup zu arbeiten. Falls nach der Evaluation Änderungsbedarf besteht, wird mit einer neuen Iteration begonnen.

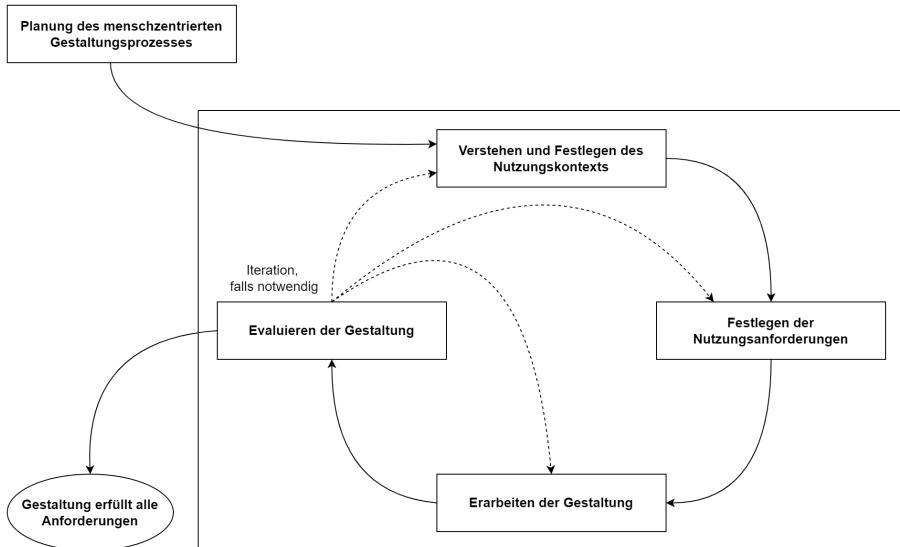


Abb. 1: Übersicht menschzentrierter Gestaltungsprozess (eigene Darstellung, [vgl. Je20])

3.3.1 Planen der menschenzentrierten Gestaltung

Vor Umsetzung der menschenzentrierten Gestaltung gilt es, den Prozess zu planen. Die Planung umfasst dabei die Identifikation und Definition von geeigneten Methoden und Ressourcen für die Umsetzung der menschenzentrierten Gestaltungsaktivitäten, die im nächsten Kapitel beschrieben werden. Essenziell sind hierbei die Verfahren zum Einholen von Rückmeldungen vom Benutzer, d.h. der Evaluation der Gestaltungslösung.

Außerdem müssen die einzelnen Gestaltungsaktivitäten im Blick auf zeitlichen und personellen Aufwand abgeschätzt werden. Dafür wird im Zuge der Planung des menschenzentrierten Gestaltungsprozesses ein Zeitplan erstellt.

3.3.2 Menschenzentrierte Gestaltungsaktivitäten

Verstehen und Festlegen des Nutzungskontexts

Zu Beginn der Anforderungsanalyse steht die Erfassung und Beschreibung des Nutzungskontexts an. Der Nutzungskontext ist die Gesamtheit von Benutzern, deren Aufgaben und Zielen, die Umgebung, in der das Produkt eingesetzt wird und die zur Verfügung stehenden Ressourcen [vgl. DIc, S. 8]. Festgehalten wird der Kontext in einer sogenannten Nutzungskontextbeschreibung, die während des menschenzentrierten Gestaltungsprozesses ständig angepasst werden kann.

Festlegen der Nutzungsanforderungen

In der nächsten Phase werden die Anforderungen der Benutzer und die Erfordernisse weiterer Stakeholder an das Produkt bestimmt. Dies geschieht unter Berücksichtigung des vorher bestimmten Nutzungskontexts.

Die Nutzungsanforderungen werden in einer Spezifikation festgehalten, die beinhaltet, was die Benutzer erreichen wollen. Dabei wird nicht darauf eingegangen, wie sie es erreichen. Außerdem sollen alle durch den Nutzungskontext gegebenen Einschränkungen festgehalten werden. Diese Spezifikation soll anschließend durch die relevanten Stakeholder verifiziert werden und bei Bedarf während des Projekts angepasst werden [vgl. DIA, S. 22 f.].

Erarbeiten von Gestaltungslösungen

Nach der Ausarbeitung der Nutzungsanforderungen werden unter Berücksichtigung dieser möglichen Gestaltungslösungen erstellt. Dabei ist es außerdem wichtig, die Interaktionsprinzipien zu beachten, die in der Norm „Ergonomie der Mensch-System-Interaktion - Teil 110“ (DIN EN ISO 9241-110:2020) beschrieben sind [vgl. DIA, S. 11]. Zu diesen Prinzipien gehört die Aufgabenangemessenheit. Diese ist erfüllt, wenn das System die Benutzenden bei der Erfüllung der Aufgaben unterstützt. Dafür müssen die Funktionen der Anwendung auf den tatsächlichen Aufgaben der Benutzenden basieren und nicht auf der verwendeten Technologie.

Ein weiteres Prinzip ist die Selbstbeschreibungsfähigkeit, d.h. dem Benutzenden muss ohne zusätzliche Interaktion mit der Anwendung ersichtlich sein, welche Funktionen die Anwendung hat und wie diese auszuführen sind. Dazu gehört auch die dauerhafte und eindeutige Anzeige des Systemzustands [vgl. DIA, S. 12].

Erwartungskonformität besagt, dass das Verhalten der Anwendung für den Benutzenden vorhersehbar sein muss. Außerdem sollen die Webanwendung Rückmeldung geben, nachdem der Benutzende eine Aktion ausgeführt hat [vgl. DIA, S. 21].

Das Prinzip Erlernbarkeit fordert, dass der Benutzende die Funktionen der Anwendung und deren Verwendung entdecken kann. Steuerbarkeit ist ein weiteres Prinzip, das aussagt, dass der Benutzende zu jedem Zeitpunkt die Kontrolle über die Anwendung und die Interaktionen mit dieser behalten muss. Dazu gehört zum Beispiel, dass der Benutzende eine Aufgabe zu jedem Zeitpunkt unterbrechen kann. Das Prinzip Robustheit gegen Benutzungsfehler beschreibt, dass die Anwendung den Benutzenden beim Vermeiden von Fehlern unterstützt. Des Weiteren sollen Benutzungsfehler toleriert werden [vgl. DIA, S. 30].

Das letzte Prinzip ist die Benutzerbindung. Das Ziel dieses Prinzips ist es den Benutzenden durch eine einladende und motivierende Darstellung der Funktionen an die Anwendung zu binden [vgl. DIA, S. 11].

Evaluieren der Gestaltung

Schlussendlich soll die Gestaltungslösung benutzerzentriert evaluiert werden. Für diese Evaluierung bietet die menschzentrierte Gestaltung mehrere Methoden.

Die Methode, die hauptsächlich angewandt werden soll, ist die **Prüfung mit Benutzern**. Diese kann auch in einem sehr frühen Stadium (z.B. Evaluierung von Skizzen oder Mockups) eingesetzt werden. Die Benutzer sollen hierbei die Gestaltungskonzepte bewerten, die ihnen vorgelegt werden. Später können den Benutzern Prototypen vorgelegt werden, mit welchen diese unter Beobachtung realistische Aufgaben mit dem Produkt durchführen. Sobald das Produkt umgesetzt ist, kann mit der Prüfung mit Benutzern evaluiert werden, ob die Ziele im Nutzungskontext erreicht wurden [DIA, S. 29]. Mit den gewonnenen Informationen wird anschließend die Gestaltung verbessert.

Eine weitere Methode ist die **inspektionsbasierte Evaluierung**. Diese wird von Fachleuten auf dem Gebiet der Gebrauchstauglichkeit durchgeführt und kann ergänzend vor der Prüfung mit Benutzern durchgeführt werden. Die inspektionsbasierte Evaluierung ist vor allem sinnvoll, um größere Probleme vorab zu beseitigen und die Evaluierung mit den Benutzern dadurch wirtschaftlicher zu machen [DIA, S. 29].

Die dritte Methode, die immer Teil der Evaluierung sein sollte, ist die **Langzeitbeobachtung**. Diese Beobachtung beinhaltet das Erfassen von Benutzerrückmeldungen auf unterschiedliche Weise über eine gewisse Zeit [DIA, S. 30]. Dies ist wichtig, da einige Probleme erst erkennbar werden, wenn das Produkt längere Zeit genutzt wird.

3.4 Vergleich der Methoden

Die Methoden der Usability Inspection aus Kapitel 3.1 dienen ausschließlich zur Evaluation. Diese können aber sowohl zur Evaluation von einfachen Mockups als auch zur Evaluation von Oberflächen, die kurz vor der Fertigstellung sind, eingesetzt werden. Der Usability Engineering Lifecycle und die menschenzentrierte Gestaltung sind hingegen Prozesse zur Erstellung von Oberflächen, die diese Evaluation integrieren, um Usability sicherzustellen.

Bei der Evaluation setzen die heuristische Evaluation und der kognitive Durchgang ausschließlich auf Usability-Experten. Dies senkt die Kosten und steigert die Effizienz, kann aber dazu führen, dass bestimmte Usability-Probleme nicht erkannt werden. Außerdem besteht die Gefahr zur Identifikation von sogenannten „Scheinproblemen“, die für die Benutzer nicht von Bedeutung sind. Der pluralistische Durchgang setzt für die Evaluation sowohl Experten als auch Entwickler und Benutzer ein. Damit können mehr Probleme identifiziert werden, gleichzeitig steigt aber auch der Aufwand.

Zusammenfassend lässt sich sagen, dass Prozesse wie die menschenzentrierte Gestaltung aufgrund der dauerhaften Beachtung der Benutzer gut geeignet ist, um bei der Entwicklung von neuen Produkten Usability sicherzustellen. Einfache Methoden wie die heuristische Evaluation oder der kognitive Durchgang eignen sich hingegen, um bei Produkten in der Entwicklung oder fertiggestellten Produkten mit geringem Aufwand die grundlegende Usability zu überprüfen.

4 Untersuchen von Web-Usability anhand eines Webshops

In diesem Kapitel wird das Vorgehen der Usability Inspection anhand der Webshops der Böttcher AG und von Nike Inc. untersucht. Dabei wird zuerst darauf eingegangen warum Web-Usability wichtig für Webshops ist. Daraufhin wird durch eine Heuristische Evaluation mithilfe den Heuristiken aus Kapitel 1 die beiden Webshops evaluiert. Schlussendlich wird die Anwendbarkeit der Usability Inspection mit den Heuristiken für die zwei verschiedenen Arten von Webshops verglichen. Dabei wird zuerst überprüft ob die Methode der Heuristischen Evaluation generell sinnvoll zur Verbesserung der Usability von Webshops. Auch werden die Heuristiken aus Kapitel auf ihre Anwendbarkeit bezüglich Webshops generell überprüft. Schlussendlich wird auf Einzelheiten der Methode bezüglich der Besonderheiten beider Webshops eingegangen.

4.1 Bedeutung von Web-Usability für Webshops

Immer mehr Menschen benutzen das Internet für den elektronischen Handel [CC02]. Alleine von 2020 bis 2021 stieg der Umsatz im Bereich E-Commerce in Deutschland um 20% [Ru21]. Doch für diesen Trend sorgt nicht nur die Covid-19 Pandemie. Auch in den Jahren von 2015 bis 2020 verdoppelt sich der Umsatz des E-Commerce in Deutschland von ca. 50 Milliarden auf knappe 100 Milliarden Euro [Ru21].

Dabei gibt es große Unterschiede zwischen verschiedenen Arten von Webshops, die sich wiederum auch auf die Usability auswirken. So gibt es Unterschiede ob der Webshop einen Produkt oder eine Dienstleistung vertreibt [CC02]. Ein Beispiel für einen Webshop der eine Dienstleistung vertreibt ist z.B. ein Buchungsportal eines Flugunternehmens. Ein weitere großer Unterschied ist die Distributionspolitik der E-Commerce Unternehmen. So unterscheiden sich Webshops hinsichtlich ihrer Zielgruppe und dadurch auch durch ihre Nutzer. Unternehmen die Produkte an andere Unternehmen vertreiben (B2B) haben dabei andere Usability-Anforderungen als Unternehmen die ihre Produkte an Konsumenten vertreiben (B2C) [TK21].

Webshops sorgen für einige weitere Herausforderungen im Bezug auf Usability. Während man in einem physischen Geschäft z.B. Mitarbeiter nach Hilfe fragen kann, muss ein Kunde in einem Webshop selbst die gewollten Produkte finden. Dies ist eine große Herausforderung den aufgrund der großen Konkurrenz im E-Commerce wechseln Kunden häufig den Anbieter nur weil sie ihr gewolltes Produkt nicht finden [Sü16]. Durch eine Studie nach Jakob Nielsen im Jahr 1998 wurde herausgefunden das 60% der Online-Kunden ihren gesuchten Artikel nicht finden können und deswegen den Webshop frühzeitig verlassen [Sü16].

Und das sind nicht die einzigen Herausforderungen von Webshops. Nicht umsonst schafft man laut Süß einen Umsatzzuwachs von 200% nur durch eine Optimierung des Webshops bezüglich der Usability für die Kund*innen [Sü16]. Auch Kaur ist der Meinung dass "der wichtigste Aspekt der den Wert eines Webseite bestimmt ist die Usability" [KKK18].

Die fünf Bereiche in denen Usability vor allem einen Vorteil schaffen kann sind dabei laut Süss [Sü16]:

- Umsatzsteigerung
- besserer Zugang zu Produkten
- Loyalität und Vertrauen
- Kosteneinsparung für Kundendienst
- Vorteile im Wettbewerb

Kennzahlen

Kennzahlen sind ein wichtiges Messinstrument zur Bestimmung der Usability von Webshops. Dabei gibt es eine große Anzahl verschiedener Kennzahlen. Heimann und Mahrdt teilen diese jedoch in sechs Bereiche ein. Abbildung 2 zeigt eine Auswahl an Kennzahlen die in die jeweilige Bereiche sortiert wurden.

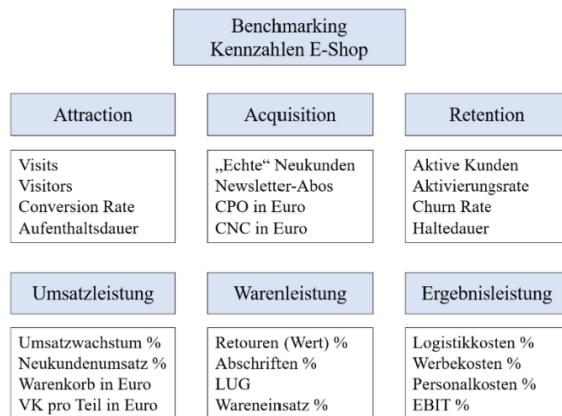


Abb. 2: Kennzahlen für Webshops in sechs Bereiche [Ga18]

Die wichtigste Kennzahl für die Usability eines Webshops ist aber die Conversion Rate [Ga18]. Die Conversion Rate gibt an wie viele Besuche eines Webshops zu einem Erfolg führen (Erfolg: meistens Kauf eines Artikels). Berechnet wird die Conversion Rate durch die Anzahl der Erfolge * 1000 / Anzahl der Besuche. Das Ergebnis dieser Formel wird in Prozent angegeben. Dicht gepaart mit der Conversion Rate ist die Abbruchrate. Die Abbruchrate gibt an wie viel Prozent der Besucher keinen Kauf erfolgreich abschließen. Die Abbruchrate wird daher durch 1 - Conversion Rate errechnet [Ga18].

Um die Conversion Rate zu verbessern müssen die Abbruchursachen identifiziert werden und den Benutzer einen simplen und verständlichen Weg bereitstellen einen Kauf abzuschließen.

4.2 Vorgehen

Für die Evaluation des Webshops wird eine Usability Inspection durchgeführt. Als konkrete Methode wird dabei die Heuristische Evaluation verwendet.

Der Grund für diese Entscheidung ist, dass die Usability Inspection bzw. Heuristische Evaluation eine sehr kostengünstige und effiziente Methode ist. Damit ist sie besonders attraktiv für E-Commerce Unternehmen mit begrenzten Ressourcen bzw. Fachwissen über Usability. Die Methode ist für fast jeden Webshop umsetzbar, weswegen die Relevanz einer praktischen Evaluation dieser Methode gegeben ist. Die Heuristische Methode wird wie folgt durchgeführt:

1. Zu Beginn der Evaluation wird aus den beschriebenen Heuristiken in Kapitel 1 ein Fragebogen (s. Anhang) erstellt.
2. Anhand dieses Fragebogens werden von einem Experten Usability-Probleme identifiziert.
3. Im Kapitel 4.3 werden dann die gefundenen Probleme beschrieben.
4. Daraufhin wird jedes Problem bewertet. Für diese Bewertung wird die Scala aus dem Kapitel 3.1 verwendet. Dabei werden Usability-Probleme von einer Scala von null bis vier nach ihrer Schwere eingeordnet (0 = kein Problem, 4 = Usability-Katastrophe).
5. Schlussendlich wird für beide Webshops das Gesamtergebniss dargestellt. Dabei wird die Anzahl der Probleme pro Heuristik in einem Balkendiagramm dargestellt. Zusätzlich wird die durchschnittliche Schwere der Probleme pro Heuristik berechnet und dargestellt. Dadurch bekommt man einen guten Überblick wie gut die beiden Webshops bei der Heuristischen Evaluation abgeschnitten haben und welche Probleme besonders schwerwiegend sind.

4.3 Evaluation der Webshops

In diesem Kapitel werden zwei Webshops mithilfe der Methode Usability Inspection evaluiert. Dabei wird das Vorgehen wie im vorherigen Kapitel beschrieben verwendet. Während der Evaluation wird vor allem auf die Usability-Probleme der Webshops eingegangen. Jedoch werden vereinzelt besonders positive Aspekte herausgehoben.

Für die Evaluation der zwei Webshops wurden die Webshops der Böttcher AG und Nike Inc. ausgewählt.

4.3.1 Böttcher AG

Die Böttcher AG ist ein deutscher Online-Händler für Büroartikel, die ihre Ware hauptsächlich B2B vertreibt. Ihr Produktsortiment besteht aus knapp 200.000 Artikel und sie haben eine Kundenstamm von 7 Millionen Menschen [Bö22]. Dabei fokussiert sich der Online-Händler auf Verbrauchsgegenstände im Büroalltag. Jedoch ist das Produktpotfolio mittlerweile auf z.B. Arbeitskleidung und Werkzeuge erweitert. Ihrem Webshop erreicht man unter der URL <https://www.bueromarkt-ag.de/>.

Ästhetisches und minimalistisches Design

Der Webshop der Böttcher AG ist sehr überladen. Böttcher versucht auf der Startseite so viele Artikel wie möglich dem/der Nutzer*in anzubieten. Die eigentliche Intention einer Startseite ist jedoch dem/der Nutzer*in zu erlauben sich im Webshop zu navigieren. Besonders im B2B-Bereich haben die Nutzer*innen eine klare Vorstellung welche Produkte bestellt werden müssen. Es ist also fraglich wie sinnvoll das Werben von Produkten auf der Startseite ist. Das Problem ist, dass die Werbung viel Platz auf der Startseite verbraucht, die für wichtigere Funktionen genutzt werden kann. Dazu kommt das die verschiedenen Werbebereiche redundant sind. Es gibt zum Beispiel einen Bereich „Top-Empfehlungen für Sie“ und einen Bereich „Ausgewählte Artikel für Sie“. Diese Bereiche werden als unterschiedliche Funktionen verkauft, bedeuten aber dasselbe. (Schwere: 4)

Im Navigationsmenü werden ebenfalls verschiedene Werbe-/Angebotsbereiche verlinkt. Die eigentliche Navigation über die verschieden Produktkategorien befindet sich an letzter Stelle des Navigationsmenüs. Die mit wichtigste Funktion der Startseite nimmt also eine sehr untergeordnete Rolle ein. Besser wäre es die Angebotsbereiche zu entfernen und das gesamte Navigationsmenü für die Navigation über die Kategorien zu nutzen. Durch den gewonnenen Platz könnten die Kategorien noch feiner und untergliedert dargestellt werden. (Schwere: 2)

Auf der Startseite werden ebenfalls Kerndaten und Rezensionen des Webshops angezeigt. Auch diese Informationen sind für die Nutzer*innen an erster Stelle nicht relevant. Kunden im B2B-Bereich sind vorrangig Bestandskunden. Informationen über den Webshop sollten also auf einer getrennten Seite zu finden sein. Wenn Neukunden sich für diese Informationen interessieren, sollte die Startseite gut sichtbar auf diese Seite verweisen. (Schwere: 1)

Wählt man eine Kategorie aus werden erst weitere Empfehlungen und Filtermöglichkeiten angezeigt. Erst dann findet man eine Liste mit den Produkten der ausgewählten Kategorie. Bei kleineren Bildschirmen kann es sogar sein das die Produkte auf den ersten Blick gar nicht sichtbar sind. Man findet die Artikel nur durch das Scrollen nach unten. Die Filtermöglichkeiten sind außerdem nicht einklappbar. So kann es vorkommen das der

Filterbereich die eigentlichen Artikel verdeckt, ohne dass der/die Nutzer*in diese Funktion nutzen möchte. (Schwere: 4)

Übereinstimmung von System und realer Welt

Wählt man eine Kategorie aus kann man nur noch über eine „A-Z Leiste“ weiter navigieren. Wählt man einen Buchstaben auf dieser Leiste aus bekommt man alle Unterkategorien angezeigt, die mit diesem Buchstaben beginnen. Das Problem ist das Böttcher Kategorien anders benennen kann wie die Nutzer*innen erwarten. Sucht man z.B. unter dem Buchstaben K nach der Kategorie „Kugelschreiber“ findet man nichts. (Schwere: 3)

Die Sortierung der Unterkategorie nach dem Alphabet ist außerdem aus einem weiteren Grund unnatürlich. In einem physischen Geschäft werden die Produkte nicht nach dem Alphabet sortiert. In einem Supermarkt werden die Waren in Bereiche unterteilt. Innerhalb dieser Bereiche werden die Waren weiter unterteilt. Sucht man z.B. in einem Supermarkt nach einer Flasche Weißwein sucht man zuerst in der Getränkeabteilung. In dieser Abteilung sucht man den Bereich mit alkoholischen Getränken. Dort sucht man dann nach Weinen. Die natürliche Vorgehensweise ist das gesuchte Produkt in immer kleiner werden Kategorien zu suchen. (Schwere: 3)

Eine weitere Möglichkeit wie man zu dieser Leiste gelangt ist durch den Link „Artikel A-Z“ auf der Startseite. Wenn man weiß, wie die Navigationsfunktion hinter diesem Link funktioniert scheint diese Benennung sinnvoll. Weiß man aber nichts von dieser Funktion ist die Benennung verwirrend. Was sich hinter diesen Link nämlich befindet, ist eine Liste aller Artikel. Eine passendere Benennung des Links wäre also eher „alle Artikel“. (Schwere: 2)

Positiv ist das sich der Webshop in bestimmten Bereichen an bestimmte Normen hält. So ist es z.B. üblich, dass der Warenkorb in der rechten oberen Ecke durch ein Einkaufswagen-Icon erreicht werden kann. (Schwere: 0)

Erkennen vor Erinnern

Anzumerken ist das die Ergebnisse dieser Heuristik für die Böttcher AG weniger relevant sind. Die Kunden im B2B-Bereich sind überwiegend Bestandskunden, die regelmäßig Produkte bestellen. Benutzt man den Shop regelmäßig erlernt man seine Funktionen. Diese Funktionen müssen also nicht jedes Mal neuerkannt werden.

Viele Funktionen des Webshops sind versteckt oder gehen unter. Der Link „Artikel A-Z“ ist z.B. sehr unauffällig über der Suchleisten platziert. Diese Funktion würde man eher im Navigationsmenü erwarten. Dort wäre die Funktion ihrer Bedeutung gerecht auffällig genug. (Schwere: 3)

Eine besonders gute Funktion ist die Einkaufsliste. In dieser Liste kann man Artikel über eine Sitzung hinweg speichern. Wenn regelmäßig Artikel nachbestellt werden kann man diese in der Einkaufsliste vermerken. Dadurch muss man sich nicht erinnern welches Produkt man das letzte Mal bestellt hat, sondern kann es aus der Einkaufsliste direkt in den Warenkorb verschieben. (Schwere: 0)

Eine weitere gute Funktion ist das Artikel über die Bestellnummer in den Warenkorb hinzugefügt werden können. Möchte man wieder bestimmte Artikel nachbestellen und hat diese nicht in der Einkaufsliste gespeichert, muss man nur die Bestellnummer der letzten Rechnung heraussuchen. (Schwere: 0)

Konsistenz und Standardisierung

In dieser Heuristik schneidet der Webshop der Böttcher AG sehr schlecht ab. Oft passiert es dass ähnliche Aktionen zu unterschiedlichen Ergebnissen führen. Wählt man z.B. eine Kategorie aus, kann es sein das Empfehlungen oder Filtermöglichkeiten angezeigt werden. Wählt man eine andere Kategorie kann es sein das beides oder weder noch angezeigt wird. Diese Inkonsistenz sorgt für Verwirrung bei den Nutzer*innen, da sie das Gefühl bekommen in einem ganz anderen Bereich zu sein. (Schwere: 3)

Sichtbarkeit des Systemstatus

In dieser Heuristik schneidet der Webshop positiv ab. Wenn ein/e Nutzer*in ein Produkt in den Warenkorb legt, wird durch eine grüne Erfolgsmeldung signalisiert das die Aktion erfolgreich war. Ebenfalls sehr positiv für die Sichtbarkeit des Systemstatus ist, dass der Gesamtpreis des Warenkorbs dem Nutzer immer angezeigt wird. Dadurch verliert der/die Nutzer*in nicht den Überblick wie viel er/sie gerade ausgibt. (Schwere: 0)

Dennoch gibt es kleinere Probleme. Ein Problem findet statt, wenn man im Warenkorb die Menge eines Produkts ändert. Das System setzt voraus, dass der/die Nutzer*in den Warenkorb manuelle aktualisieren muss, damit die Änderung durchgeführt wird. Der/Die Nutzer*in muss dabei auf einen unauffälligen Button drücken. Dieses manuelle Aktualisieren kann von dem/der Nutzer*in übersehen werden, was dann zu einer falschen Bestellung führen könnte. Der Zustand des Warenkorbs sollte sich demnach automatisch aktualisieren, wenn die Menge geändert wird. (Schwere: 2)

Benutzerkontrolle und Freiheit

Der/Die Nutzer*in kann sich frei in dem Webshop bewegen. Teilweise müssen dabei die Funktionalitäten des Browser zur Unterstützung eingesetzt werden. Zum Beispiel gibt

es kaum Möglichkeiten zurückzugehen. Diese Aktion passiert häufig und kann durch zusätzliche Buttons verbessert werden. Die Freiheiten der Nutzer*innen im Warenkorb sind standardmäßig. Man kann Artikel hinzufügen, entfernen und die Menge verändern. (Schwere: 1)

Eine wichtige Funktion die die Freiheit der Nutzer*innen erhöht sind Breadcrumbs. Diese Funktion stellt im Webshop der Böttcher AG die einzige Möglichkeit da, beliebige Stufen zurückzugehen. Gerade bei der unübersichtlichen Navigation im Webshop sind Breadcrumbs besonders wichtig. Leider ist diese Funktion ebenfalls sehr unauffällig gestaltet. Die Funktion geht trotz ihrer hohen Wichtigkeit auf der Webseite unter. (Schwere: 3)

Gegen das vertikale Scrollen wurde eine „Gehe nach oben“-Funktion eingeführt. Sobald man nach unten scrollt, erscheint ein Button, mit dem man wieder an den Anfang der Seite gelangt. Jedoch ist auch diese Funktion sehr versteckt. Der Text des Buttons hat eine hellgraue Farbe was auf dem weißen Hintergrund schwer erkennbar ist. (Schwere: 1)

Vermeiden von Fehlern

Während der Evaluation konnte keine Mängel bezüglicher dieser Heuristik gefunden werden. (Schwere: 0)

Hilfe und Dokumentation

Zur Hilfe wird den Nutzer*innen ein FAQ angeboten. Im Vergleich zu dem Webshop ist dieses sehr übersichtlich und minimal. Das FAQ beantwortet die wesentlichen Fragen. Es kann über den Footer der Webseite erreicht werden. Das ist eine übliche Positionierung, weswegen es keine Bedenken geben muss, dass man es nicht findet. (Schwere: 0)

Navigation und Suche

Die Navigation ist recht kompliziert. Das kann daran liegen das Böttcher eine große Anzahl an verschiedene Produkte vertreibt. Kategorien sind meistens nicht logisch gruppiert, sondern stehen einzeln. Das liegt daran das die Kategorie nach dem Alphabet sortiert sind. Deswegen kann es sein das ähnliche Kategorie sehr weit auseinander stehen. (Schwere: 3)

Die Navigationstruktur ist sehr breit angelegt. Jedoch muss man durch die große Anzahl an Artikel trotzdem sehr tief in den Navigationsbaum einschreiten. (Schwere: 1)

Durch die unauffällig Breadcrumbs-Funktion ist den Nutzer*innen nicht klar wo sie sich in der Navigationstruktur befinden. (Schwere: 2)

Positiv anzumerken ist die globale Suchfunktion. Auch wenn man diese nicht weiter einschränken kann, findet man durch die Suche häufiger die richtigen Artikel als mit der Navigation. (Schwere: 0)

Sprache

Böttcher vertreibt seine Ware nur deutschlandweit, weswegen die Sprache weniger bedeutsam ist. (Schwere: 0)

4.3.2 Nike Inc.

Nike Inc. ist ein amerikanischer Sportartikelhersteller. Nike vertreibt seine Artikel hauptsächlich über einen eigenen Webshop. Im Vergleich zu Böttcher handelt es sich bei Nike um einen Lifestyle-Shop. Dementsprechend vertreiben sie ihre Artikel B2C. Lifestyle-Shops fokussieren sich auf Artikel, die im Zusammenhang mit einem aufstrebenden Lebensstil und Popkultur stehen. Nike erwirtschaftete durch ihren Webshop im Jahr 2021 einen Umsatz von 10 Milliarden US-Dollar [Pe22]. Den Webshop der Nike Inc. erreicht man unter der URL: <https://www.nike.com/de/>

Ästhetisches und minimalistisches Design

Der Webshop von Nike ist sehr minimalistisch gehalten. Nike hat in diesem Aspekt auch einen gewissen Vorteil gegenüber Böttcher. Nike's Auswahl an verschiedenen Produkten ist wesentlich geringer als das Sortiment von Böttcher. Das vereinfacht z.B. die Navigation, da weniger Kategorien angeboten werden.

Insbesondere die Farbpalette ist sehr minimalistisch. Der Webshop ist grundsätzlich schwarz-weiß. Vereinzelt werden verschiedene Grautöne eingesetzt. Die einzigen farbigen Elemente im Webshop sind die Produkte selbst bzw. Werbungen für diese Produkte. Die Produktbilder sind im Vergleich zu Böttcher hochwertig produziert. Nike lenkt damit den Fokus der Nutzer*innen auf ihre Produkte. Der Webshop um die Produkte herum ist schlicht und minimalistisch gehalten. Die Funktionen sind auf die wichtigsten reduziert. Die Aufgabe des Webshops ist daher rein funktional. Durch das minimalistische Design wirkt der Webshop sehr übersichtlich und strukturiert.

Auch werden nur die wichtigsten Informationen angezeigt. Alle Informationen, die auf den ersten Blick nicht relevant sind, werden zu Beginn zugeklappt. Erst wenn ein Nutzer sich wirklich für eine Information interessiert, kann er diesen Bereich aufklappen. Zum Beispiel sind die Produktdetails oder Informationen zum Versand erst eingeklappt. (Schwere: 0)

Übereinstimmung von System und realer Welt

Für diese Heuristik wurden im Webshop von Nike keine Mängel festgestellt. Funktionen und Bereiche werden konsistent gut benannt. Auch werden die Funktionen mit passenden Icons untermauert. (Schwere: 0)

Erkennen vor Erinnern

Der Webshop von Nike verfügt über eine Funktion zum Speichern von Produkten über eine Sitzung hinweg. Diese Funktion heißt Favoriten. Gleich wie bei Böttcher sorgt diese Funktion, dass sich Kunden an Produkte nicht erinnern müssen. Durch diese Funktionen können Sie ihre Lieblingsprodukte auf einen Blick sehen. Die Funktion wird zusätzlich durch ein Herz-Icon verdeutlicht. Jedoch können Produkte nur über die Produktübersicht zu den Favoriten hinzugefügt werden. Über die Produktliste direkt ist es nicht möglich. Möchte ein Nutzer also die Produktliste überfliegen um Produkte, die im Gefallen in die Favoriten zu speichern muss dieser immer erst in die Produktübersicht. Nach dieser Aktion muss er/sie dann wieder zurück. Durch die Möglichkeit direkt von der Produktliste Produkte in den Favoriten zu speichern könnte der/die Nutzer*in diese Aktion effizienter ausführen. (Schwere: 0)

Eine Funktion, die etwas unnatürlich ist, ist das Ein- und Ausklappen der Filterfunktion. Die Filterfunktion befindet sich auf der linken Seite des Webshops. Das Icon, um die Filterfunktion ein- und auszuklappen, befindet sich jedoch auf der rechten Seite. Diese Platzierung ist sehr unnatürlich für die Nutzer*innen. Besser wäre, dass das Filter-Icon ebenfalls auf der linken Seite platziert wäre, damit klarer wird dass das Icon mit der Filterfunktion korreliert. (Schwere: 1)

Konsistenz und Standardisierung

Der Webshop hat eine hervorragende Konsistenz. Unabhängig von der Kategorie der Produkte sehen Produktliste und Produktübersicht identisch aus. Dadurch sind die Nutzer weniger verwirrt und müssen sich nicht verschiedene Vorgehensweisen aneignen. (Schwere: 0)

Sichtbarkeit des Systemstatus

Befindet man sich in der Produktliste werden, wenn man weiter nach unten scrollt weitere Produkte angezeigt. Der Nike Shop verzichtet also auf eine Pagination-Funktion. Das Laden neuer Produkte ist eine asynchrone Aktion, die von der Netzwerkverbindung zum dahinter liegenden Anwendungsserver abhängt. Aus diesem Grund kann es sein, dass der Nutzer auf

der Clientseite schon das Ende der Seite erreicht, der Webshop aber noch nicht alle Produkte geladen hat. Dieser Systemstatus wird den Nutzer*innen passend mit einem Ladebalken signalisiert. (Schwere: 0)

Parallel zum Webshop von Böttcher wird den Nutzer*innen bestätigt, wenn sie erfolgreich ein Produkt in den Warenkorb oder zu den Favoriten hinzugefügt haben. Bei Nike wird ein kleines Menü angezeigt, welches das Produkt anzeigt und Links zu den Favoriten bzw. zum Warenkorb bereitstellt. Dieses Menü wird weiter hervorgehoben, indem der Webshop im Hintergrund dunkler wird. Möchte man normal fortfahren muss man das Menü manuell schließen. Diese Aktion kann sehr repetitiv werden wenn man oft Produkte zu dem Warenkorb oder zu den Favoriten hinzufügt. Auf der anderen Seite geht Nike damit sicher, dass der/die Nutzer*in erkannt hat das die Aktion erfolgreich war. (Schwere: 0)

Negativ bei Nike ist, dass der Webshop nicht den aktuellen Gesamtpreis des Warenkorbs immer anzeigt. Um die Gesamtsumme zu sehen, muss man den Warenkorb öffnen. (Schwere: 1)

Benutzerkontrolle und Freiheit

Parallel zum Webshop von Böttcher gib es wenige Zurück-Buttons. Auch hier muss man für diese Funktion die Browserfunktionalitäten zur Unterstützung benutzen. Die einzige Alternative ist ebenfalls die Breadcrumb-Funktion. Diese Funktion ist aber ähnlich zu Böttcher unauffällig platziert. (Schwere: 3)

Der Webshop von Nike verzichtet auf eine Pagination-Funktion. Anstelle davon werden neue Produkte geladen, sobald man weiter nach unten scrollt. Aus Nutzersicht fühlt diese Funktion sich natürlicher an, da man einfach die Scroll-Aktion weiterführen kann. Jedoch birgt diese Funktion auch einige Gefahren.

1. Wenn ein/e Nutzer*in weiß das ein Produkt weiter hinten in der Liste auftaucht muss er/sie ganz nach unten scrollen und kann nicht Seiten überspringen.
2. Möchte man zum Footer gelangen muss man ebenfalls bis zum Ende scrollen, weil sonst immer weitere Inhalte geladen werden. (Schwere: 1)

Potenzial wurde beim Bestätigungs-Menü , das auftaucht wenn man Produkte in den Warenkorb oder zu den Favoriten hinzufügt, verschwendet. Durch einen extra Button könnte man den Nutzer*innen erlauben die getäigte Aktion rückgängig zu machen. Diese Funktion würde den Nutzern noch mehr Freiheit gewähren. Hat man beispielsweise fälschlicherweise ein Produkt in den Warenkorb getan, muss man aktuell erst in den Warenkorb um dieses wieder zu entfernen. (Schwere: 1)

Vermeiden von Fehlern

Während der Evaluation konnte keine Mängel bezüglicher dieser Heuristik gefunden werden. (Schwere: 0)

Hilfe und Dokumentation

Die Hilfe kann im Footer gefunden werden. Das ist ein üblicher Platz, wo man eine Hilfe finden würde. Die Hilfe an sich besteht aus zwei Teilen. Teil 1 ist eine Suchfunktion, mit der man nach Fragen und deren Antworten suchen kann. Diese Funktion ist also ein sehr ausführliches FAQ. Teil 2 sind verschiedene Kontaktinformationen. So kann man für verschiedene Themen verschiedene Hotlines anrufen. Außerdem kann man per Chatfunktion mit Support-Mitarbeitern chatten. Zudem zeigt der Webshop den nächsten physischen Shop zum eigenen Standort an, sodass dort vor Ort mit einem Mitarbeiter gesprochen werden kann. (Schwere: 0)

Navigation und Suchfunktion

Die Navigation ist sehr gut. Die Navigation ist simpel und logisch gehalten. Die verschiedenen Kategorien werden bestimmten logischen Bereichen zugeordnet, weswegen man schnell zu den gewünschten Produkten findet. (Schwere: 0)

Die Navigation ist ebenfalls sehr breit angelegt. Durch die geringere Anzahl an Produkten im Vergleich zu Böttcher schafft Nike, dass der/die Nutzer*in nicht tief in den Navigationsbaum eintreten muss. Aufgrund der geringen Tiefe ist es für den/die Nutzer*in auch einfacher zu erkennen wo er/sie sich in der Navigationstruktur befindet. Deswegen ist in diesem Fall die unauffällig Breadcrumb-Funktion zu vernachlässigen. (Schwere: 0)

Besonders hervorzuheben ist die Suchfunktion des Webshops. Dem Nutzer werden passende Vorschläge für den Suchbegriff angegeben. Zusätzlich werden gleich Produkte mit Produktbildern, Name und Preis angezeigt. Dadurch ist die Suche sehr übersichtlich. (Schwere: 0)

Sprache

Nike passt beim Starten des Webshops die Sprache automatisch zur Staatssprache am aktuellen Standort an. Befindet man sich in Deutschland wird man automatisch zu deutschen Variante des Webshops gebracht.

Manuell die Sprache zu ändern ist möglich, jedoch ist diese Funktion versteckt. So befindet sich ganzen unten im Footer ein Link mit dem man manuelle seinen Standort ändern kann. Diese Funktion ist erstens schwer zu finden und zweitens macht die Funktion nicht klar, dass wenn man den Standort wechselt dass dann die Sprache auch wechselt.

Die Funktion die Sprache ändern zu können sollte deutlicher platziert sein. Wenn ein/e Benutzer*in nicht die Sprache ihres aktuellen Standortes spricht wird er/sie Schwierigkeiten haben diese zu wechseln. (Schwere: 2)

Ergebnis

In diesem Kapitel werden die Ergebnisse der Evaluation zusammengefasst. Dabei werden jeweils die drei größten Usability-Probleme der beiden Webshops noch einmal dargestellt. Daraufhin wird die Anzahl der Usability-Probleme pro Schwere der beiden Webshops gegenübergestellt. Anmerkungen aus dem Kapitel 4.3 die mit der Schwere 0 (kein Usability-Problem) gekennzeichnet sind, werden dabei weggelassen. Schlussendlich wird die durchschnittliche Schwere der Usability-Probleme pro Heuristik gegenübergestellt. Bei der Berechnung des Durchschnitts, werden ebenfalls Anmerkungen mit der Schwere 0 weggelassen, da diese das Ergebnis verfälschen würden.

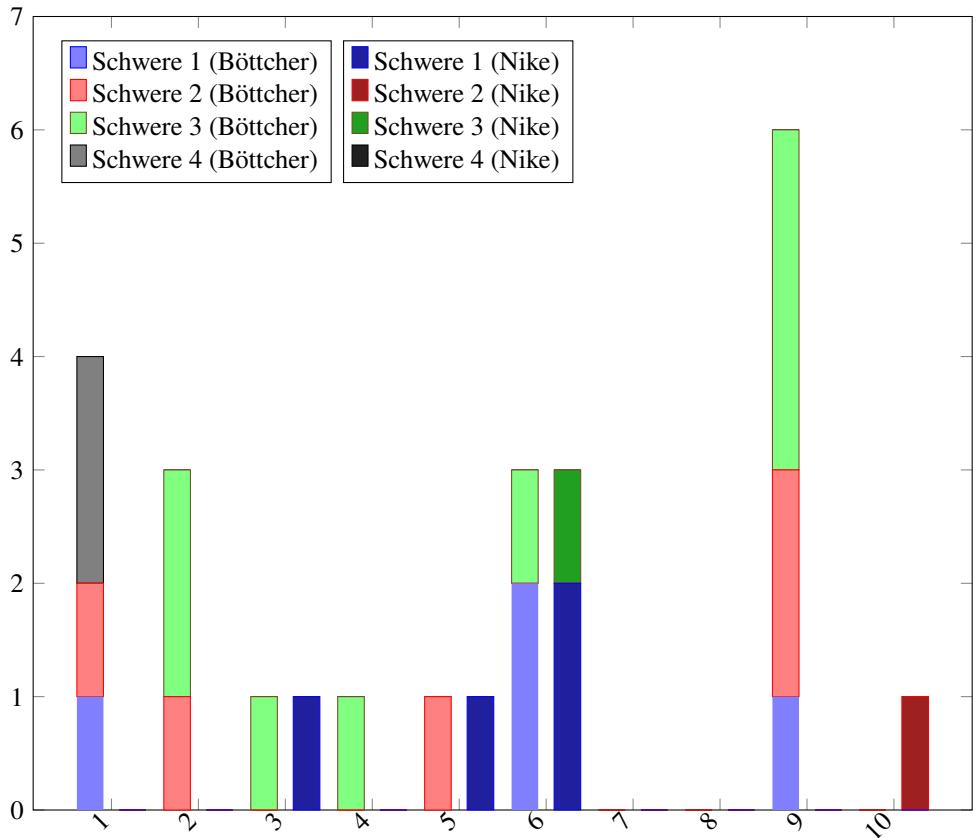
Die drei größten Usability-Probleme des Webshops der Böttcher AG:

1. Webshop ist mit Information und Funktionen überladen (redundante Funktionen, Werbung, ...)
2. Wichtige Informationen wie z.B. die Produktliste sind außerhalb der direkten Sichtfläche der Nutzer*innen, da z.B. Filterfunktionen ihnen den Platz wegnehmen.
3. Keine Konsistenz.

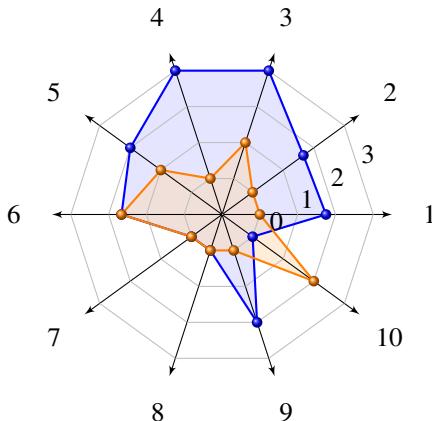
Die drei größten Usability-Probleme des Webshops der Nike Inc.:

1. Unauffällige Breadcrumbs
2. Manuelle Sprachänderung ist nicht intuitiv, da Standort geändert werden muss.
3. Aktueller Gesamtpreis des Warenkorbs wird nur im Warenkorb selbst angezeigt.

Im folgenden Balkendiagramm werden die Anzahl der Fehler, unterteilt in Schwere pro Heuristik, angezeigt. Die Werte der Böttcher AG befinden sich jeweils Links vom X-Wert und die Werte von Nike befinden sich rechts vom X-Wert. Außerdem sind die Werte der Böttcher AG heller als die von Nike. Man kann erkennen, dass für den Webshop von Böttcher wesentlich mehr Probleme identifiziert wurden.



Im folgenden Diagramm wird die durchschnittliche Schwere der Probleme pro Heuristik angezeigt. Dabei werden die Werte von Böttcher blau dargestellt. Die Werte von Nike werden orange dargestellt. Man kann erkennen das die Probleme des Webshops von Böttcher durchschnittlich schwerer ausfallen wie die von Nike. Außerdem erkennt man das bei Böttcher vor allem Probleme bei den Heuristiken 1 bis 6 sowie 9 bestehen. Nikes Probleme liegen bei der Heuristik 6 und 10.



4.4 Vergleich der Anwendbarkeit

Die Methode der Heuristischen Evaluation ist sehr konstengünstig. Aus diesem Grund kann man sie fast immer zur Verbesserung der Usability einsetzen. Vor allem kann man die Methode schon frühzeitig während der Entwicklung des Webshops einsetzen. Ein/e Entwickler*in/Designer*in hat einen geringen Mehraufwand während der Entwicklung die Heuristiken zu beachten.

Mit der Methode schafft man es große Usability-Probleme oder Katastrophen zu verhindern. Je weniger schwer die Probleme jedoch werden, desto höher ist die Gefahr Scheinprobleme zu identifizieren. Möchte man also im Detail auch kleinere Usability-Probleme identifizieren, muss eine qualitativer Methode in Zusammenarbeit mit den Kund*innen gewählt werden.

Ein Thema bei dem die Heuristische Evaluation kein gutes Ergebnis erzielen konnte ist die Navigation. Es ist schwer für die Navigation eine Heuristik zu definieren. Die Navigation hängt stark von der Intuition der Nutzer*innen eines Webshops ab. Deswegen ist die Formulierung einer allgemeingültigen Heuristik für die Navigation schwierig. Für die Verbesserung der Navigation müssen also Methoden verwendet werden, die die Nutzer*innen mit einbeziehen (z.B. Beobachtung oder Befragung).

Gute Ergebnisse erzielte die Methode beim Thema Konsistenz und minimalistisches Design. Bei diesem Heuristiken geht es um die klare und deutliche Darstellung von Informationen. Für diese Aspekte kann man sehr gut allgemeingültige Regeln definieren.

Die Heuristik „Erkennen vor Errinnern“ scheint bei dem Webshop von Böttcher und ähnlichen Webshops weniger relevant. B2B-Geschäfte haben oft Bestandskunden die immer wieder bei dem Webshop Ware nachbestellen. Aus diesem Grund können die Nutzer*innen die Funktion des Webshops erlernen. Deswegen sind sie weniger wahrscheinlich darauf angewiesen Funktionen zu erkennen.

Bei Lifestyle-Shops wie Nike scheint die Heuristik „minimalistisches und ästhetisches Design“ besonders wichtig zu sein. Lifestyleshops sind ausgelegt, dass die Nutzer*innen gerne durch den Webshop stöbern. Durch ein ästhetisches Design und hochwertige Produktbilder bleiben die Nutzer*innen gerne im Shop um diese schönen Dinge zu betrachten.

Schlussendlich kann man sagen das die Methode der Heuristischen Evaluation für den durchschnittlichen Webshop eine gute Methode ist. Für einen besseren Erfolg dieser Methode muss mit dem Heuristikbogen weiter experimentiert werden, um diese noch genauer auf die Usability von Webshops zu zuschneiden. Um den Erfolg dieser Methode messbar zu machen müssen die identifizierten Usability-Probleme umgesetzt werden und einen Vorher-Nachher-Vergleich durchführen.

5 Fazit

Im folgenden Kapitel werden die Ergebnisse der Seminararbeit präsentiert und kritisch beleuchtet. Außerdem wird ein Ausblick gegeben.

Ergebnisse

Es wurden verschiedene Heuristiken der Usability recherchiert und gegenübergestellt. Die Heuristiken wurden in einer Tabelle (1) dargestellt. Daraufhin wurden die konkreten Vorgehensweisen Usability Inspection, Menschzentrierte Gestaltung und der Usability Engineering Lifecycle zur Sicherstellung der Heuristiken erarbeitet und verglichen. Auf Grundlage der Heuristiken und der Vorgehensweisen wurden nun zwei Webshops evaluiert. Hierfür wurden die Heuristische Evaluation verwendet, sowie die erarbeiteten Heuristiken der Tabelle.

Kritische Würdigung

Die Ziele der Arbeit konnten erreicht werden. Zusätzlich zu den erarbeiteten Heuristiken könnten weitere Heuristiken anderer Quellen betrachtet werden. Die Evaluation könnte weiter verbessert werden, indem man die verwendeten Heuristiken und den Fragebogen überarbeitet. Bei der Überarbeitung sollten die Heuristiken noch spezifischer für Webshops angepasst werden. Außerdem konnten der Erfolg der Evaluation nicht messbar überprüft werden. Für eine messbare Überprüfung müssten die Probleme der Webshops verbessert werden und dann ein Vorher-Nachher-Vergleich durchgeführt werden.

Ausblick

Da sich Normen und Standards zu Gestaltung und Usability häufig ändern, sollten die Heuristiken in Zukunft ständig überprüft und bei Bedarf aktualisiert und überarbeitet werden. Dies gilt auch für die Vorgehensweisen zur Evaluierung.

Literatur

- [Bö22] Böttcher: Böttcher AG - die Nr. 1 für B2B Kunden mit günstigen Preisen, <https://www.bueromarkt-ag.de/>, (Accessed on 11/22/2022), 2022.
- [CC02] Chariton, C.; Choi, M.-H.: User Interface Guidelines for Enhancing Usability of Airline Travel Agency E-Commerce Web Sites. In: CHI '02 Extended Abstracts on Human Factors in Computing Systems. CHI EA '02, Association for Computing Machinery, Minneapolis, Minnesota, USA, S. 676–677, 2002, ISBN: 1581134541, URL: <https://doi.org/10.1145/506443.506541>.
- [DIA] DIN Deutsches Institut für Normung e. V., Hrsg.: DIN EN ISO 9241-110:2020-10, Ergonomie der Mensch-System-Interaktion - Teil_110: Interaktionsprinzipien (ISO_9241-110:2020); Deutsche Fassung EN_ISO_9241-110:2020.
- [DIb] DIN Deutsches Institut für Normung e. V., Hrsg.: DIN EN ISO 9241-151:2008-09, Ergonomie der Mensch-System-Interaktion - Teil_151: Leitlinien zur Gestaltung von Benutzungsschnittstellen für das World Wide Web (ISO_9241-151:2008); Deutsche Fassung EN_ISO_9241-151:2008, ST, N.
- [DIC] DIN Deutsches Institut für Normung e. V., Hrsg.: DIN EN ISO 9241-210:2020-03, Ergonomie der Mensch-System-Interaktion - Teil_210: Menschzentrierte Gestaltung interaktiver Systeme (ISO_9241-210:2019); Deutsche Fassung EN_ISO_9241-210:2019.
- [DID] DIN Deutsches Institut für Normung e. V.: Ergonomie der Mensch-System-Interaktion - Teil 11: Gebrauchstauglichkeit: Begriffe und Konzepte (ISO 9241-11:2018); Deutsche Fassung EN ISO 9241-11:2018.
- [Ga18] Gast, O.: User Experience im E-Commerce - Messung von Emotionen bei der Nutzung interaktiver Anwendungen. Springer Gabler Wiesbaden, 2018, ISBN: 978-3-658-22484-4.
- [Ja22] Jakob Nielsen, H. L.: Prioritizing Web Usability, 2022, URL: https://books.google.de/books?hl=de&lr=&id=YQsje6Ec14UC&oi=fnd&pg=PT26&dq=web+usability&ots=nYRGHJNo00&sig=BvShw1GkxYo05yYCUM8vUNiY8Wg&redir_esc=y#v=onepage&q=web%20usability&f=false.
- [Je20] Jendryschik, M.: Gesamtbetrachtung: DIN EN ISO 9241-210 konkretisiert User Experience. iX - Magazin für professionelle Informationstechnik/7/2020, S. 108–111, 2020.
- [KKK18] Kaur, S.; Kaur, K.; Kaur, P.: Analysis of Website Usability Evaluation Methods. In: Dez. 2018.
- [Ma10] Mayhew, D. J.: The usability engineering lifecycle: A practitioner's handbook for user interface design. Morgan Kaufmann, San Francisco, Calif., 2010, ISBN: 9781558605619.
- [Ni94a] Nielsen, J.: Usability engineering. Morgan Kaufmann, San Francisco, 1994, ISBN: 9780125184069.

- [Ni94b] Nielsen, J., Hrsg.: Usability inspection methods. Wiley, New York, 1994, ISBN: 0471018775.
- [Pe22] Peters, L.: E-Commerce net sales of nike.com from 2014 to 2022 | Statista, <https://www.statista.com/forecasts/1218320/nike-revenue-development-ecommerce>, (Accessed on 11/22/2022), Okt. 2022, URL: <https://www.statista.com/forecasts/1218320/nike-revenue-development-ecommerce>.
- [Ru21] Rusche, C.: Die Effekte der Corona-Pandemie auf den Onlinehandel in Deutschland - Institut der deutschen Wirtschaft (IW), <https://www.iwkoeln.de/studien/christian-rusche-die-effekte-der-corona-pandemie-auf-den-onlinehandel-in-deutschland.html>, (Accessed on 11/15/2022), 2021.
- [St22] Statista: Importance of usability regarding online fashion stores in the U.S. 2017 | Statista, 2022, URL: <https://www.statista.com/forecasts/786324/usability-as-criterion-for-online-fashion-stores-in-the-us>.
- [Sü16] Süss, Y.: Usability als wichtiger Erfolgsfaktor von E-Commerce-Webseiten. Wirtschaftsinformatik & Management 8/1, S. 6–15, Feb. 2016, ISSN: 1867-5913, URL: <https://doi.org/10.1007/s35764-016-0017-7>.
- [TK21] Tharindu, P. K.; Koggalage, R.: Usability of E-Commerce Websites: State of the Art and Future Directions. 3/, S. 314–319, Feb. 2021.

Anhang

A Heuristikbogen

1. **Minimalistisches Design**
 - Werden zu jedem Zeitpunkt nur die nötigsten Informationen angezeigt?
 - Werden weitere Informationen ausgelagert (z.B. in Hilfsdialoge)
 - Sind gruppierte Elemente als solche gekennzeichnet?
 - Folgt die Platzierung der Elemente dem natürlichen Lesefluss?
 - Werden wenige schlichte Farben verwendet?
 - Kann man die Benutzeroberfläche auch ohne Farbe nutzen?
 - Werden keine Farben zur Informationsbereitstellung verwendet?
 - Werden Nutzer*innen keine unnötigen Optionen angeboten?
 - Werden die Informationen möglichst auf einen Singel Screen dargestellt?
2. **Übereinstimmung von System und realer Welt**
 - Entspricht die Terminologie des Webshops der Sprache der Nutzer*innen?
 - Werden Icons zur Symbolisierung von Informationen verwendet?
 - Werden merhdeutige Worte vermieden?
 - Werden alle Interaktionen aus der Sicht der Nutzer*innen formuliert?
 - Kann ein Nutzer Alias definieren?
3. **Erkennen vor Errinern**
 - Unterstützt der Webshop den Nutzer beim Errinern von Informationen?
 - Basiert der Webshop aus einer kleinen Menge an Regeln?
4. **Konsistenz und Standardisierung**
 - Führen ähnliche Aktionen zu denselben oder ähnlichen Ergebnissen?
 - Findet man die gleichen Informationen immer an der selben Stelle?
 - Befinden sich Informationen, die sich auf einander beziehen in der gleichen Reihenfolge?
5. **Sichtbarkeit des Systemstatus**
 - Wird der/die Nutzer*in informiert, wie das System einen Input interpretiert?
 - Wird der/die Nutzer*in informiert, was das System mit dem Input macht?

- Werden Aktionen mit langer Antwortzeit gekennzeichnet?
- Werden bei Systemfehlern informative Fehlermeldungen angezeigt?

6. Benutzerkontrolle und Freiheit

- Kann man alle Dialoge verlassen?
- Kann man Aktionen rückgängig machen ?
- Kann man bei Wartezeiten den Prozess abbrechen?

7. Vermeiden von Fehler

- Fängt das System Fehler ab, bevor eine Fehlermeldung kommt?
- Sind Fehlermeldungen klar und verständlich?
- Sind Fehlermeldungen präzise formuliert?
- Hilft die Fehlermeldung dem/der Nutzer*in das Problem zu lösen?
-

8. Hilfe und Dokumentation

- Ist die Dokumentation vollständig und verständlich?
- Ist die Dokumentation zugänglich?
- Ist die Dokumentation auf dem aktuellsten Stand?
- Stellt die Dokumentation Kontaktinformationen bereit?

9. Navigation und Suche

- Macht die Navigation deutlich wo sich der/die Benutzer*in in der Navigationsstruktur befindet?
- Werden verschiedene Navigationsstrategien angeboten?
- Ist die Navigation einfach gehalten?
- Sind Verknüpfungen logisch gruppiert und sinnvoll benannt?
- Ist die Navigationsstruktur breit angelegt?
- Ist die Navigationsstruktur konsistent?
- Bietet der Webshop eine Suchfunktion an?
- Ist die Suche global zugänglich?
- Sind die Suchergebnisse sinnvoll angeordnet?
- Sind die Suchergebnisse ausreichend gut beschrieben?
- Kann die Suche weiter eingeschränkt werden?

10. **Sprache**

- Werden verschiedene Sprachen unterstützt?
- Ist die Sprache global einstellbar?

XaaS: Künstliche Intelligenz im Trend von Everything-as-a-Service

Niklas Arnold¹ Luca Negron-Martinez²

Abstract: Diese Seminararbeit widmet sich dem Thema Everything-as-a-Service, oder kurz XaaS. Hierbei werden kurz die drei Hauptkomponenten davon erläutert und die Frage, was XaaS überhaupt ist geklärt. Danach wird dann der Fokus auf den Bereich von AIaaS, also der künstlichen Intelligenz als Service gelegt. Dabei werden verschiedene Einsatzbereiche und Anwendungsmöglichkeiten aufgezeigt und welche Anbieter welche Systeme zur Verfügung stellen. Diese Darstellung wird mit Hilfe einer Marktanalyse realisiert. Bei AIaaS gibt es viele Vor-, jedoch auch einige Nachteile bzw. Risiken. Pro- und Contra-Argumenten werden mit einer Argumentation gegenübergestellt und analysiert. Zusammenfassend wird dargestellt, warum AIaaS immer weiter verbreitet ist und wie sich der Nutzen in der Zukunft weiterentwickeln könnte.

Keywords: XaaS; AIaaS; Künstliche Intelligenz

1 Einleitung

1.1 Motivation

Unsere Welt befindet sich in einem andauernden Umbruch. Immer mehr befindet sich in einem stetigen Wandel. Alles wird schneller und kurzlebiger. So ist es auch in der Informatik, speziell in der Softwareentwicklung bzw. der Bereitstellung von Software. Die Anforderungen an ein heutiges System sind vor allem Flexibilität und Agilität. Hier kommt die Idee von XaaS ins Spiel. Unter XaaS versteht das Prinzip von Everything-as-a-Service. Unter diesem Ansatz versteht man hauptsächlich Dienstleistungen rund um Infrastruktur (IaaS), Software (SaaS) und Plattformen (PaaS) als einen Service zu beziehen. Erweitert reicht XaaS aber auch bis hin zum Einsetzen von künstlicher Intelligenz als Service (AIaaS). Bei einer solchen Umstellung auf serviceorientierte Betriebsmodelle kann der Verwaltungsaufwand erheblich sinken, da man sich nicht mehr um die Wartung und Instandhaltung der genutzten Dienste kümmern muss. Außerdem können durch ein solches Modell die Anwendungen und Systeme nach Bedarf skaliert werden. Dadurch entfallen die Kosten für die Altsysteme und es entstehen ausschließlich neue Kosten für die in Anspruch genommenen Leistungen.

¹ DHBW Stuttgart Campus Horb, Informatik, Florianstraße 15, 72160 Horb am Neckar, Deutschland i20002@hb.dhbw-stuttgart.de

² DHBW Stuttgart Campus Horb, Informatik, Florianstraße 15, 72160 Horb am Neckar, Deutschland i20024@hb.dhbw-stuttgart.de

1.2 Fragestellung

Immer mehr Anwendungen basieren auf einer Künstlichen Intelligenz. Ob es diverse Bilderkennungstools, Chatbots oder maschinelle Übersetzer sind, immer mehr setzt man auf die neuronalen Netze. Bei einer genaueren Betrachtung von XaaS ergeben sich mehrere Fragen, welche es zu beantworten gilt. Zum einen, was ist Everything-as-a-Service? Darunter zählt was man unter XaaS versteht, sowie in welche Unterpunkte man den Begriff untergliedern kann. Der Fokus soll dabei auf AIaaS, also Artificial Intelligence-as-a-Service (Künstliche Intelligenz als Service), liegen. Hierbei wird auf den Nutzen, die möglichen Anwendungsbereiche, sowie die Vor- und Nachteile eingegangen, um abwägen zu können, welche Faktoren AIaaS bzw. XaaS allgemein begünstigen und weshalb sie von immer mehr Unternehmen eingesetzt werden. Außerdem soll analysiert werden, auf welche Serviceanbieter hier in der Regel zurückgegriffen wird, welche Funktionen und Tools sie anbieten und wie dabei das jeweilige Preismodell aufgebaut ist.

2 Everything-as-a-Service

2.1 Was ist XaaS?

Everything-as-a-Service, auch als XaaS bekannt, ist ein Begriff, der die Bereitstellung einer breiten Palette von Diensten über das Internet beschreibt. Diese Dienste können über herkömmliche webbasierte Anwendungen oder, was häufiger gemeint ist, über modernere cloudbasierte Plattformen bereitgestellt werden. XaaS kann alles umfassen, von Speicher- und Sicherungsdiensten über Software und Anwendungen bis hin zu Infrastruktur- und Plattformdiensten. [vgl. Kol20, 986f.] Im Grunde genommen kann jeder Dienst, der über das Internet bereitgestellt werden kann, als XaaS betrachtet werden. Im weitesten Sinne ist auch Human-as-a-Service (HuaaS) miteinbezogen, also das Nutzen von menschlicher Intelligenz über das Internet wie einen herkömmlichen Webservice. [vgl. Com22] Der Begriff wird im Allgemeinen verwendet, um den Wechsel von herkömmlichen, vor Ort installierten Software- und Hardwarelösungen zu cloudbasierten Lösungen zu beschreiben. Mit XaaS können Unternehmen nur für die Dienste zahlen, die sie benötigen, wenn sie sie benötigen, ohne im Voraus in teure Hardware und Software investieren zu müssen. Der Begriff wird im Allgemeinen verwendet, um den Wechsel von herkömmlichen, vor Ort installierten Software- und Hardwarelösungen zu cloudbasierten Lösungen zu beschreiben. Mit XaaS wird es Unternehmen ermöglicht nur für die Dienste zu zahlen, die sie benötigen, wenn sie sie benötigen, ohne im Voraus in teure Hardware und Software investieren zu müssen. XaaS kann eine kostengünstige Möglichkeit für Unternehmen sein, die benötigten Dienste zu erhalten, ohne in die Infrastruktur zur Unterstützung dieser Dienste investieren zu müssen. Außerdem können Unternehmen dadurch flexibler werden, da sie ihre Dienste je nach Bedarf auf- oder abbauen können.

Klassisch betrachtet, spricht man bei XaaS von drei aufeinander aufbauenden Schichten. Dabei beinhaltet eine Ebene auch immer die Eigenschaften der Darunterliegenden.

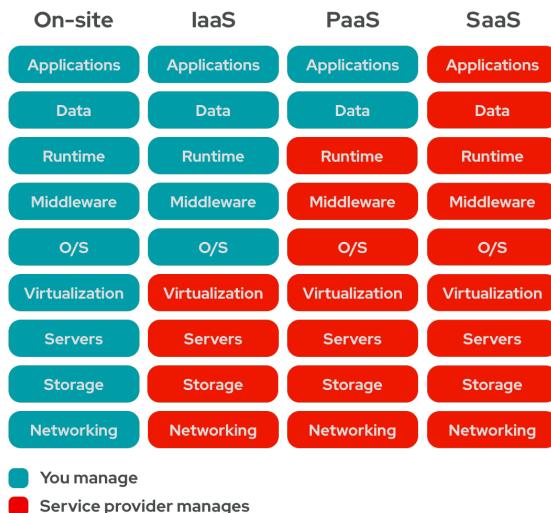


Abb. 1: Stufenartig aufgebautes Cloud Computing Modell [Red22]

Die drei Hauptkomponenten sind Infrastructure-, Plattform- und Software-as-a-Service. Wie in der Abbildung zu sehen ist kommen bei jeder Stufe im Gegensatz zu dem klassischen On-Site-Ansatz, bei dem jegliche Infrastruktur selbst beschafft und vor Ort gehostet wird, Services dazu, die die Service Provider verwalten und nicht mehr selbst gemanaged und gewartet werden müssen. [vgl. Red22]

2.2 Infrastructure-as-a-Service (IaaS)

Infrastructure-as-a-Service ist die unterste Ebene des dreischichtigen Servicemodells von Cloud Computing. Man versteht darunter das Modell, bei dem ein Drittanbieter eine Computerinfrastruktur als Dienst anbietet. Das heißt, bei IaaS wird das Hosting geoutsourced, wobei der IaaS-Kunde nicht der Eigentümer der Infrastruktur ist, sondern diese bei einem Provider mietet. Die IaaS-Anbieter stellen diese Ressourcen dann auf Abruf aus ihren umfangreichen, in Rechenzentren installierten Gerätelpools bereit. Dieser ist dann auch für die Wartung und den Support der Hardware und Software verantwortlich, so dass sich der Kunde auf sein Kerngeschäft konzentrieren kann. Die Kunden zahlen nur für die Menge an Rechen-, Speicher- und Netzwerkressourcen, die sie nutzen, in der Regel auf einer Pay-as-you-go-Basis. Dies steht im Gegensatz zum traditionellen Modell des Kaufs und der on-premise Verwaltung von Hardware und Software, die kapitalintensiv sein können und erhebliche laufende Wartungs- und Supportkosten erfordern.

Zusammenfassend lässt sich sagen, dass IaaS ein Cloud-Computing-Modell ist, das es Unternehmen ermöglicht, das Hosting und die Wartung ihrer Hardware- und Software-Ressourcen auszulagern. Es bietet Unternehmen die Möglichkeit, Ressourcen schnell und bedarfsgerecht zu skalieren, auf die neuesten Technologien zuzugreifen, ohne in teure Hardware- und Software-Upgrades investieren zu müssen, und individuelle Lösungen für ihre spezifischen Anforderungen zu entwickeln. [vgl. Hua23, S. 31]

2.3 Platform-as-a-Service (PaaS)

Die nächste Ebene im Cloud Computing Modell ist Platform-as-a-Service oder auch PaaS. PaaS-Anbieter bieten eine Umgebung an, in der Anwendungen entwickelt, ausgeführt und verwaltet werden können, ohne dabei die zugrundeliegende Infrastruktur aufzubauen zu müssen. PaaS-Dienste ermöglichen es den Benutzern, über eine Internetverbindung auf eine Plattform und alle zugehörigen Dienste zuzugreifen, wodurch die Installation, Konfiguration und Verwaltung von Hardware und Software entfällt. Dies kann die Kosten senken und den Entwicklungsprozess beschleunigen. PaaS-Dienste umfassen in der Regel Betriebssysteme, Webserver, Anwendungsserver, Datenbanken, Speicherdienste und andere Dienst zur Erstellung und Bereitstellung von Anwendungen.

Zusammenfassend spricht man bei PaaS häufig auch von einem „Cloud-Betriebssystem“. PaaS kann für die schnelle Entwicklung und Bereitstellung von Webanwendungen, mobilen Anwendungen und anderen Softwareanwendungen verwendet werden, ohne dass dabei die Komplexität der Verwaltung der zugrundeliegenden Infrastruktur auf sich genommen werden muss. [vgl. Hua23, 31ff.]

2.4 Software-as-a-Sercice (SaaS)

Auf der obersten Ebene des dreistufigen Cloud Computing Modells und somit die umfangreichste Bereitstellung von Diensten ist Software-as-a-Service oder auch SaaS. SaaS ist eine Softwarebereitstellungsmethode, bei der die gesamte Software, also inklusive Infrastruktur und Betriebssystem, auf entfernten Servern zentral gehostet wird und die Nutzer über das Internet mit einem standard Webbrowser darauf zugreifen können. SaaS-Anwendungen werden manchmal auch als webbasierte Software, On-Demand-Software oder gehostete Software bezeichnet. Mit SaaS können die Kunden von jedem Ort, zu jeder Zeit und mit jedem Gerät, das über eine Internetverbindung verfügt, auf die Software zugreifen, ohne die Software auf ihren eigenen Servern oder Geräten installieren und verwalten zu müssen. SaaS-Lösungen sind in der Regel abonnementbasiert und werden auf monatlicher oder jährlicher Basis bezahlt.

Diese Dienste sind sowohl für allgemeine Nutzer (Anwendungen wie Google Calendar und Gmail), als auch für Unternehmensgruppen zur Unterstützung von Gehaltsabrechnungen, Personalverwaltung, und für die Verwaltung von Kunden- und Geschäftspartnerbeziehungen. Durch diese Anwendungen wird der Zeitaufwand für die Installation und Wartung der Software reduziert und es muss auch keine Hardware, Software und spezielles IT-Personal beschafft werden. [vgl. Hua23, 33f.]

3 Begünstigende Faktoren von Servicedienstleistungen

Verschiedene Servicedienstleistungen, speziell im Bereich des Cloud Computings bzw. genauer XaaS, gewinnen in den letzten Jahren immer mehr an Popularität. Aber warum?

Für das rapide Wachstum gibt es eine Vielzahl von Faktoren. Einer der Hauptgründe ist die Kosteneffizienz. XaaS macht kostspielige Hardware- und Softwareinvestitionen sowie die damit verbundenen Wartungs- und Supportkosten überflüssig. Darüber hinaus ermöglicht das flexible Abonnementmodell von XaaS den Unternehmen, ihre Dienste je nach Bedarf zu erweitern oder zu reduzieren, ohne dass zusätzliche Kosten anfallen. Schließlich bieten EaaS-Lösungen auch einen einfachen Zugang zu den neuesten Technologien und Funktionen, d.h. die Unternehmen können ohne größere Vorabinvestitionen schnell von den neuesten Innovationen profitieren. Das ist auch ein weiterer Punkt: die Bequemlichkeit. Durch Cloud-Dienste können Unternehmen jederzeit und von überall auf ihre Daten und Anwendungen zugreifen und diese nutzen. Dies erleichtert die Zusammenarbeit, die gemeinsame Nutzung von Informationen und die Anpassung an sich verändernde Kundenbedürfnisse.

Diese gesteigerte Vernetzung ist nur durch die in den vergangenen Jahren stark gestiegene Bandbreite die den Unternehmen, aber auch Privatpersonen, zur Verfügung stehen zu erreichen. Erst durch diese wird eine Verbindung untereinander und das Nutzen von cloudbasierten Diensten ermöglicht. Das ist wohl der entscheidendste Punkt der das rapide Wachstum von Cloud Computing erklärt und weiterhin auch antreibt. [vgl. HL16]

4 Artificial Intelligence-as-a-Service (AIaaS)

Ein stark wachsender Bereich in der Digitalisierung und der Automatisierung ist die Künstliche Intelligenz. Sie kann für eine Vielzahl von Aufgaben eingesetzt werden. Sie kann zum Beispiel Analysen und Erkenntnisse liefern, Maschinen helfen Menschen zu verstehen und mit ihnen zu interagieren oder alltägliche Aufgaben automatisieren. KI wird auch im Gesundheits-, Bank- und Finanzwesen eingesetzt, um Prozesse zu automatisieren, Prognosemodelle zu erstellen und die Kundenerfahrung zu verbessern.

Das Erstellen und Trainieren von KI-Modellen kann allerdings sehr zeitaufwendig und gegebenenfalls auch sehr komplex sein. Außerdem benötigen KI-Anwendungen in der Regel eine hohe Rechenleistung. Somit ist es häufig in kleinen bis mittleren Unternehmen nicht gegeben, dass zum einen das notwendige KI-KnowHow und KI-Entwickler vorhanden sind und zum anderen auch die benötigte Infrastruktur nicht gegeben ist. Aus diesen Gründen gibt es auch hier eine Nachfrage nach einem cloudbasierten Produkt, mit welchem sich, ohne großes Vorwissen und Aufwand, künstliche Intelligenzen in die eigenen Anwendungen einbinden lassen. Die Lösung dafür nennt sich Artificial Intelligence-as-a-Service, oder kurz AIaaS. [vgl. Lin+21, 441f.]

Artificial Intelligence-as-a-Service (AIaaS), oder Künstliche Intelligenz als Service, ist ein Cloud-basierter Dienst, der es Unternehmen ermöglicht, Funktionen einer künstlichen Intelligenz in ihre Anwendungen und Systeme zu integrieren, ohne die erforderliche Infrastruktur aufzubauen oder warten zu müssen. AIaaS-Lösungen können eine Reihe von Funktionen bieten, von der Verarbeitung natürlicher Sprache, Bilderkennung und maschinellem Lernen bis hin zu prädiktiven Analysen und der Automatisierung von Robotikprozessen. Sie können Unternehmen in die Lage versetzen, ihre betriebliche Effizienz zu steigern, Kosten zu senken und den Kundenservice zu verbessern.

Für den Zugriff auf die AIaaS-Lösung zahlen Unternehmen in der Regel eine Abonnementgebühr an den Anbieter, und sie können dann über eine API oder eine andere Integrationsmethode auf die KI-Funktionen zugreifen. Diese Art von Service kann Unternehmen die Ressourcen zur Verfügung stellen, die sie benötigen, um die Entwicklung von KI-Anwendungen zu beschleunigen, ohne in kostspielige Infrastruktur und Technologie investieren zu müssen. [Cuo22, vgl.]

5 Globaler AlaaS-Markt

Der globale Markt für Artificial Intelligence as a Service steigt immer weiter an. Die gesamte Marktgröße lag 2021 noch etwa bei 4,7 Milliarden US-Dollar, allerdings wird prognostiziert, dass der globale Markt bis 2030 auf ca. 92 Milliarden US-Dollar anwachsen könnte. Das entspräche, über einen Zeitraum von acht Jahren, einer durchschnittlichen jährlichen Wachstumsrate von 39,16 Prozent. Zu den wichtigsten Faktoren, die den Markt vorantreiben, gehören unter anderem der Bedarf und die Nachfrage an kostengünstigen KI-Lösungen von kleinen und mittelständigen Unternehmen, die Notwendigkeit die Markteinführungszeit von KI-Anwendungen zu verkürzen und der Mangel an Firmeninterner KI-Expertise. Zu den wichtigsten Akteuren auf dem Markt zählen Google LLC, IBM Corporation, Microsoft Corporation und Amazon Web Services Inc. All diese Akteure haben verschiedene Konzepte, Herangehensweisen und Wachstumsstrategien wie Partnerschaften, Kooperationen und neue Produkteinführungen, um ihre Präsenz auf dem globalen Markt für künstliche Intelligenz als Dienstleistung zu etablieren und zu erweitern. Eine aussagekräftige Marktanalyse zu den jeweiligen Marktanteilen ist nicht möglich, da die Unternehmen ihre Umsatzzahlen im Bereich von AlaaS nicht einzeln veröffentlichen. Man weiß lediglich den gesamten Umsatz den sie mit künstlicher Intelligenz generieren, wovon AlaaS wohl nur ein Bruchteil davon ausmacht. [vgl. Pre21]

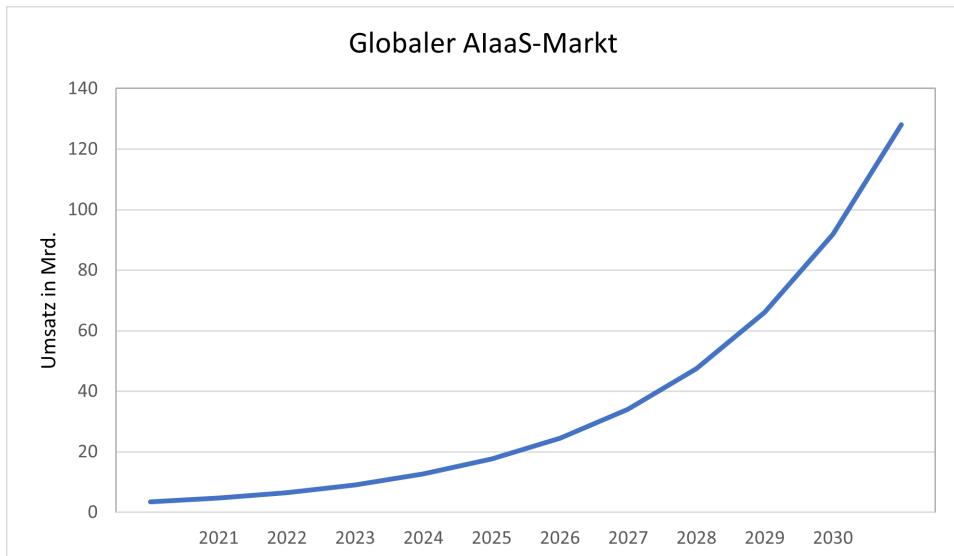


Abb. 2: Prognostizierter, durchschnittlicher Marktverlauf bis 2030

5.1 Google LLC

Google bietet eine Reihe von Diensten für künstliche Intelligenz an, mit denen Entwickler ihren Anwendungen problemlos intelligente Funktionen hinzufügen können. Zu diesen Diensten gehören vorab trainierte Modelle für maschinelles Lernen, die für Aufgaben wie Bilderkennung, Verarbeitung natürlicher Sprache und prädiktive Analysen verwendet werden können. Google bietet auch eine Cloudbasierte Entwicklungsumgebung für die Erstellung und das Training benutzerdefinierter maschineller Lernmodelle. Dieser Ansatz erleichtert Entwicklern den Einstieg in die KI, ohne dass sie in teure Hardware oder Software investieren müssen. Zu den von Google angebotenen Produkten gehören unter anderem die Google Cloud Natural Language API, Cloud Speech API, Cloud Translation API und Cloud Vision API. Diese Dienste ermöglichen es Entwicklern, Anwendungen zu erstellen, die menschliche Sprache verstehen, Sprache in Text umwandeln, zwischen Sprachen übersetzen und Objekte in Bildern identifizieren können. Dabei können die Services mit einer simplen API-Anbindung in die eigenen Softwareprodukte eingebunden werden. [vgl. Goo22b]

5.2 Microsoft Corporation

Microsofts Ansatz für AIaaS bietet Entwicklern eine Reihe von Möglichkeiten, KI-Funktionen in ihre Anwendungen mit einzubinden. Dabei werden über eine cloudbasierte Plattform Bereiche wie die Bildverarbeitung, Sprachverarbeitung, Übersetzungen und Suchen bereitgestellt. Microsofts Ansatz für KI als Service zielt darauf ab, das Hinzufügen von KI-Fähigkeiten zu Anwendungen zu erleichtern, ohne dass dafür tiefgreifende KI-Kenntnisse erforderlich sind. Das Unternehmen bietet auch eine Reihe von Tools und Ressourcen, die Entwickler für die Erstellung von KI-Anwendungen nutzen können, darunter zählt beispielsweise die Azure Machine Learning-Plattform. Ein Key-Product von Microsoft im Bereich von Künstlicher Intelligenz als Service ist die Microsoft Azure Cognitive Services Suite. Diese stellt eine Reihe von APIs in den bereits genannten Gebieten: Bild- und Sprachverarbeitung, Übersetzungen und Suchen bereit, mit denen intelligente Funktionen durch einfache API-Anfragen in Anwendungen hinzugefügt werden können. Ein weiteres Produkt ist die Cortana Intelligence Suite. Diese umfasst intelligente Dienste zur Datenanalyse, zum maschinellen Lernen und zur Verarbeitung natürlicher Sprache. [vgl. Azu22]

5.3 Amazon Web Services Inc.

Amazons Ansatz der künstlichen Intelligenz als Service ist ein Cloud-basierter Dienst, der Entwicklern Tools zur Erstellung und Bereitstellung von KI-Anwendungen zur Verfügung stellt. Der Service bietet eine Vielzahl von Funktionen, darunter vortrainierte Modelle, eine benutzerfreundliche Oberfläche und Unterstützung für zahlreiche Programmiersprachen. Amazon bietet eine Reihe von KI-Diensten an, die es Entwicklern ermöglichen, anspruchsvolle Anwendungen mit Funktionen der künstlichen Intelligenz zu erstellen. Zu diesen Services gehören Amazon Lex, Amazon Polly, Amazon Rekognition und Amazon SageMaker. Amazon Lex ist ein Service, der es Entwicklern ermöglicht, Conversational Bots zu erstellen. Er nutzt die Verarbeitung natürlicher Sprache (NLP), um die Absichten der Benutzer zu verstehen und Benutzeranfragen zu erfüllen. Amazon Polly ist ein Text-to-Speech-Service, der Text in lebensechte Sprache umwandelt. Er unterstützt eine Reihe von Sprachen und kann verwendet werden, um Anwendungen zu erstellen, die sprechen. Das kann beispielsweise beim Erstellen einer Barrierefreien Software helfen. Amazon Recognition ist ein Bilderkennungsdienst, der zur Identifizierung von Objekten, Personen und Szenen in Bildern verwendet werden kann. [vgl. Ama22b]

5.4 IBM Corporation

Der IBM-Ansatz für künstliche Intelligenz als Service basiert auf der Watson-Plattform des Unternehmens. Neben der „klassischen“ Möglichkeit eigene KI-Modelle zu trainieren und sie dann in einer Cloud-Umgebung einzusetzen, umfasst die Plattform auch eine Reihe von APIs, die es Entwicklern ermöglicht, von ihren eigenen Anwendungen aus auf die Fähigkeiten von Watson zuzugreifen. Einer der populärsten Bereiche, die die IBM Watson Services abdecken ist das maschinelle Lernen. Dabei wird Unternehmen die Möglichkeit geboten, automatisch aus Daten zu lernen und die eigenen Prozesse zu verbessern. Die Services nutzen eine Vielzahl von Algorithmen, um Unternehmen in die Lage zu versetzen, Vorhersagemodelle zu erstellen, die für eine Reihe von Aufgaben wie Betrugserkennung, Kundensegmentierung und vorausschauende Wartung verwendet werden können. Ein weiteres wichtiges Segment ist die Verarbeitung natürlicher Sprache. Hier wird es zum einen ermöglicht, aus unstrukturierten Daten wie Text und Audio eine Bedeutung zu extrahieren. Es ist zum anderen aber auch möglich die Services für Stimmungsanalysen, Inhaltsklassifizierungen und Entitätsextraktion zu verwenden. IBMs Lösungen für die Informationsextraktion aus Bildern nennt man IBM Computer Vision Services. Sie können für Aufgaben wie Objekterkennung, Bildklassifizierung und Bildsuche genutzt werden. [vgl. Wat21]

5.5 Preisvergleich

Jeder Anbieter bietet verschiedene Preis- und Abrechnungsmodelle an und stellt verschiedene KI-Lösungen zur Verfügung. Ein eindeutiger Preisvergleich über alle Funktionen zwischen den verschiedenen Unternehmen ist daher nur begrenzt möglich. Um trotzdem einen direkten Vergleich zu erhalten, haben wir uns die Speech-to-Text-Lösungen der einzelnen Anbieter angeschaut. Diese wurde gewählt, da sie von allen Firmen angeboten wird und es sich hierbei um eine Standardlösung handelt, die in vielen Bereichen angewendet werden kann und kein „Nischenprodukt“, das speziell für eine Branche entwickelt wurde, ist. Bei Speech-to-Text wird gesprochene Sprache erkannt und in geschriebenen Text umgewandelt. Die Anwendungsgebiete dafür reichen von intelligenten persönlichen Assistenten wie Apples Siri oder Amazons Alexa, über das Mitschreiben und Dokumentieren von Meetings oder Interviews, bis hin zur barrierefreien Gestaltung von Softwareprodukten, um Menschen mit Behinderung einen leichteren Zugriff auf digitale Inhalte zu ermöglichen. Zusätzlich gibt es zum Teil Zusatzfunktionen, die das Gesprochene gleichzeitig noch übersetzen oder sogar den Sprechenden verifizieren oder identifizieren können.

Google bietet seinen Kunden eine kostenlose Nutzung der Speech-to-Text-API von 60 Audiominuten pro Monat. Für alles darüber hinaus berechnet Google pro 15 Sekunden mindestens 0,004 \$. Das sind 0,016 \$ pro Minute. Dabei wird jede Anfrage auf die nächsten vollen 15 Sekunden aufgerundet. Bestimmt wird der endgültige Preis durch weitere Faktoren wie die Spracherkennung mit einem erweiterten KI-Modell, dem aktivieren von Daten-Logging oder der Anzahl der erkannten Audiokanäle der Daten. Begrenzt ist die Nutzung der API auf 1.000.000 Audiominuten pro Monat. Möchte man mehr Sprache erkennen, muss man bei Google eine Kontingentanfrage einreichen und Google über seinen Bedarf aufklären. [vgl. Goo22a]

Amazon Transcribe bietet seinen Kunden ebenfalls 60 Freiminuten pro Monat, allerdings nur für ein Jahr. Spätestens danach muss man auch hier nach Audiominuten dafür bezahlen. Hierbei ist das Preismodell Stufenweise mit drei verschiedenen Tier-Stufen aufgebaut. Für die ersten 250.000 Minuten im Monat zahlt man 0,024 \$/Minute. Für die nächsten 750.000 0,015 \$/Minute und für jede weitere Minute über 1.000.000 0,0102 \$. [vgl. Ama22a]

Bei den Microsoft Speech Services erhalten Kunden in der kostenlosen Version pro Monat fünf Audiostunden umsonst. Um mehr Daten zu verarbeiten, bietet Microsoft diesen Service für einen Dollar pro Audiostunde an. Das entspricht etwa 0,0167 \$/Minute. Außerdem bietet Microsoft auch verschiedene „Commitment Tiers“ an. Darauf muss man sich bewerben und kann dann für einen fixen Preis eine fixe Anzahl an Audiostunden erwerben. So zahlt man für 2.000 Stunden 1.600 Dollar, für 10.000 Stunden 6.500 Dollar und für 50.000 Stunden 25.000 Dollar. [vgl. Mic22]

Auch IBM mit dem Watson Speech-to-Text-Service bietet verschiedene Preispläne an. Mit dem „Lite“-Plan erhält man kostenlos pro Monat 500 Audiominuten. Mit dem „Plus“-Plan zahlt man bis zu 1.000.000 Minuten 0,02 Dollar pro Minute, alles darüber kostet dann nur noch die Hälfte, nämlich 0,01 Dollar pro Minute. Zusätzlich gibt es noch die Pläne

„Premium“ und „Deploy Anywhere“. Diese bieten weitere Zusatzfunktionen, wie zum Beispiel eine zusätzliche Datenverschlüsselung. Die Preise dieser Pläne sind allerdings nur auf Nachfrage einsehbar. [vgl. IBM21]

Die Tabelle (siehe Tabelle 1) fasst die Preise der Anbieter zusammen.

	kostenlos	Weiterführend	Begrenzung
Google	60min/M	0.016\$/min	1.000.000min/M
Amazon	60min/M (1 Jahr lang)	0.024\$/min(bis 250.000min) 0,015\$/min(bis 1.000.000min) 0.0102\$/min(danach)	keine
Microsoft	300min/M (kostenlose Version)	Einfacher Gebrauch: 0,0167\$/min Commitment Tiers: 1.600\$ für 2.000h 6.500\$ für 10.000h 25.000\$ für 50.000h	keine
IBM	Lite-plan 500min/M	0.02\$/min (für 1.000.000min) 0.01\$/min (danach)	keine

Tab. 1: Preisvergleich

6 Vor- und Nachteile von AIaaS

6.1 Vorteile

AIaaS zu nutzen kann viele Vor- aber auch Nachteile mit sich bringen.

Wird AIaaS in einem Unternehmen genutzt, können Risiken verhindert werden, die in Verbindung mit dem Entwickeln und Warten von eigenen KI-Infrastrukturen auftreten könnten. Beim Neuentwickeln eigener KI's ist keine Sicherheit gegeben, dass diese von Anfang an wie gewünscht funktioniert. Die Entwicklung von KI's kann sehr komplex werden und kann deshalb bei Unternehmen, die bisher eher geringe Berührungs punkten mit KI's hatten zu Fehleinschätzungen führen. Ein weiteres Risiko wäre zum Beispiel

der Kontrollverlust der KI. Wird unbedacht oder blind auf diese Systeme vertraut, kann die Unabhängigkeit und Entscheidungsfreiheit verloren werden. Mit Nutzung von AIaaS können Risiken wie diese zwar nicht ausgelöscht werden, jedoch ist dort ein Know-how vorhanden, welches man selbst möglicherweise nicht besitzt.

Ein bekannter Fall, bei dem sich eine Künstliche Intelligenz selbstständig machte, ist ein Forschungsprojekt aus dem Jahr 2016. Hier versuchten zwei Forscher aus dem Google Brain Team zwei Maschinen miteinander kommunizieren zu lassen. Die Aufgabe dabei war, das Die Maschinen namens Bob und Alice Nachrichten verschicken, die nur der Gegenüber entziffern kann. Dagegen arbeitete eine dritte Maschine Namens Eve, die versuchte die Nachrichten zu entziffern. Anfangs konnte Eve viele Nachrichten entziffern, jedoch lernte Bob schnell und nach etwa 15 000 Durchläufen hatte Eve keine Chancen mehr die Nachrichten immer korrekt zu entziffern. Bob und Alice hatten einen Verschlüsselungsalgorithmus gefunden, den nur die beiden entschlüsseln konnten. Selbst die Wissenschaftler des Google-Teams konnten die Nachrichten nicht mehr nachvollziehen, da sie vergessen hatten zu implementieren, dass die Bots sich ausschließlich auf Englisch unterhalten sollten und mussten das Projekt vorerst abbrechen. [vgl. Beu16] [vgl. Phi17]

Außerdem kann durch das Nutzen von AIaaS Diensten den Unternehmen bei der Einhaltung von Datenschutz- und Sicherheitsvorschriften helfen, da diese mit den Regelungen und Vorschriften vertraut sein sollten. Falls personenbezogene Daten vom Dienstleister verarbeitet werden, muss der Nutzer einen Auftragsverarbeitungs-Vertrag schließen. Der abgeschlossene AV-Vertrag regelt die Rechte und Pflichten des Auftraggebers und Auftragnehmers sowie etwaiger eingesetzter Sub-Dienstleister. Es sollte sichergestellt werden, dass Auftragnehmer die einem anvertrauten Daten nur für den Zweck verarbeitet werden, für den der Auftraggeber die Daten erhoben hat. Vor allem aber sind Dienstleister verpflichtet, Daten in angemessenem Umfang zu schützen. Um dies in der Praxis sicherzustellen, gibt der Vertrag dem Kunden diesbezüglich umfassende Kontrollrechte. [vgl. Act22]

Mit AIaaS können KI-Fähigkeiten schnell und effizient skaliert werden. Das bedeutet, dass bei hohem Gebrauch der Dienste, mehr Cloud-Speicher dazu gebucht werden kann. Das gleiche gilt auch für die andere Richtung. Werden weniger Kapazitäten benötigt, so kann die Nutzung herunter skaliert werden. Testet ein Unternehmen beispielsweise mehrere Anbieter, um zu sehen, welche am bestem geeignet für es ist, so werden geringere Kapazitäten benötigt, wie bei einer Vollauslastung auf ein anstehendes Projekt bei einem Anbieter. [vgl. Cuo22]

Die Lufthansa Industry Solutions beschreiben die Skalierbarkeit ihrer Service mit folgendem Satz: „AIaaS kann je nach Bedarf und Wachstum des Unternehmens nach oben oder unten skaliert werden. Sämtliche KI-Tools lassen sich einzeln hinzufügen, sodass Unternehmen auch mit kleinen Schritten den Weg zur Nutzung von künstlicher Intelligenz gehen können“. [vgl. Sol22]

Ein weiterer Vorteil für die Nutzung von AIaaS ist der Zugang zu den neuesten und fortschrittlichsten KI-Technologien für die Unternehmen. Anbieter von AIaaS müssen

stets auf dem aktuellen Stand sein, um auf dem Markt kompetitiv zu bleiben. Dadurch kann als Verbraucher immer eine verlässliche und fortschrittliche Technik erworben werden. Im Vergleich zu einer eigenen Implementierung wird auch hier viel Zeit gespart. Es würde einige Zeit dauern, um qualitativ auf das Level der Anbieter in dem Gebiet zu kommen. [vgl. Fol22]

Der wohl größte Vorteil, der sich durch AIaaS ergibt, sind die Kosteneinsparungen. Durch Nutzung einer KI-Dritter, werden die kompletten Kosten für Entwicklung und Wartung einer eigenen KI eingespart. Es müssen ebenfalls keine Mitarbeiterkosten dafür gezahlt werden. Ein weiterer Vorteil, der sich dadurch ergibt, ist dass man sich auf seine Kernkompetenzen konzentrieren und die Pflege der KI-Infrastruktur den Experten überlassen kann.

Zahlen der Lufthansa Industry Solutions ergeben, dass die Effektivität der Prozesse um 60% gesteigert werden können und die damit einhergehende Zeiterbsparnis eine Kostensenkung von bis zu 20 Prozent zur Folge hat. Des weiteren wird ein Tool zur Verfügung gestellt, bei dem man seine Anfragen pro Monat, die Minuten zur Bearbeitung einer Anfrage und die Kosten pro Stunde angibt. Mit diesen Informationen lassen sich die aktuellen Kosten in etwa bestimmen und gleichzeitig lassen sich die Kosten bei Nutzung von AIaaS ansehen. Somit kann sich der Kunde ein Bild davon machen, ob es sich lohnt, AIaaS einzuführen oder nicht. [vgl. Sol22]

6.2 Nachteile

Es gibt demzufolge viele Vorteile, die AIaaS mit sich bringt. Jedoch gibt es auch einige Nachteile, weshalb AIaaS nicht die perfekte Lösung darstellen könnte:

Es besteht immer die Gefahr, dass KI als Dienstleistung zur Entwicklung von autonomen Waffen oder anderen gefährlichen Technologien genutzt werden könnte. Auch wenn der Fall eher unwahrscheinlich ist, muss dieser beachtet werden. Der Nutzer muss nicht einmal unbedingt etwas böses im Sinn haben. Es reicht aus, wenn durch Fehler die künstliche Intelligenz intelligenter als der Mensch wird. Dadurch wird die KI unkontrollierbar und unvorhersehbar und es entstehen potenzielle Bedrohungen für die Welt und unsere Spezies. In der 42. Deutschen Konferenz über Künstliche Intelligenz ging es unter anderem um KI-Systeme für das Militär. KI-Systeme werden sowohl für autonome Waffensysteme, als auch zur Lagebeurteilung benutzt. Damit soll frühzeitig erkannt werden, ob beispielsweise ein feindlicher Raketenangriff droht. Somit kann ein menschlicher Entscheidungsträger schnell Vergeltungsschläge ausführen. Laut Experten könnten solche KI's zur Destabilisierung der internationalen Beziehungen führen und auch zu Konflikten, die sich verselbstständigen. Viele Länder möchten ein Verbot autonomer Waffen. Jedoch lehnen große Waffenhersteller wie Russland, die USA und Israel die Forderungen ab. [vgl. Deu19]

Der Einsatz von AIaaS könnte zu Arbeitsplatzverlusten und anderen wirtschaftlichen Störungen führen. Logischerweise folgt aus den Kosteneinsparungen bei Nutzung von KI's

ein geringeres Personal, das benötigt wird. Setzen sich Dienstleistungen und KI's in einem größerem Feld durch, können viele Arbeiter ihren Arbeitsplatz dadurch verlieren. Das kann wiederum zu wirtschaftlichen Störungen auf dem Markt führen.

Ein anderer Punkt ist, dass AIaaS das Risiko von Cyberangriffen und anderen bösartigen Aktivitäten erhöhen könnte. Durch die einfache Möglichkeit, an qualitativ hochwertige KI's zu kommen, können Menschen mit boshaften Absichten diese KI's ausnutzen, um Leute oder gar Firmen mit Cyberangriffen zu drohen oder gar durchzuführen.

Letzten Endes muss noch angesprochen werden, dass AIaaS genutzt werden kann, um Nutzer auszubeuten und zu manipulieren sowie ihre Privatsphäre zu verletzen. KI's sind sehr modern und viel gefragt. Dadurch kann ein potenzieller Nutzer sehr einfach manipuliert werden, sehr hohe Preise für eine kleine KI zu zahlen. Außerdem kann die Privatsphäre von Kunden sehr schnell angegriffen werden. Zum Lernen benötigt die KI meist sehr viele Daten. Oft werden dafür auch Daten von Kunden genommen, die normalerweise einen privaten Status haben. [vgl. InA20]

Ein aktueller Fall zeigt, dass das Einsetzen und zur Verfügung stellen von einem KI-Tool auch rechtliche Konsequenzen für den Anbieter haben kann, welcher die Schäden aber auch an seine Kunden weitergeben könnte. Eine aktuelle Sammelklage richtet sich gegen die Unternehmen GitHub, den Mutterkonzern Microsoft und das KI-Unternehmen OpenAI. Die von GitHub angeboten Programmierhilfe mit dem Namen GitHub Copilot soll laut Klage keine Quellen angeben, woher der generierte Code kommt, der zur Verfügung gestellt wird. Dadurch werden die Open-Source-Lizenzmodelle und die Rechte der Programmierer des Codes verletzt.

Copilot ist seit Juni 2022 zugänglich und bietet dem Nutzer Vorschläge, wie der momentane Code vervollständigt werden kann. Das KI-System greift dabei auf viele öffentliche Code-Repositorys in GitHub zurück. Dabei wird der Ersteller es Codes jedoch nicht um Erlaubnis gefragt. Die Klage beschreibt einen Verstoß gegen die Richtlinien des Digital Millennium Copyright Act (DMCA). Laut Klage verstößt die Software drei Mal gegen Abschnitt 1202 DMCA – fehlender Verweis aufs Urheberrecht, fehlende Namensnennung, sowie fehlender Lizenztext. Würde man davon ausgehen, dass jeder Copilot Nutzer einmal gegen den Abschnitt 1202 verstoßen hat, so würden sich 3.600.000 Verstöße durch GitHub ansammeln. Der Mindestschadenersatz läge dabei bei 2500 US-Dollar pro Einzelfall. Somit kommt man auf eine Gesamtsumme von 9 Milliarden Dollar. Dazu kommen noch weitere in der Klageschrift vorgeworfene Rechtsverstöße wie unlauterer Wettbewerb oder der Verstoß gegen das kalifornische Verbraucherschutzgesetz. [vgl. Wit22]

7 Fazit

7.1 Argumentationsfazit

Zusammenfassend lässt sich also sagen, dass es durchaus Zahlreiche Argumente gibt, die für den Einsatz von AIaaS im Unternehmensumfeld sprechen. Allerdings gibt es auch einige Gegenargumente die potenziell auftretende Probleme und Gefahren beschreiben. Hierbei gilt, dass jedes Unternehmen für sich erörtern und entscheiden muss wie mit diesen Punkten umgegangen wird und ob man die eventuellen Sicherheitslücken mit eigenen Ressourcen und Know-How schließen kann.

Man kann hier also deutlich sehen, dass, auch wenn das so von den verschiedenen Service Providern vermarktet und angepriesen wird, es nicht möglich, bzw. nicht empfehlenswert ist, einfach eine vorhandene künstliche Intelligenz in seine Systeme einzubinden. Man muss zahlreiche Modifikationen vornehmen. Allerdings: Hat man einmal initial die erforderlichen Anpassungen vorgenommen, hat man in der Regel weniger Wartungsaufwand als bei einer hauseigenen on-premise-Lösung, da das weitere Trainieren der KI und auch weitere Optimierungen bezüglich der Sicherheit oder Performance von den Providern durchgeführt wird.

7.2 Anbieterfazit

Wenn man sich für einen AIaaS-Dienst entschieden hat, geht es bei der Anbieterwahl primär darum, welcher von ihnen eine Lösung anbietet, die für mein Produkt geeignet ist. Viel Provider bieten eine Vielzahl an Diensten an, welche sich nur wenig unterscheiden. Neben den oben genannten Hauptakteuren am Markt kann man sich natürlich auch für einen kleineren Anbieter entscheiden. Das bietet den Vorteil, dass man eventuell einen größeren Verhandlungsspielraum hat, wenn es um Anfragen zu Zusatzfunktionen oder Anpassungen des KI-Modells geht. Bei großen Firmen wie Google oder Amazon hat man bei solchen Anfragen meist keinen Erfolg, da die Produkte für die breite Masse konzipiert sind und Individualanpassungen eher unüblich sind. Was jedoch ein großer Vorteil bei den „Big Playern“ ist, ist dass sie schon ein globales Netz an Rechenzentren zur Verfügung haben und somit eine hohe Ausfallsicherheit gewährleistet ist. Außerdem ist bei diesen Anbietern garantiert, dass die KI-Services kontinuierlich gewartet, erweitert und verbessert werden, da es dedizierte Teams dafür gibt.

7.3 Zukunftsaussicht

Die Zukunft von Everything-as-a-Service sieht sehr vielversprechend aus. Getrieben wird das Wachstum von XaaS durch die Kombination aus der hohen Nachfrage an Cloud Computing und der wachsenden Infrastruktur des globalen Internets mit einer hohen Bandbreite. Im Zuge diesen technologischen Fortschritts nutzen Unternehmen vor allem immer mehr Software-as-a-Service-Lösungen (SaaS) für ihren Betrieb. Es ist zu erwarten, dass sich in Zukunft immer mehr Unternehmen für dieses Modell entscheiden werden, das es ihnen ermöglicht, schnell und einfach auf die von ihnen benötigten Dienste zuzugreifen, ohne in teure Hardware und Software investieren zu müssen. Da Cloud Computing immer beliebter wird, werden auch die Kosten für XaaS-Lösungen immer attraktiver für Unternehmen aller Größenordnungen.

Der globale XaaS-Markt wurde 2021 von „Allied Market Research“ auf etwa 474,9 Milliarden US-Dollar bemessen. Voraussichtlich wird der Markt 2031 auf bis zu 2,63 Billionen US-Dollar anwachsen. Das wäre eine Durchschnittliche jährliche Wachstumsrate von 18,9 Prozent. Dabei gibt es vor allem starke Nachfragen bei SaaS, aber auch bei IaaS und PaaS. Hauptsächliche Treiber des Wachstums sind hierbei Regierungen, die Gesundheitsindustrie und die produzierende Industrie.

Der Artificial-Intelligence-as-a-Service-Markt für sich wird zwar im Vergleich zum allgemeinen XaaS-Markt voraussichtlich prozentual deutlich stärker anwachsen, wird jedoch trotzdem nur einen kleinen Teil des Gesamtmarktes ausmachen. Die Nachfrage an den klassischen Hauptkomponenten ist aktuell noch sehr hoch und die Anwendungsbereiche für AIaaS, vor allem in kleinen bis mittleren Unternehmen, noch nicht sehr ausgereift. Viele Unternehmen werden sich in Zukunft wohl zunächst auf die reine Digitalisierung bzw. der digitalen Abbildung von Geschäftsprozessen fokussieren, um dann danach Ansätze zu Prozessoptimierung mittels künstlicher Intelligenz zu verfolgen. [vgl. All22]

8 Literatur

- [Act22] ActiveMind AG. *Muster: Auftragsverarbeitungs-Vertrag nach DSGVO | active-Mind AG.* 2022. URL: <https://www.activemind.de/datenschutz/dokumente/av-vertrag/> (besucht am 06. 12. 2022).
- [All22] Allied Market Research. *Everything as a Service (XaaS) Market Size | Forecast - 2031.* 2022. URL: <https://www.alliedmarketresearch.com/everything-as-a-service-xaas-market-A17382> (besucht am 01. 12. 2022).
- [Ama22a] Amazon Web Services. *Amazon Transcribe Pricing – Amazon Web Services (AWS).* 2022. URL: <https://aws.amazon.com/de/transcribe/pricing/> (besucht am 06. 12. 2022).
- [Ama22b] Amazon Web Services. *Services mit künstlicher Intelligenz.* 2022. URL: <https://aws.amazon.com/de/machine-learning/ai-services/> (besucht am 06. 12. 2022).
- [Azu22] Microsoft Azure. *Azure KI-Plattform – KI-Dienst | Microsoft Azure.* 2022. URL: <https://azure.microsoft.com/de-de/solutions/ai/> (besucht am 06. 12. 2022).
- [Beu16] Patrick Beuth. „Google: Künstliche Intelligenz erfindet eigene Verschlüsselung“. In: *Die Zeit* (2016). URL: <https://www.zeit.de/digital/datenschutz/2016-10/google-kuenstliche-intelligenz-erfindet-eigene-verschluesselung> (besucht am 06. 12. 2022).
- [Com22] ComputerWeekly.de. *Was ist XaaS (Anything as a service)?* 2022. URL: <https://www.computerweekly.com/de/definition/Anything-as-a-Service-XaaS> (besucht am 24. 10. 2022).
- [Cuo22] Gennaro Cuofano. „AIaaS: Das neue Geschäftsmodell der künstlichen Intelligenz als Service“. In: *What is The FourWeekMBA* (2022). URL: <https://fourweekmba.com/de/aiaas/> (besucht am 28. 10. 2022).
- [Deu19] Deutschlandfunk.de. *Autonome Waffen - KI-Systeme im Militär.* 2019. URL: <https://www.deutschlandfunk.de/autonome-waffen-ki-systeme-im-militaer-100.html> (besucht am 06. 12. 2022).
- [Fol22] Folio3AI Blog. *Ai as a Service Guide - Advantages, Limitations, Types, Models, Platforms.* 2022. URL: <https://www.folio3.ai/blog/ai-as-a-service-aiaas/> (besucht am 01. 12. 2022).
- [Goo22a] Google Cloud. *Preise | Cloud Speech-to-Text | Google Cloud.* 2022. URL: <https://cloud.google.com/speech-to-text/pricing> (besucht am 06. 12. 2022).
- [Goo22b] Google Cloud. *Produkte und Dienste | Google Cloud.* 2022. URL: <https://cloud.google.com/products> (besucht am 06. 12. 2022).

- [HL16] Raoul Hentschel und Christian Leyh. „Cloud Computing: Gestern, heute, morgen“. In: *HMD Praxis der Wirtschaftsinformatik* 53.5 (2016), S. 563–579. ISSN: 1436-3011. doi: 10.1365/s40702-016-0254-5.
- [Hua23] Huawei Technologies Co. *Cloud Computing Technology*. 1st ed. 2023. Springer eBook Collection. Singapore: Springer Nature Singapore und Imprint Springer, 2023. ISBN: 978-981-19-3025-6. doi: 10.1007/978-981-19-3026-3.
- [IBM21] IBM Watson. *Watson Speech to Text - Pricing*. 2021. URL: <https://www.ibm.com/cloud/watson-speech-to-text/pricing> (besucht am 06. 12. 2022).
- [InA20] InApp. *The Future and Challenges of Artificial Intelligence-as-a-Service | AIaaS - InApp*. 2020. URL: <https://staging.inapp.com/blog/artificial-intelligence-as-a-service-its-future-and-challenges/> (besucht am 01. 12. 2022).
- [Kol20] Tobias Kollmann. *Handbuch Digitale Wirtschaft*. Wiesbaden: Springer Fachmedien Wiesbaden, 2020. ISBN: 978-3-658-17290-9. doi: 10.1007/978-3-658-17291-6.
- [Lin+21] Sebastian Lins u. a. „Artificial Intelligence as a Service“. In: *Business & Information Systems Engineering* 63.4 (2021), S. 441–456. ISSN: 2363-7005. doi: 10.1007/s12599-021-00708-w.
- [Mic22] Microsoft Azure. *Cognitive Speech Services Pricing | Microsoft Azure*. 2022. URL: <https://azure.microsoft.com/en-us/pricing/details/cognitive-services/speech-services/> (besucht am 06. 12. 2022).
- [Phi17] Johann Philipp. „Künstliche Intelligenz entwickelt Geheimsprache – Facebook zieht den Stecker [Update]“. In: *Giga* (2017). URL: <https://www.giga.de/extra/forschung-und-wissenschaft/news/kuenstliche-intelligenz-entwickelt-geheimsprache-facebook-zieht-den-stecker-update/> (besucht am 01. 12. 2022).
- [Pre21] PrecedenceResearch. *Artificial intelligence as a Service Market Size, Report 2030*. 2021. URL: <https://www.precedenceresearch.com/artificial-intelligence-as-a-service-market> (besucht am 06. 12. 2022).
- [Red22] RedHat. „Vergleich von IaaS, PaaS und SaaS“. In: (2022). URL: <https://www.redhat.com/de/topics/cloud-computing/iaas-vs-paas-vs-saas> (besucht am 01. 12. 2022).
- [Sol22] Lufthansa Industry Solutions. *AI as a Service: Geschäftsprozesse automatisieren | LHIND*. 2022. URL: <https://www.lufthansa-industry-solutions.com/de-de/loesungen-produkte/kuenstliche-intelligenz/ai-as-a-service-automatisierung-von-geschaeftsprozessen> (besucht am 06. 12. 2022).
- [Wat21] IBM Watson. *IBM Watson-Produkte*. 2021. URL: <https://www.ibm.com/de-de/watson/products-services> (besucht am 06. 12. 2022).

- [Wit22] Tilman Wittenhorst. „Microsoft, GitHub und OpenAI verklagt: KI-Programmierhilfe Copilot kopiert Code“. In: *heise online* (2022). URL: <https://www.heise.de/news/Microsoft-GitHub-und-OpenAI-verklagt-KI-Programmierhilfe-Copilot-kopiert-Code-7331566.html> (besucht am 01.12.2022).

Resilienz in Systemarchitekturen

Robin Epple,¹ Timo Vollert²

Abstract: Für Organisationen aller Art gehört es zu den obersten Prioritäten bestimmte interne und externe Prozesse am Laufen zu halten, um ihre Geschäftsziele zu erreichen. Diese Prozesse werden im Zuge der Digitalisierung zunehmend abhängiger von Softwaresystemen. Infolgedessen muss moderne Software entsprechenden Resilienzanforderungen gerecht werden. Die Schwierigkeit dabei ist, die Komplexität des Systems zu bewältigen, die aus der hohen Anzahl der Systemkomponenten und deren Vernetzungsgrad resultiert. In dieser Arbeit werden Prinzipien für verteilte Systeme behandelt, die es ermöglichen auch unter Ausfall von Teilsystemen die Funktionalität des Gesamtsystems zu erhalten. Anschließend werden konkrete Entwurfsmuster vorgestellt, die diese Prinzipien implementieren.

Keywords: Resilienz; Resilience Management Model; Normal Accident Theory; Verteilte Systeme; Entwurfsmuster

Einleitung

Hin und wieder gehen Schlagzeilen durch die Medien, die von Ausfällen in der Infrastruktur oder in der Industrie berichten. Je nach Ausmaß und Schwere des Vorfalls können solche Ausfälle katastrophale Folgen für die Organisation selbst und alle Abhängigen bedeuten. Sowohl der Ausfall interner Strukturen, als auch von Produkten und Dienstleistungen stellt für viele Organisationen ein zu priorisierendes Risiko dar. Das Forschungsbereich der Resilienz sucht Methoden, um Prozesse ausfallsicherer und verlässlicher zu machen. In Konsequenz der wachsenden Abhängigkeit von IT-Systemen übertragen sich diese Anforderungen auch auf die Software. Für die Entwicklung eines neuen Softwaresystems ist es heutzutage daher unabdingbar, entsprechende Resilienzanforderungen und -maßnahmen von Anfang an in den Entwurf einfließen zu lassen.

Die Resilienz eines Systems zu gewährleisten ist ein komplexes und breites Themengebiet, das in enger Beziehung zu vielen anderen Fachbereichen steht: von der Unternehmensführung bis hin zum Systemhaus.

In diesem Beitrag soll daher zunächst die Bedeutung softwaretechnischer Maßnahmen für Resilienz in Organisationen erarbeitet werden. Im Anschluss wird sich die Arbeit darauf konzentrieren, wie ein Softwaresystem resilenter gemacht werden kann.

¹ DHBW Stuttgart Campus Horb, Studiengang Informatik, Florianstraße 15, 72160 Horb am Neckar, Deutschland
i20010@hb.dhbw-stuttgart.de

² DHBW Stuttgart Campus Horb, Studiengang Informatik, Florianstraße 15, 72160 Horb am Neckar, Deutschland
i20033@hb.dhbw-stuttgart.de

Im Zeitalter der Digitalisierung sind zwei Bewegungen in der Softwareentwicklung zu beobachten: Softwaresysteme werden umfangreicher und komplexer, gleichzeitig ist ein Ausfall zunehmend schlechter verkraftbar [Char00, S. 3]. Das Forschungsgebiet, das sich mit dieser Problematik befasst heißt „Resilience Engineering“ [Jona20]. Bevor näher auf konkrete Maßnahmen in der Softwareentwicklung eingegangen wird, soll in den nächsten Abschnitten zunächst die Rolle der Softwareentwicklung in das breite Themenfeld der Resilienz eingeordnet werden.

Begriffsdefinition und Abgrenzung

Für diese Einordnung ist es wichtig, ein präzises Verständnis von dem Begriff Resilienz zu erhalten. Die Definitionen in der Literatur sind keineswegs einheitlich, vor allem aus dem Grund, dass sich das Themengebiet über viele verschiedene Fachbereiche erstreckt. Resilienz kann sich auf Software, Prozesse, Organisationen oder gar sozio-ökologische Systeme beziehen [HBO+18, S. 279]. Über all diese Bereiche hinweg ist allerdings folgender Konsens erkennbar: *Resilienz ist die Eigenschaft eines Systems, seine Funktionalität vor, während und nach einer Störung zu erhalten*. Was genau die Funktionalität des Systems ist, was unter „Erhalt der Funktionalität“ verstanden wird und was eine Störung bedeutet variiert je nach Kontext [HBO+18, S. 279].

Für IT Systeme kann diese Definition konkretisiert werden, um sie von ähnlichen Eigenschaften abzugrenzen: *Die Resilienz eines Softwaresystems beschreibt die Eigenschaft, seine Funktionalität unter partiell Defekt in Toleranzgrenzen aufrecht zu erhalten und sie nach Überwindung des Defekts zu regenerieren* [ZhLi10, S. 99]. Resilienz lässt sich damit vor allem in einer Hinsicht von ähnlichen Eigenschaften wie Robustheit und Zuverlässigkeit abgrenzen: Die Resilienz eines Systems zeigt sich, wenn es partiell beschädigt ist [ZhLi10, S. 99].

Zuverlässigkeit beschreibt die Fähigkeit eines Systems, seine Funktionalität unbeschädigt und unter vorgesehenen Bedingungen zu liefern [ZhLi10, S. 100]. Die Robustheit eines Systems ist gefragt, wenn es zu unvorhergesehenen Störungen kommt, das System aber noch vollständig in Takt ist [ZhLi10, S. 100]. Erst wenn eine Störung die Zuverlässigkeit und Robustheit eines Systems übersteigt und es zum Ausfall einer oder mehrerer Komponenten kommt ist die Resilienz gefragt.

Im Zusammenhang mit diesen Definitionen ist es wichtig, sich Gedanken über den Betrachtungsrahmen zu machen: Ein System ist unbeschädigt, wenn alle seine Systemkomponenten die jeweilige Aufgabe erfüllen, die sie im System übernehmen sollen. Je nach Granularität der Betrachtung verschieben sich allerdings die Definitionen, was als „Gesamtsystem“ und als „Komponente“ betrachtet wird. Ist der Rahmen beispielsweise ein einzelner Server, so stellt der Ausfall einer Festplatte den Ausfall einer Komponente dar. In diesem Betrachtungskontext wäre ein RAID System als Resilienzmaßnahme zu betrachten, die den Erhalt der Systemfunktionalität unter partiell Defekt gewährleistet. Wird als Rahmen allerdings

ein Softwaresystem gewählt, das mehrere Geräte in einem Netzwerk umfasst, so ist der Server als ganzes eine Komponente. Das RAID System erhöht damit die Zuverlässigkeit dieser einzelnen Komponente, löst aber im Gesamtsystem kein Resilienzproblem. In diesem größeren Betrachtungskontext müssten Resilienzmaßnahmen den Ausfall des gesamten Servers kompensieren.

Diese Arbeit wird zur Einordnung mit einer gesamten Organisation als größtem Rahmen beginnen, um sich anschließend schrittweise in Softwarearchitekturen zu vertiefen. Vorab wird der nächste Abschnitt Grundlagen behandeln, die auf jeden Rahmen anwendbar sind.

Normal Accident Theory

Bei näherer Betrachtung der Definitionen aus dem letzten Abschnitt liegt der Gedanke nahe, dass die Resilienz eines Systems überflüssig wird, wenn die Zuverlässigkeit und Robustheit der Komponenten nahe der Perfektion sind.

Dieser Gedanke ist keineswegs neu und wird heute der High Reliability Theory (HRT) zugeordnet. In der HRT wird davon ausgegangen, dass mittels durchdachtem Management von Systemen und Prozessen Ausfälle von vorne herein ausgeschlossen werden können. Typische Maßnahmen in der Umsetzung der HRT sind eine hohe Priorisierung der Ausfallsicherheit in den Führungsebenen, dezentrale Autorität für schnelle Reaktionen, eine hohe Redundanz in Ressourcen und Personal und kontinuierliches Lernen an eigenen und fremden Fehlern mittels Personaltraining. [ThGe22, S. 100]

Nachdem es im 20. Jahrhundert zunehmend zu Unfällen in scheinbar sicheren Systemen kam (beispielsweise im Kernkraftwerk von Three Mile Island) begann mit Charles Perrow eine Bewegung des Umdenkens [Perr04, S. 9]. Perrow begründete die Idee der „Normal Accidents“. Die Normal Accident Theory (NAT) hebt die Möglichkeit hervor, dass selbst bei scheinbar zur Perfektion gemanagten Systemen ein ungünstiges Zusammenspiel mehrerer Störungen zum Ausfall eines Teilsystemen führen können [Char00, S. 4].

Hervorzuheben ist an dieser Stelle, dass die NAT keineswegs zu einer Vernachlässigung von Zuverlässigkeit und Robustheit aufruft. Fehler in einzelnen Komponenten werden von Perrow nicht als „Normal Accidents“ klassifiziert. Die Vermeidung von Komponentenfehlern und damit ein zuverlässiges System sind vielmehr die Grundlage, auf der die NAT aufbaut [Perr04]. Die NAT weißt auf die Imperfektion des Menschen hin: Es darf niemals davon ausgegangen werden, dass ein System fehlerfrei ist, nur weil bisher keine Fehler entdeckt wurden. Es sollte immer die Möglichkeit in Betracht gezogen werden, dass Umstände auftreten, die vorher nicht berücksichtigt wurden, und dass es bei ungünstigen Kombinationen zum Ausfall von (Teil-)Systemen kommen kann.

Die NAT verlangt also, aufbauend auf einem zuverlässigen System den Fehlerfall zu erwarten. Sollte eine Komponente ausfallen, muss das System reagieren, um diesen Ausfall

zu kompensieren. Ein Komponentenfehler soll nicht zu kaskadierende Abhängigkeitsfehlern führen.

Die Maßnahmen zur Resilienzsteigerung lassen sich aus den Ursachen der „Normal Accidents“ ableiten. Die Arbeiten von Perrow führen diese Fehler auf zwei Dimensionen zurück: Die Komplexität der Interaktionen in einem System und die Enge der Kopplung zwischen Systemkomponenten [Perr04].

Die Komplexität eines Systems wird definiert durch die Anzahl, Anordnung und Ersichtlichkeit der Interaktionen zwischen seinen Komponenten. Sie spielt vor allem aufgrund der menschlichen Imperfektion eine Rolle in der NAT. Je linearer die Interaktionen zwischen den Komponenten sind, desto übersichtlicher wird das System für menschliche Betrachter [Char00, S. 72]. In einem linearen Fluss ist es einfach, die Auswirkungen von lokalen Problemen auf benachbarte Komponenten zu überblicken und Gegenmaßnahmen einzuplanen. Je komplexer und undurchsichtiger das Netzwerk aus Interaktionen ist, desto wahrscheinlicher wird es, dass bestimmte Umstände und Aspekte übersehen werden, die Katastrophenpotential bergen.

Die Eigenschaft der Kopplung beschreibt, wie starr die Abhängigkeiten zwischen den Komponenten sind. Je enger die Kopplung desto weniger Handlungsspielraum hat eine Komponente. Fällt ein Interaktionspartner aus, dann führt eine starre Abhängigkeit unausweichlich zu Folgefehlern. Es kommt zu sich ausbreitenden Abhängigkeitsfehlern, die nur mit einer zentralen Maßnahme gestoppt und behoben werden können. Lose Kopplungen hingegen erlauben es einem System, lokal alternative Prozessabläufe einzuleiten um die verlorene Abhängigkeit zu kompensieren. Man spricht von einer dezentralen Fehlerbehandlung. [ThGe22, S. 101]

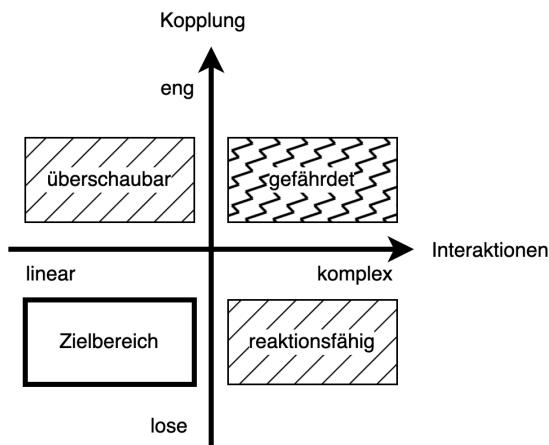


Abb. 1: Interaktionsdimensionen NAT

Bildet man aus diesen beiden Eigenschaften ein Koordinatensystem, dann lassen sich die

Systeme in vier Quadranten einteilen (siehe Abbildung 1) [Char00, S. 97]. Ein resilientes System strebt möglichst lineare, übersichtliche Interaktionen bei gleichzeitig loser Kopplung an [ThGe22, S. 101]. Lässt sich eine enge Kopplung nicht vermeiden, sollte diese durch möglichst einfache Interaktionen ausgeglichen werden und umgekehrt. Wird das System in den Quadranten für enge Kopplung und komplexe Interaktionen eingeordnet, ist das ein Warnsignal.

Im Kontext der resilienten Softwareentwicklung können zwei Lehren aus der NAT gezogen werden: Um Abhängigkeitsfehler beim Ausfall einer Komponente zu vermeiden benötigen die Prozessabläufe Ausweichmöglichkeiten und lokalen Spielraum. Lose Kopplungen erfordern redundante Funktionalität im System.

Die zweite Lehre betrifft die Komplexität der Interaktionen. Natürlich kann innerhalb des Softwaresystems auf möglichst übersichtliche Prozessabläufe geachtet werden. Entscheidend ist aber, dass Softwaresysteme letztendlich nur ein Teil von größeren Systemen sind. Um die Komplexität nachhaltig zu reduzieren, müssen ganze Geschäftsprozesse überarbeitet werden, nicht nur die Software.

Resilience Management Model

Dieser Gedanke führt unweigerlich zum Resilienzmanagement. Resilienzmanagement ist dafür verantwortlich, die Kerngeschäfte einer Organisation abzusichern und zentrale Unternehmensaktivitäten unter allen Umständen am Laufen zu halten. Es betrifft also alle Komponenten des Systems innerhalb einer Organisation. Dieses Kapitel soll einen Überblick über das breite Themengebiet geben, um anschließend einzuordnen welche Rolle Software in einem resilienten Prozess übernehmen muss, und welche Anforderungen an das System daraus resultieren.

Das Resilience Management Model (RMM) von CERT will einen Leitfaden für Unternehmen bieten, mit dem priorisierte Leistungen aufrecht erhalten werden können. Die Informationen im nachfolgenden Abschnitt referenzieren die Originalpublikation [Rich10] sofern nicht anders angegeben. Sinngemäß übersetzt lautet die Definition für Resilienzmanagement in dem Standard wie folgt:

Resilienzmanagement umfasst Prozesse und Praktiken einer Organisation, die Strategien entwerfen, entwickeln, implementieren und steuern, die dem Schutz und Erhalt von Dienstleistungen, Wirtschaftsgütern oder anderen Prozessen mit hohem Wert dienen. [Rich10, S. 19]

Die Definition macht bereits deutlich, dass das Management einen anderen Blickwinkel auf das Thema Resilienz hat. Ressourcen sind nur ein Teil der Zielmenge von Schutzmaßnahmen. Vorrangig ist zunächst, schützenswerte Wirtschaftsgüter zu identifizieren und notwendige

Maßnahmen einzuleiten. Abbildung 2 veranschaulicht, wie ein Unternehmen zu Strategien und Maßnahmen finden kann, die die Unternehmensresilienz steigern.

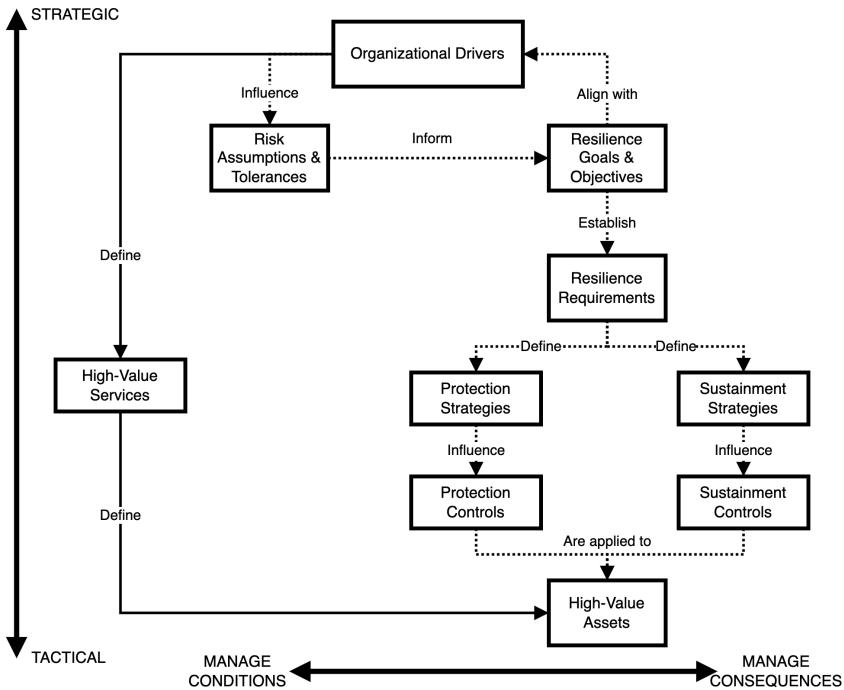


Abb. 2: Ableiten von Resilienzmaßnahmen aus Unternehmenszielen. Nachbildung von [Rich10, S. 26, Figure 9].

An oberster Stufe der Grafik stehen Organisationstreiber. Das sind Strukturen, die ein förderndes Umfeld für die Organisation entwickeln oder erhalten wollen [Nati].

Davon ausgehend verfolgt der linke Pfad in der Grafik den Prozess, einzelne schützenswerte Ressourcen aus übergeordneten Unternehmenszielen abzuleiten. In diesem Prozess definiert das RMM drei Ebenen: Dienstleistungen, Prozesse und Ressourcen.

Dienstleistungen sind das eigentliche Objekt, das es zu schützen gilt. Darunter kann jede Tätigkeit einer Organisation verstanden werden: Von internen Managementstrukturen über Produktionstätigkeiten bis hin zu externen Dienstleistungen bei Kunden. Es gilt die Bedeutung der Dienstleistung für den Erhalt der Tätigkeit des Unternehmens zu bewerten. Mit diesen Informationen kann eine Priorisierung der Leistungen vorgenommen werden.

Eine Dienstleistung besteht aus einem oder mehreren Prozessen. Prozesse sind beliebige Geschäftsprozesse. Sie bestehen aus Aktivitäten, die manuelle Aufgaben, vollautomatisierte Prozessschritte oder interne und externe Kommunikation abbilden können.

Prozesse sind der Hauptfokus von RMM, da sie konkrete Arbeitsabläufe darstellen, die analysiert und überarbeitet werden können. In der Übersichtsgrafik 2 sind sie allerdings nicht abgebildet, da Prozesse letztlich „nur“ konkrete Umsetzungen der Dienstleistungen sind. Resilienzmaßnahmen schützen Prozesse, um Dienstleistungen zu erhalten.

Im Ableitungsprozess stellen Prozesse dennoch einen wichtigen Schritt dar, da sie Dienstleistungen mit ihren Abhängigkeiten verbinden. Abhängigkeiten sind beliebige Ressourcen der Organisation. Über die Prozesse können Ressourcen identifiziert werden, die für den Erhalt der Unternehmensaktivität besondere Bedeutung besitzen.

Ressourcen werden im Standard in vier übergeordnete Kategorien eingeteilt:

1. Personen, die beteiligt oder verantwortlich sind.
2. Informationen, die der Prozess benötigt oder generiert.
3. Technologien, die für die Durchführung benötigt werden.
4. Einrichtungen oder Gebäude, von denen der Prozess abhängt.

Softwaresysteme sind der Kategorie „Technologien“ zuzuordnen. Abbildung 3 bildet den Zusammenhang von Dienstleistungen und Ressourcen ab. Wichtige Ressourcen können auch Teil mehrerer priorisierter Dienstleistungen oder Prozesse sein.

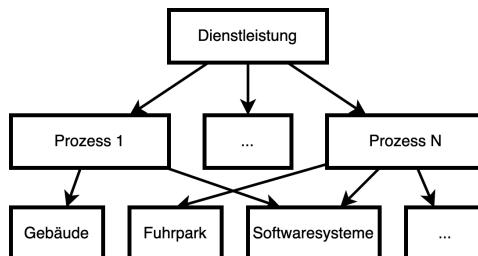


Abb. 3: Schützenswerte Ressourcen von priorisierten Dienstleistungen ableiten.

In Abbildung 2 führt noch ein zweiter Pfad von den Organisationstreibern zu den priorisierten Ressourcen. Er spiegelt den Prozess wieder, konkrete Schutzmaßnahmen zu entwickeln.

In diesem zweiten Pfad wird deutlich, dass Resilienzmanagement alle Ebenen eines Unternehmens betrifft. Er hat seinen Ursprung in der strategischen Ebene der Unternehmensführung und zieht sich in der Grafik bis in die taktische Ebene. Die Umsetzung ist in dieser Grafik nicht abgebildet, ist aber ebenso Teil der Kette.

Auf strategischer Ebene findet sich auf der linken Seite der Punkt „Risikoannahmen und Toleranzen“, der die Informationsgrundlage für „Resilienzziele“ bildet. Hier wird der enge Zusammenhang von Resilienzmanagement und Risikomanagement ersichtlich. Resilienzmanagement ist eine Methode, um identifizierten Risiken zu begegnen.

Sind die Resilienzanforderungen einmal definiert, teilt das RMM den Pfad in die Entwicklung schützender und erhaltender Strategien.

Schützende Strategien sollen erreichen, dass Ressourcen von Grund auf weniger Störungen ausgesetzt sind. Für Resilienz rund um Softwaresysteme bedeutet das, dass die Software nicht alle Störungen selbst abfangen muss. Es kann auch Mechanismen in der Betriebsumgebung geben, die viele Gefahren abblocken, bevor sie das System erreichen.

Erhaltende Strategien sollen die Funktionalität der Ressource in Toleranzgrenzen aufrecht-erhalten, während sie einer Störung ausgesetzt ist. Diese Definition macht zum einen deutlich, dass auch im Störungsfall nicht zwingend die Ressource, also das Softwaresystem selbstständig reagieren können muss. Wie in Abschnitt „Theoretische Prinzipien“ ersichtlich werden wird, sind einige Resilienzanforderungen an ein Softwaresystem sehr komplex. Es kann eine valide Entscheidung sein, die Komplexität der Software zu reduzieren, indem andere begleitende Systeme die entsprechenden Eigenschaften sicherstellen.

Zu den Strategien werden auf taktischer Ebene konkrete Maßnahmen definiert, die auf die Ressourcen angewendet werden. Insgesamt macht dieser Pfad deutlich, dass beim Resilienzmanagement Softwaresysteme nicht isoliert betrachtet, sondern in ihren Kontext der Geschäftsprozesse eingeordnet und als Gesamtsystem resilenter gemacht werden. Neben der Software spielen Personal, Management, Umgebungsbedingungen und viele weitere Faktoren eine Rolle. Im Folgenden wird sich diese Arbeit allerdings konkret Software und dessen Entwicklungsprozess konzentrieren.

Softwareentwicklung

Der Standard RMM identifiziert insgesamt sechsundzwanzig Prozessräume in vier Kategorien. Resilienz in Software und Softwaresystemen wird einem davon zugeordnet: „Resilient Technical Solution Engineering“ in der Kategorie „Engineering“ [Rich10, S. 31].

RMM selbst gibt wenig konkrete Maßnahmen vor, die ein Unternehmen umsetzen kann oder muss. Stattdessen gibt es Anreize, was die Führungsebene beim Resilienzmanagement beachten muss.

Ein hohes Gewicht bekommt die Ermahnung, dass effektive Resilienzmaßnahmen bereits von Beginn an in den Entwicklungsprozess neuer Software einfließen müssen [Rich10, S. 178]. Um Resilienz effektiv im Unternehmen zu etablieren wird empfohlen, nicht jedes System einzeln zu behandeln, sondern Richtlinien für die Softwareentwicklung zu definieren, die bei jedem Projekt eingehalten werden [Rich10, S. 179].

RMM definiert nur vage, was in diesen Richtlinien enthalten sein sollte, es wird aber auf verschiedene Quellen mit guter Reputation verwiesen, die als Orientierung dienen können. Eine davon ist der Microsoft Security Development Lifecycle (SDL), der nachfolgend näher beleuchtet werden soll [Rich10, S. 180].

Dieser Standard macht deutlich, dass Resilienzmanagement ebenfalls eng mit Sicherheitsmanagement zusammen hängt.

SDL definiert zwölf Praktiken, die bei der Softwareentwicklung beachtet werden sollten [Micr]. Nachfolgend werden diese Praktiken in vier Faktoren klassifiziert: beteiligte Personen, Anforderungen, Umfeld und Tests.

Practice #1: beteiligte Personen Die Basis für wirksame Resilienzmaßnahmen ist, dass sie von betroffenen Mitarbeitern umgesetzt werden. Der erste Faktor ist daher Personaltraining in allen beteiligten Bereichen, von der Entwicklung bis hin zum Produktmanagement. Gleichzeitig steigert das Training das Bewusstsein für Gefahren und die Notwendigkeit von Absicherungen.

Practice #2-6: Anforderungen Der zweite Faktor ist die Analyse der Anforderungen an das System. Es müssen Gefahren ermittelt, eventuell modelliert und anschließend in Anforderungen umgesetzt werden. Um die Erfüllung der Anforderungen sicherzustellen ist es sinnvoll, Metriken zu definieren, die bestanden werden müssen. Für die konkrete Umsetzung ist es häufig sinnvoll, globale Designentscheidungen und Implementierungspraktiken zu definieren, beispielsweise die Entscheidung für bestimmte Sicherheitsstandards.

Practice #7-8: Umfeld Die dritte Kategorie von Praktiken ruft dazu auf, verwendete Werkzeuge und Drittherstellerkomponenten unter die Lupe zu nehmen, um festzustellen, ob sie den Ansprüchen genügen. Gleichzeitig sollte ein Inventar über verwendete Drittherstellersoftware gehalten und Maßnahmen definiert werden, um einschreiten zu können falls ein Sicherheitsproblem bekannt wird.

Practice #9-12: Tests Der letzte Faktor ist die Sicherstellung, dass die Software den Anforderungen entspricht. Hier können verschiedene Testmethoden zum Einsatz kommen, beispielsweise statische und dynamische Sicherheitstests und Penetrationstests. Zusätzlich ist es eine wertvolle Maßnahme, einen Standardprozess für die Reaktion auf Probleme im Einsatz zu definieren.

Angestrebte Eigenschaften

Um resilient zu sein muss ein Softwaresystem gewisse Eigenschaften erfüllen. Diese definiert [ThGe22] als die Eigenschaften Kapazität, Flexibilität, Toleranz und Kohäsion.

Kapazität Die Kapazität eines Softwaresystems beschreibt die Fähigkeit Bedrohungen zu widerstehen. Um das zu erreichen soll das System gewisse Belastungen absorbieren können. Dazu gehört auch, dass sowohl physische, als auch funktionale Redundanzen existieren, auf welche ausgewichen werden kann. Im Rahmen der Resilienz kann die Kapazität eines Systems überschritten werden. In dem Fall verlässt sich das System darauf, dass sich das System anhand der drei weiteren Eigenschaften wieder erholt.

Flexibilität In Angesicht einer Bedrohung ermöglicht die Flexibilität dem System sich zu restrukturieren. So kann dieses bei Bedarf die eigene Architektur anpassen um beispielsweise funktionale Redundanzen zu verwenden. Für diese kann es nötig sein, dass der Programmablauf angepasst werden muss. Dafür ist es sehr hilfreich, wenn das System eine lose Kopplung realisiert. So können die einzelnen Komponenten mit weniger Aufwand restrukturiert werden.

Toleranz Wird die Kapazität eines Systems überschritten, so gewährt die Toleranz eine geordnete Reduktion der Funktionen. Wie bereits bei der Flexibilität hilft hier ein lose gekoppeltes System. Dieses ermöglicht die ausgefallenen Komponenten einfacher aus dem Programmablauf herauszunehmen, so dass sich die Ausfälle auf einen Systemteilausfall beschränken. Das System soll zudem in der Lage sein sich von der Reduktion zu erholen, indem die Funktionen wiederhergestellt oder ersetzt werden.

Kohäsion Durch die Kohäsion eines Systems wird der totale Systemzerfall unterbunden. Das System bleibt vor, während und nach einer Bedrohung funktionstüchtig. Dafür muss eine Basiskommunikation der Systemknoten aufrecht erhalten werden. So kann ermöglicht werden, dass sich das System eigenständig erholt.

Theoretische Prinzipien

In [ZhLi10, S. 104] werden fünf theoretische Prinzipien aufgelistet, nach welchen ein resilientes System modelliert werden sollte, welches diese Eigenschaften besitzt. Diese Prinzipien dienen als Richtlinien für ein theoretisch ideal resilientes System. Eine vollständige Umsetzung dieser Prinzipien ist in vielen Systemen unrealistisch durch eine reine Softwarelösung zu realisieren.

Darum werden in der Realität viele der Funktionen durch menschliche Akteure ausgeführt. Aber auch mit menschlichen Akteuren dienen sie als Richtlinien, welche Elemente ein System besitzen muss, um resilient zu sein.

Die Prinzipien beschreiben ein Zusammenspiel von Hard- und Software in einem System, welches modular und anpassbar ist. In Abbild 4 wird das Zusammenspiel dieser Prinzipien abgebildet.

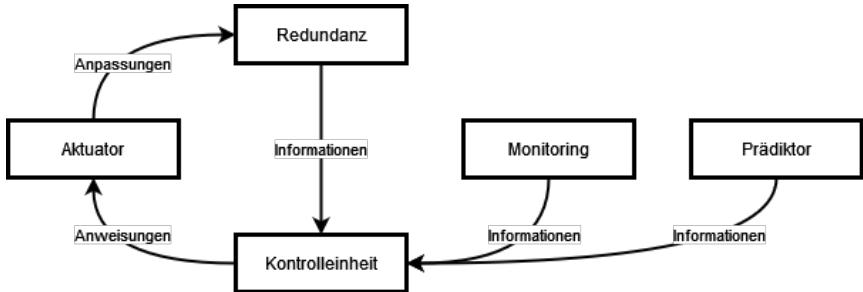


Abb. 4: Zusammenspiel der fünf Prinzipien. Adaption von [ZhLi10, Figure 5, S. 109].

Redundanz Ein resilientes System sollte mit einem bestimmten Grad an Redundanz entworfen werden. Dabei sollten physische und funktionale Redundanz implementiert werden. Diese erhöhen die Kapazität des Systems.

Die Redundanz eines Systems misst sich darin, dass Systemfunktionen jeweils von mehreren Komponenten ausgeführt werden können. Diese unterteilt sich in die funktionale und die physische Redundanz. Die funktionale Redundanz bedeutet, dass einzelne Funktionen mehrfach umplementiert werden. Dabei können die Implementierungen voneinander abweichen, lediglich die zu erfüllende Funktion muss die gleiche sein. Bei der physischen Redundanz geht es darum, dass das System auf mehrere Maschinen verteilt ist. So kann das System auch bei dem Ausfall der Hardware die Funktion aufrecht erhalten.

Monitoring Ein resilientes System sollte ein Monitoring besitzen, welches verantwortlich ist für die zeitliche und räumliche Überwachung der Systemfunktionen und Arbeitsleistung, Überwachung der Auslastung der Systemkapazitäten und Überwachung der Systemanforderungen.

Durch das Monitoring wird das gesamte System überwacht. Es wird die Auslastung des Gesamtsystems und einzelner Komponenten gemessen. Die ermittelten Informationen werden an die Kontrolleinheit übermittelt, damit diese auf dessen Basis Entscheidungen treffen kann.

Prädiktor Ein resilientes System sollte einen Prädiktor besitzen, der verantwortlich ist für das Vorhersagen von potentiellen Gefahren für das System und das Analysieren potentieller Schwachstellen des Systems.

Mit Hilfe des Prädiktors werden potentielle interne und externe Bedrohungen auf die Stabilität des Systems gleichermaßen ermittelt. Informationen über diese möglichen Bedrohungen können der Kontrolleinheit dabei helfen Präventivmaßnahmen für das System festzulegen.

Kontrolleinheit Ein resilientes System sollte eine Kontrolleinheit besitzen, welche für Redundanzmanagement und Funktionen lernen verantwortlich ist.

Die Kontrolleinheit plant Anpassungen um das System an die Belastung anzupassen. Sie legt fest, wie das System bei dem Ausfall von Komponenten oder Funktionen angepasst werden muss, dass die Funktionen von einer Redundanten Einheit übernommen werden. Dafür erhält die Kontrolleinheit Informationen von dem Monitoring und dem Prädiktor. In die Entscheidungsfindung wird zudem die Aufstellung aus der Redundanz berücksichtigt. Durch die Kontrolleinheit kann auch entschieden werden, dass Komponenten trainiert werden um neue Funktionen zu übernehmen.

Aktuator Ein resilientes System sollte einen Aktuator besitzen, der verantwortlich ist für die Implementierung von Änderungen an dem System in sowohl der kognitiven, als auch der physischen Domäne und der Implementierung von Trainings von einer Komponente, oder einem Subsystem zur Ausführung von neuen Funktionen.

Der Aktuator setzt somit die Entscheidungen der Kontrolleinheit um. So kann er zum einen vorhandene Redundanzen nutzen um ausgefallene Funktionen oder Komponenten zu ersetzen, indem er diese in dem Systemablauf einbindet. Zum anderen kann der Aktuator durch Training den Funktionsumfang von Komponenten erweitern.

Ein lose gekoppeltes System ist für alle genannten Prinzipien eine Grundvoraussetzung. Eine dynamische Redundanz über mehrere physische Maschinen hinweg ist anders schwer realisierbar. Zudem fordert es die Reaktionsmöglichkeit des Systems, bei Ausfällen einzelner Komponenten. Diese Funktion wird in der Normal Accident Theory gefordert. Sie ermöglicht es das System dynamisch anzupassen, wenn einzelne Komponenten ausgefallen sind. So kann die ausgefallene Funktion der entsprechenden Komponente weiter durchgeführt werden.

Verteilte Systeme

[Tv08, S. 19] definiert ein verteiltes System als *eine Ansammlung unabhängiger Computer, die den Benutzern wie ein einzelnes kohärentes System erscheinen*. Dabei kann es sich bei *Computern* um verschiedenste physikalische, aber auch virtuelle Maschinen handeln. Um diese Abstraktion zu verdeutlichen werden sie deshalb in der Folge als *Komponenten* bezeichnet.

Verteilte Systeme werden in dieser Arbeit als die Basis für die Implementierung von Resilienz in Softwaresystemen betrachtet. Ein verteiltes System ist bereits grundsätzlich erweiterbar und skalierbar durch die unabhängigen Komponenten [Tv08, S. 19]. Diese Eigenschaft wird benötigt um das System dynamisch anzupassen, wie es als Systemeigenschaft der Flexibilität gefordert wird. Darum bietet es sich an den Betrachtungsrahmen für die Entwurfsmuster auf verteilte Systeme zu beschränken. Die genannten Entwurfsmuster lassen sich bei Bedarf auf monolithische Systeme runterbrechen, allerdings müssten dafür Anpassungen gemacht werden, welche nicht in dieser Arbeit behandelt werden.

Entwurfsmuster

Die Kommunikation zwischen den einzelnen Komponenten eines verteilten Systems ist, wo Maßnahmen für die Resilienz des Gesamtsystems implementiert werden können. Eigentlich gelten die vielen Komponenten in der Hinsicht als ein Schwachpunkt von verteilten Systemen, da häufig Fehler in einer dieser Komponenten weitreichende Auswirkungen auf abhängige Komponenten haben, wenn das System keine Resilienz aufweist. Um diesen Schwachpunkt zu der großen Stärke der Architektur umzuwandeln beschreibt [Wo16] die folgenden Entwurfsmuster.

Timeout Werden synchrone Anfragen gestellt, so muss auf die Antwort gewartet werden, bevor die weitere Bearbeitung erfolgen kann. Eine lange Wartezeit auf die Antwort ist dabei ein Indiz, dass die angefragte Komponente überlastet, oder fehlerhaft ist. Je länger also auf eine Antwort gewartet wird, desto wahrscheinlicher, dass diese keinen Mehrwert bringt. Deshalb können Anfragen mit einer zeitlichen Begrenzung gestellt werden. Wird diese überschritten, so läuft sie in einen Timeout. Es wird in der Folge angenommen, dass die Komponente überlastet, ausgefallen oder anderweitig fehlerhaft ist.

Die Implementierung einesTimeouts verhindert die übermäßige Blockierung von Ressourcen. Um die Anfrage zu wiederholen bietet es sich an eine exponentiell steigende Zeit vor dem Neuversuch zu warten. So sinkt die Wahrscheinlichkeit die angefragte Ressource überlastet zu halten. Zudem sollte nicht endlos lange probiert werden die Anfrage zu wiederholen, ansonsten sind Endlosschleifen möglich.

Eine Umsetzung des Timeout benötigt von den theoretischen Prinzipien das Monitoring, die Kontrolleinheit und den Aktuator. Die Kontrolleinheit verwaltet den eigentlichen Timer. Kommt innerhalb des Timers keine Antwort, die das Monitoring an die Kontrolleinheit weitergeben kann, so wird die Anfrage abgebrochen. Dafür gibt die Kontrolleinheit dem Aktuator die Anweisung für den Neuversuch nach der definierten Zeit.

Bulkhead Bulkheads (Schotten) sind Kammern, in die ein Schiffsrumpf aufgeteilt ist. Sie stellen sicher, dass potenzieller Wassereintritt nicht zum Sinken des gesamten Schifffes führt.

Das Wasser kann maximal die Kammer mit dem Schaden in der Schiffswand fluten, die anderen Kammern bleiben dabei intakt. Auf die Softwareentwicklung übertragen bedeuten Bulkheads, dass ein Gesamtsystem in mehrere Komponenten unterteilt wird. Ausfälle und Probleme in einzelnen Komponenten bleiben dabei isoliert in der entsprechenden Komponente und dürfen keine andere Komponente beeinflussen.

Idealerweise wird bei Bulkheads durch redundante Komponenten ermöglicht die weggefahrene Funktion zu ersetzen. Aber auch ohne diese Redundanz verhindern Bulkheads häufig den vollständigen Absturz des Gesamtsystems. Viele kleineren Fehler können so frühzeitig abgefangen und behandelt werden. Dem Gesamtsystem wird dadurch die Möglichkeit gegeben sich eigenständig von diesem zu erholen. Das Bulkhead Entwurfsmuster realisiert somit die Systemeigenschaft Kohäsion. Zudem bietet es die Möglichkeit Redundanzen einzubauen.

Steady State Das Steady State Entwurfsmuster besagt, dass ein Programm einen Status haben sollte, auf den es bei einem Ausfall zurückkehren kann. Das ermöglicht es den Schaden eines Ausfalls zu reduzieren. So entstehen keine Folgeschäden durch die Annahme, dass sich der Status der Komponente geändert hat, obwohl diese in der Zwischenzeit ausgefallen war. In der Extremform kann auch vollständig auf einen Status verzichtet werden. Eine Komponente ohne Speicherung eines Status bezeichnet man als Stateless. Wie bereits das Bulkhead Entwurfsmuster realisiert Steady State die Systemeigenschaft der Kohäsion.

Fail Fast Fail Fast besagt, dass eine Komponente schnellstmöglich die Anfrage beantworten soll, wenn Fehler auftreten. So kann verhindert werden, dass erst durch ein Timeout der Fehler in der aufrufenden Komponente auftritt. Diese Herangehensweise reduziert den benötigten Ressourcenverbrauch, da schneller mit der Fehlerbehandlung begonnen werden kann. Außerdem wird das Blockieren der entsprechenden Ressourcen schneller aufgehoben. So kann die Gesamtsystemauslastung reduziert.

Circuit Breaker Circuit Breakers sind ein Entwurfsmuster, welches sein Vorbild in den Sicherungen in Stromkreisen findet. Dieser unterbricht den Stromkreis, sollte dieser beispielsweise durch einen Kurzschluss überlastet werden. In einem verteilten System ist ein Circuit Breaker ein zwischengeschaltetes Element, welche im Regelfall die Anfragen direkt an das zu schützende Element weiterleitet. Wird in diesem Element allerdings ein Fehler, oder eine Überlastung festgestellt, so werden die Anfragen abgefangen und mit dem entsprechenden Fehler beantwortet.

Ein Circuit Breaker ist somit auch eine Implementierung des Fail Fast Entwurfsmusters. Das zu schützende Element wird durch den Circuit Breaker entlastet und hat die Möglichkeit den aufgetretenen Fehler zu behandeln, oder die angestauten Anfragen zu bearbeiten. Dadurch können Folgeschäden durch den Fehler, oder die Überlastung vermieden werden. Der

Normalzustand des Circuit Breakers wird nach einer gewissen Zeit wieder hergestellt. Alternativ kann der Circuit Breaker auch Wellness Checks an das zu schützende Element schicken.

Wie bereits der Timeout werden in dem Circuit Breaker die theoretischen Prinzipien des Monitoring, der Kontrolleinheit und des Aktuators realisiert. Über das Monitoring wird festgestellt, dass die angefragte Komponente ausgefallen ist. Die Kontrolleinheit reagiert darauf, indem sie den Aktuator anweist die Verbindung zu kappen. Ebenso arbeiten die Prinzipien zusammen um die ursprüngliche Verbindung wiederherzustellen, sobald sich die entsprechende Komponente erholt hat.

An Stelle der Rückgabe eines Fehlers kann der Circuit Breaker die Anfragen auch mit Hilfe von Cache-Daten beantworten und die resultierenden Änderungen zwischenspeichern. Somit können kleine Anfragen weiterhin bearbeitet werden und die Auswirkungen des Ausfalls werden reduziert. Ist das der Fall, so dient der Circuit Breaker auch als Redundanz für die angefragte Funktion.

Handshaking Handshaking definiert die Einleitung der Kommunikation zwischen zwei Komponenten. Einleitung der Kommunikation bietet Möglichkeit bei Überlastungen direkt abzubrechen. Die Anwendung ist zwar prinzipiell erreichbar, aber das Stellen einer Anfrage ist nicht sinnvoll. Im Sinne des Fail Fast Entwurfsmusters wird die Kommunikation deshalb direkt abgelehnt. So kann sich die angefragte Komponente erholen. Gleichzeitig kann die anfragende Komponente schnell den aufkommenden Fehler behandeln. Wie bereits der Circuit Breaker implementiert Handshaking das Fail Fast Entwurfsmuster.

Eine Abwandlung des Handshaking wird in [Priy21] als Backpressure Entwurfsmuster beschrieben. Hier wird keine Verbindung initialisiert, aber die angefragte Komponente gibt ebenfalls direkt Feedback, sollte sie Überlastet sein.

Das Handshaking realisiert neben der Kontrolleinheit auch den Prädiktor. Beide sind in der angefragten Komponente dafür zuständig vorherzusagen, ob die Anfrage beantwortet werden kann, oder ob die Verbindung abgebrochen werden soll.

Entkopplung durch Middleware Asynchrone Aufrufe haben den Vorteil, dass nicht auf eine Antwort gewartet werden muss, sondern das Programm direkt weiterrechnen kann. Allerdings können auch bei asynchronen Aufrufen Fehler auftreten, welche entsprechend behandelt werden müssen. Um die Aufrufe dennoch asynchron gestalten zu können kann die Fehlerbehandlung dieser Aufrufe einer Middleware überlassen werden, wodurch weniger Ressourcen blockiert werden. Eine zentrale Fehlerverwaltung hat zudem den Vorteil, dass Systemweit einheitlich auf die Fehler reagiert wird

Die Middleware ist dabei die Implementierung des Monitoring, indem sie das System auf Fehler überwacht. Gleichzeitig dient sie aber auch als Kontrolleinheit und Aktuator, indem sie die Fehlerbehandlung eigenständig plant und durchführt.

Batch to Stream Häufig werden in Systemen regelmäßige Jobs ausgeführt um Daten zu bereinigen, alte Daten zu löschen, oder ähnliches. Diese bedeuten eine erwartete Belastung für den Server und werden deshalb häufig bereits zu Zeiten mit niedriger Systemlast gestartet. Sollte die Systemlast allerdings in dieser Zeit dennoch überschritten werden, so können diese Prozesse Probleme bereiten. Dadurch, dass diese Anfragen intern gestartet werden, wird die Last nicht durch die bereits implementierten Entwurfsmuster verwaltet.

Um das zu verbessern definiert [Priy21] zusätzlich das Batch to Stream Entwurfsmuster. Dieses besagt, dass die eigentlich interne Batchverarbeitung wie eine externe Anfrage auf das System gestartet wird. So können die übrigen Entwurfsmuster die Last auf das System wie üblich verwalten.

Fazit

Zu Beginn der Arbeit wurde hergeleitet was Resilienz ist und warum die zunehmende Komplexität von Softwaresystemen ein wachsendes Interesse an Resilienz bewirkt. Als Grundgedanke wurde die Normal Accident Theory erklärt. Zudem wurden die Begriffe der Zuverlässigkeit und Robustheit von dem Begriff der Resilienz abgegrenzt.

Dieser Beitrag erarbeitete allerdings hauptsächlich die Bedeutung von Resilienz in Softwaresystemen. Dafür wurden im Rahmen des Resilience Management Model die Faktoren außerhalb der Software betrachtet, welche Einfluss auf die Systemresilienz haben. Die Betrachtung von Personen, Anforderungen, Umfeld und Tests wurde im Anschluss auf die Betrachtung eines verteilten Softwaresystems eingeschränkt.

Für dieses System wurden im Anschluss theoretische Prinzipien, anzustrebende Eigenschaften und konkrete Entwurfsmuster aufgeführt, welche es ermöglichen eine System resilient zu gestalten. Dabei darf allerdings nicht außer Acht gelassen werden, dass auch die Faktoren jenseits der Hard- und Software einen Einfluss auf die Resilienz haben. Somit ist es auch bei der Befolgung aller Anweisungen möglich ein nicht resilientes System zu haben. Dennoch bieten sie ein gutes Fundament um das System resilenter zu gestalten.

Zum Abschluss soll noch einmal darauf hin gewiesen werden, dass Resilienz in Softwaresystemen nur ein Mittel zum Zweck ist, um bestimmte Arbeitsabläufe in Unternehmen zu schützen. Um wichtige Prozesse resilient zu machen müssen alle Abhängigkeiten, nicht nur die Software beachtet werden.

Literatur

- [Char00] Charles Perrow: Normal Accidents, Living with High Risk Technologies - Updated Edition. Princeton University Press, Princeton, 2000, ISBN: 9781400828494.
- [HBO+18] Hickford, A.J.; Blainey, S.P.; Ortega Hortelano, A.; Pant, R.: Resilience engineering: theory and practice in interdependent infrastructure systems. Environment Systems and Decisions 38/3, S. 278–291, 2018, ISSN: 2194-5411.
- [Jona20] Jonathan Johnson: Resilience Engineering: An Introduction, BMC Software, Inc, 2020, URL: <https://www.bmc.com/blogs/resilience-engineering/>, Stand: 01. 11. 2022.
- [Micr] Microsoft Security Development Lifecycle, What are the Microsoft SDL practices?, Microsoft Corporation, URL: <https://www.microsoft.com/en-us/securityengineering/sdl/practices>, Stand: 25. 11. 2022.
- [Nati] Module 1: An Overview of Active Implementation Frameworks, Organization Drivers, National Implementation Research Network, URL: <https://nirn.fpg.unc.edu/module-1/implementation-drivers/organizational>, Stand: 25. 11. 2022.
- [Perr04] Perrow, C.: A Personal Note on Normal Accidents. Organization & Environment 17/1, doi: 10.1177/1086026603262028 doi: 10.1177/1086026603262028, S. 9–14, 2004, ISSN: 1086-0266.
- [Priy21] Priyank Gupta: 5 proven patterns for resilient software architecture design, hrsg. von TechTarget.com, Sahaj Software, 2021, URL: <https://www.techtarget.com/searchapparchitecture/tip/5-proven-patterns-for-resilient-software-architecture-design>.
- [Rich10] Richard A. Caralli, Julia H. Allen, Pamela D. Curtis, David W. White, Lisa R. Young: CERT Resilience Management Model, Version 1.0, CARNEGIE-MELLON UNIV PITTSBURGH PA SOFTWARE ENGINEERING INST, 2010-05-01, URL: <https://apps.dtic.mil/sti/citations/ADA522534>.
- [ThGe22] Thierry Meyer; Genserik Reniers: Engineering Risk Management. De Gruyter, Berlin, Boston, 2022, ISBN: 9783110665338.
- [Tv08] Tanenbaum, A. S.; vanSteen, M.: Verteilte Systeme, Prinzipien und Paradigmen. Pearson, 2008.
- [Wo16] Wolff, E.: Microservices, Grundlagen flexibler Softwarearchitekturen. dpunkt, 2016.
- [ZhLi10] Zhang, W.J.; Lin, Y.: On the principle of design of resilient systems – application to enterprise information systems. Enterprise Information Systems 4/2, doi: 10.1080/17517571003763380 doi: 10.1080/17517571003763380, S. 99–110, 2010, ISSN: 1751-7575.

Green IT - Wie kann der Verbrauch natürlicher Ressourcen in der IT-Branche reduziert werden?

Fabian Heinl¹ Philipp Kremling²

Abstract: Gerade heutzutage, inmitten des menschengemachten Klimawandels und mit immer weiter steigenden Energiepreisen, die eine Belastung für Unternehmen und Privathaushalte darstellen, ist es wichtig natürliche Ressourcen zu schonen und Strom zu sparen. Speziell die IT-Branche verbraucht durch Rechenzentren und immer mehr Software viel Energie. Der Ansatz Green IT beschreibt Herangehensweisen, wie der natürliche Ressourcenverbrauch von Software reduziert werden kann. In dieser Arbeit wird zunächst betrachtet, wie der natürliche Ressourcenverbrauch von Software messbar ist. Auch die aktuell eher gegensätzliche Bewegung im Bereich des Cloud Computings wird beschrieben. Anschließend werden konkrete Umsetzungsmöglichkeiten beschrieben werden. Hierbei liegt der Fokus auf der verwendeten Software-Architektur, Software-Pattern und Vorgehensmodellen. Zudem ist es wichtig, zu gewährleisten, dass die Software auch zukünftig nachhaltig bleibt, wozu im Anschluss Möglichkeiten evaluiert werden. Abschließend erfolgt eine Bewertung der vorgestellten Umsetzungsmöglichkeiten, die sowohl wirtschaftliche Aspekte und die Ressourceneinsparung, aber auch die Schwächen des Ansatzes herausstellt.

Keywords: Software Engineering; Green IT; Green Software Engineering

1 Einleitung

Im folgenden Kapitel soll der Leser oder die Leserin mit der Arbeit vertraut gemacht werden: Es wird dafür zunächst die Motivation und der generelle Aufbau der Arbeit erläutert. Im Zuge der besseren Lesbarkeit wird im Folgenden ausschließlich das generische Maskulinum verwendet. Alle weiteren Geschlechter sind hier jedoch gleichermaßen mit eingeschlossen.

1.1 Motivation

Die Erde wärmt sich durch den menschengemachten Klimawandel immer weiter auf. So war die durchschnittliche Temperatur 2021 um 1,11° höher im Vergleich zum vor-industriellen Level. [Wo] Dies hat dramatische Auswirkungen auf unseren Planeten und zeigt sich in immer stärkeren und heftigeren Naturkatastrophen. Als Beispiele seien hier die Dürre

¹ Duale Hochschule Baden Württemberg, Campus Horb, Florianstraße 15, 72160 Horb, Deutschland i20014@hb-dhbw-stuttgart.de

² Duale Hochschule Baden Württemberg, Campus Horb, Florianstraße 15, 72160 Horb, Deutschland i20022@hb-dhbw-stuttgart.de

in Europa oder das Jahrhunderthochwasser in Australien im Jahr 2022 genannt. Ein Hauptgrund der Erwärmung ist dabei der anthropogene Treibhauseffekt [Bu22]. Dabei ist der Energiesektor in Deutschland mit 82,8% die größte Quelle für Treibhausgase im Jahr 2020. [Um22]

Speziell die IT-Branche verbraucht durch Rechenzentren und immer mehr Software viel Energie. War der Informations- und Kommunikationssektor 2007 noch für 1,5% des weltweiten CO₂-Fußabdrucks verantwortlich, so wird vermutet, dass es 2040 bereits 14% sein werden. [Po20] Vor allem der Energiebedarf von Servern in deutschen Rechenzentren wird laut Facheinschätzungen von 2015 bis 2025 um mehr als 60% steigen. [Bu20] Aus diesem Grund werden vermehrt auch hier Strategien gefordert, mittels derer die IT-Branche ressourcenschonender auftreten kann. Zudem bei immer weiter steigenden Energiepreisen, die eine Belastung für Unternehmen und auch Privathaushalte darstellen, eine energieschonendere Soft- und Hardware auch wirtschaftliche Vorteile bietet. So besteht nicht nur beim Stromverbrauch Potenzial zum Einsparen, sondern auch bei der Herstellung und dem Recycling von Computern und deren Komponenten. [Sa08]

Der Ansatz Green IT beschreibt dabei Herangehensweisen, wie der natürliche Ressourcenverbrauch von Software reduziert werden kann. Hierbei muss der gesamte Lebenszyklus des Software-Produkts betrachtet werden. So muss während der Entwicklung ein ressourcenschonender Umgang der Technik im Betrieb, aber auch die umweltschonende Entsorgung und Wiederverwendung der Einsatzmaterialien Berücksichtigung finden. [LS]

Die Vorteile von Green IT liegen auf der Hand: Indem der Ressourcenverbrauch von Software reduziert wird, verbraucht das Unternehmen weniger Strom und spart somit Geld. Auch nach außen hin wird durch einen geringeren Stromverbrauch die indirekte Produktion von CO₂ durch die Stromherstellung reduziert. So können auch IT-Unternehmen einen aktiven und nicht zu unterschätzenden Beitrag zur Verringerung des Klimawandels leisten.

In dieser Arbeit wird zunächst betrachtet, wie der natürliche Ressourcenverbrauch von Software messbar ist. Anschließend werden konkrete Umsetzungsmöglichkeiten beschrieben. Diese Umsetzungsmöglichkeiten reichen von der verwendeten Hardware bis zur Softwarearchitektur und Software-Pattern. Zudem ist es wichtig, zu gewährleisten, dass die Software auch zukünftig nachhaltig bleibt. Abschließend erfolgt eine Bewertung der vorgestellten Umsetzungsmöglichkeiten, die sowohl wirtschaftliche Aspekte und die Ressourceneinsparung, aber auch die Schwächen des Ansatzes herausstellen soll.

1.2 Aufbau der Arbeit

Der Aufbau der Arbeit soll den Gedankengängen des Lesers folgen. Begonnen wird deshalb mit Green IT generell. Zunächst wird dieses Konzept definiert. Dabei soll vor allem die

Sinnhaftigkeit des Konzepts verdeutlicht werden.

Anschließend wird erläutert, wie der natürliche Ressourcenverbrauch von Software messbar ist und es werden aktuelle Entwicklungen im Bereich Green IT aufgezeigt, so zum Beispiel im Bereich des Cloud Computings. Dabei wird insbesondere darauf eingegangen, dass aktuell eher entgegengerichtete Ansätze zu Green IT sichtbar sind.

Nachdem die Grundlagen zu Green IT erläutert wurden, sollen konkrete Umsetzungsmöglichkeiten erklärt werden. Hierbei wird zunächst auf die Softwarearchitektur eingegangen. Auch diese hat einen Einfluss darauf, wie viele natürliche Ressourcen eine Software verbraucht. Es sollen dabei verschiedene Architekturentypen gegenübergestellt werden.

Anschließend werden bestimmte Software-Patterns beachtet durch die eine effizientere Software geschaffen und so der natürliche Ressourcenverbrauch durch diese reduziert werden kann.

Abschließend wird auch der Einfluss von Vorgehensmodellen während der Entwicklung eines Software-Produkts betrachtet. Auch diese können einen Einfluss auf den natürlichen Ressourcenverbrauch einer Software haben. Speziell agile Vorgehensweisen können mit Blick auf diese Tatsache optimiert werden.

Bisher wurde lediglich betrachtet, wie eine Software anfangs so erstellt werden kann, dass sie möglichst ressourcenschonend ist. Wenn diese Eigenschaft jedoch im weiteren Lebenszyklus des Software-Produkts vernachlässigt wird, ist dies nicht ausreichend. Aus diesem Grund wird anschließend auf Möglichkeiten eingegangen, wie gewährleistet werden kann, dass die Software auch zukünftig nachhaltig bleibt.

Im Anschluss werden das generelle Konzept Green IT und die verschiedenen vorgestellten Umsetzungsmöglichkeiten bewertet. Hierbei werden Aspekte der CO₂- und Ressourceneinsparung, aber auch wirtschaftliche Aspekte und Schwächen des Ansatzes wie Greenwashing, wirtschaftliche Widersprüchlichkeit und Reboundeffekte betrachtet.

Abschließend soll ein Ausblick gegeben werden, der nochmals die Vorteile von Green IT aufzeigt und weitere mögliche Verbesserungen und Entwicklungen in der Zukunft beschreibt.

2 Green IT

Green IT ist ein Begriff der auch durch die steigende Bedrohung durch den Klimawandel immer mehr Entwicklern bekannt wird. Dieser Begriff bezieht sich nicht nur auf den enormen Stromverbrauch durch Software oder durch die allgemeine IT-Infrastruktur, was auch der Öffentlichkeit in letzter Zeit bewusster wird. So wurde neben dem Energieverbrauch des Internets im Zuge des Bitcoin-Hypes auch viel über den enormen Stromverbrauch, der für das Mining der Kryptowährung nötig ist, in den Medien berichtet. Im Jahr 2020 wurden allein beim Mining für Bitcoins 75,4 TWh verbraucht, was mehr als der Bedarf Österreichs mit 69,9 TWh entspricht. [ZD22]. Nicht nur in Bezug auf Bitcoins und das

Internet wäre es also von Vorteil, wenn beim Betrieb von Software Energie gespart werden könnte. So gehen Experten davon aus, dass die Nachfrage an Rechenleistung allein in deutschen Rechenzentren von 2015 bis 2025 um mehr als 60% steigt. [Bu20]

Green IT bezieht sich jedoch nicht nur auf den Stromverbrauch, auch wenn dieser das Hauptaugenmerk dieser Arbeit ist. So wird im Gabler Wirtschaftslexikon folgende Definition gegeben: „[Green IT] bezeichnet die ressourcenschonende Verwendung von Energie und Einsatzmaterialien in der Informations- und Kommunikationstechnologie über den gesamten Lebenszyklus hinweg, d. h. dass bereits bei der Entwicklung nicht nur ein möglichst ressourcenschonender Umgang der Technik im Betrieb, sondern auch eine umweltschonende Entsorgung und Wiederverwendung der Einsatzmaterialien Berücksichtigung findet.“ [LS] Es wird also nicht nur der Betrieb, wie z. B. der Ressourcenverbrauch von Rechenzentren, beachtet und versucht klimafreundlicher zu gestalten. Auch bei der Entwicklung der Anwendungen und dem Aufbau von Infrastrukturen soll auf Ressourcenschonung geachtet werden. Hierzu werden im Folgenden einige Beispiele gegeben. Auch der Umgang mit der Hardware muss als weiterer Aspekt neben der Software Beachtung finden. Diese soll nicht nur möglichst klimafreundlich produziert und z. B. durch Ökostrom betrieben werden, auch bei der Entsorgung muss auf umweltschonende Prozesse und Recycling geachtet werden.

Im Rahmen dieser Arbeit wird der Fokus jedoch auf die Ansätze für ressourcenschonendere Softwareentwicklung (auch als Green Software Engineering bekannt) gelegt. So wird zwar durch Hardware die Energie verbraucht, jedoch wird der Energieverbrauch von der Software erst ausgelöst. Asmin Hussain, Leiter der Interessensvertretung für Green Cloud Computing bei Microsoft, beschreibt zwei Philosophien grüner Softwareerstellung: Zum einen hat jeder einen Teil an der Lösung des Klimawandels und somit einen Einfluss auf die Zukunft. Zum anderen gibt es viele Vorteile bei nachhaltiger Software: Diese sind oft günstiger, performanter und resilenter. Der Aspekt der Nachhaltigkeit sollte jedoch der Hauptfokus von Green Software Engineering sein. Liegt der Fokus stattdessen z. B. auf Performanz, kann der Aspekt der Nachhaltigkeit schnell vernachlässigt werden. Neben den Philosophien können laut Hussain unabhängig von Anwendungsbereich, Industriebereich, Größe oder Art der Organisation, Hosting und Programmiersprache acht Prinzipien der nachhaltigen Softwareentwicklung Beachtung finden: „Carbon, Electricity, Carbon Intensity, Embodied Carbon, Energy Proportionality, Networking, Demand Shaping [und] Measurement & Optimization“ [Hu21] Diese zielen nicht alle direkt auf den Stromverbrauch ab, sind jedoch für das Ziel ressourcenschonender IT auch ein wichtiger Faktor, weshalb diese Folgend ein weniger genauer beleuchtet werden.

Carbon

Treibhausgase sind durch ihre wärmende Wirkung ein Faktor, warum Leben auf der Erde überhaupt möglich ist. Durch den Ausstoß großer Mengen von Treibhausgasen, den die Menschheit seit der industriellen Revolution verursacht, ändert sich jedoch das Klima schneller, als sich Flora und Fauna anpassen können. Das von den Vereinten Nationen aufgestellte und von 195 Staaten unterzeichnete Pariser Klimaabkommen, versucht dem entgegenzuwirken. Es zielt darauf ab den Kohlenstoffausstoß zu minimieren und so

den Temperaturanstieg bis 2100 um 1,5°C im Vergleich zu vorindustriellen Werten zu stabilisieren. Anwendungen sollen also für jedes Gramm Kohlenstoff, das ausgestoßen wird, den gleichen Mehrwert für Nutzer bieten, aber weniger CO2-Emissionen verursachen.

Electricity

Der meiste Strom wird durch die Verbrennung fossiler Brennstoffe erzeugt. Auch beim vorher erwähnten Beispiel Bitcoin-Mining ist dies der Fall. So werden in vielen Fällen Kohlekraftwerke zur Erzeugung des benötigten Stroms verwendet. Es kann also eine Beziehung zwischen Stromverbrauch von Software und Treibhausgasemission hergestellt werden. So beginnt es nicht beim Computer und den Anwendungen an sich, sondern schon bei der Herstellung des Stroms. Indem ein Entwickler den Energieverbrauch seiner Programme minimiert, wird ebenfalls Energie eingespart. Eine Möglichkeit dafür ist die Verwendung effizienterer Algorithmen, um so ein leistungsfähigeres Programm zu schaffen. [Hu21]

Carbon Intensity

Die Kohlenstoffintensität beschreibt, wie viele Kohlenstoffemissionen pro verbrauchter Kilowattstunde Strom entstehen, also wie viel Kohlenstoff pro Kilowattstunde ausgestoßen wird. Ist ein Computer z. B. direkt an einem Wasserkraftwerk oder einer Solaranlage angeschlossen, wäre die Kohlenstoffintensität gleich null, vorausgesetzt diese nachhaltigen Methoden produzieren genug Strom. Die Kohlenstoffintensität ändert sich je nach Standort, da überall ein unterschiedlicher Energiemix vorhanden ist. Das Problem mit erneuerbaren Energien ist jedoch, dass diese nicht immer gleichmäßig vorhanden und von der Wetterlage abhängig sind. Als Basis werden deshalb im Stromnetz fossile Energien benutzt, um dem Gesamtverbrauch gerecht zu werden. Strom aus erneuerbaren Energien wird bei einem größeren Angebot als Nachfrage gekürzt. Ist man als Softwareentwickler flexibel in der Ausführung der Arbeitslasten, kann in einem solchen Fall die Nachfrage erhöht werden und so die geringere Kohlenstoffintensität ausnutzen. So ist es z. B. möglich, dass ein maschinelles Lernmodell in einer Region oder zu einem Zeitpunkt trainiert wird, zu dem viele erneuerbare Energien im Energiemix enthalten sind, sodass weniger Kohlenstoff ausgestoßen wird. Studien konnten zeigen, dass Maßnahmen solche Maßnahmen, je nach Anzahl der erneuerbaren Energien im Stromnetz, den ausgestoßenen Kohlenstoff um 45 – 99% reduzieren können. [CEA11]

Embodied Carbon

Für die Herstellung jedes Geräts, darunter z. B. auch Komponenten für Server, wurde bei deren Herstellung Kohlenstoff freigesetzt. Auch am Ende der Lebensdauer des Geräts kann durch die Entsorgung Kohlenstoff freigesetzt werden. Bei der Berechnung der Kohlenstofffreisetzung einer Hardwarekomponente dürfen also nicht nur die Kosten für den Betrieb einberechnet werden, sondern auch die der Herstellung und Entsorgung. Vermindern kann man diesen Wert durch die Verlängerung der Nutzungsdauer von Hardware. So werden alte Geräte entfernt, weil diese defekt oder nicht mehr in der Lage sind, neue Aufgaben zu bewältigen. Defekte Geräte lassen sich aus Sicht des Softwareentwicklers nicht vermeiden. Wird Software jedoch so entwickelt, dass weniger Geräte überlastet werden, müssen diese

nicht durch leistungsstärkere Geräte ersetzt werden. Zudem können Anwendungen auch so entwickelt werden, dass diese von älterer Hardware noch genutzt werden können.

Energy Proportionality

Die Auslastung gibt an, wie stark die Ressourcen eines Computers genutzt werden. Im Leerlauf ist diese geringer als bei voller Last. Die Energieproportionalität ist ein Maß für das Verhältnis von verbrauchter Energie und der Auslastung einer Hardware-Komponente. Im Fall von Hardwarekomponenten ist die Energieproportionalität nicht konstant und variiert je nach Kontext. So verbraucht ein Computer bei 0% Auslastung z. B. 100W, bei 50% 180W und bei 100% 200W an Energie. Je mehr die Hardware genutzt wird, desto effizienter ist diese bei der Umwandlung von Strom in Rechenoperationen. Werden also möglichst wenig Server auf höchster Auslastung beschäftigt, maximiert man deren Energieeffizienz.

Networking

Das Internet besteht aus einer großen Menge von Routern und Servern und anderen Geräten. Diese verbrauchen Strom und setzen so abhängig von der lokalen Kohlenstoffintensität auch Kohlenstoff frei. So erzeugen alle Daten, die man über das Internet sendet oder empfängt, Kohlenstoffemissionen. Diese sind abhängig vom Umfang der Daten, Entfernung, Anzahl der Sprünge zwischen Netzwerkgeräten, Energieeffizienz dieser, der Kohlenstoffintensität der verwendeten Energie und dem Netzwerkprotokoll. Als wichtigste Faktoren gelten dabei die Größe und Entfernung der Pakete. Dieser Fakt kann bei der Konzipierung und Entwicklung der Software einbezogen werden, sodass z. B. die Größe der von einer Anwendung verschickten Pakete möglichst klein gehalten wird. [Hu21]

Demand Shaping

Nachfragestrukturierung ist eine ähnliche Strategie wie die Nachfrageverschiebung. Statt die Rechenleistung örtlich oder zeitlich zu verlagern, wird hier die Energie-Nachfrage an das bestehende Angebot angepasst. Wenn also gerade viel erneuerbare Energie im Energiemix enthalten ist, wird auch der Umfang der Anwendung erhöht. So kann z. B. die Benutzererfahrung geändert werden, um die Kohlenstoffemissionen zu verringern. Dies ist vergleichbar mit Energiesparmodi, die z. B. in modernen Spül- und Waschmaschinen vorkommen. Sind diese eingeschaltet, wird die Leistung des Geräts gedrosselt, sodass für die Erfüllung derselben Aufgaben weniger Ressourcen verbraucht werden, indem z. B. die benötigte Zeit erhöht wird. Ob der Energiesparmodus eines Geräts durch Benutzer manuell eingestellt werden kann oder automatisch verwaltet wird, muss während der Entwicklung entschieden werden. Auf Kosten der Rechenleistung kann so also zusätzlich Kohlenstoff eingespart werden.

Measurement & Optimization

Bei der Erstellung einer nachhaltigen Software, genügt nicht nur eine Optimierung, sondern es benötigt deutlich mehr. So muss die Anwendung im Ganzen betrachtet werden und von der Benutzererfahrung bis zur Vernetzung und den Rechenzentren schrittweise vorgegangen werden. Die Wahl der Messkriterien hat dabei einen großen Einfluss auf den Erfolg der Dekarbonisierung einer Anwendung. So ist es z. B. nicht sinnvoll bei einer Anwendung die

Netzwerklast zu verringern, wenn die Datenbankabfragen deutlich häufiger vorkommen und deshalb deutlich mehr Kohlenstoff freisetzen. Die verbrauchte Energie einer Anwendung kann je nach Zeitpunkt und Ort variieren, jedoch ist weniger Stromverbrauch bei gleicher Leistung bereits ein guter Indikator, um die Nachhaltigkeit des Softwareprodukts zu bestimmen. Auch die Kosten für Strom und Hardware können als Indikator verwendet werden. So ist eine Anwendung, die keine hohe Last erzeugt, meist auch mit einem geringem Kohlenstoffausstoß verbunden. Auch die Vernetzung muss betrachtet werden. So kann durch Messung und anschließende Verringerung der Menge und Entfernung, die die Daten zurücklegen müssen, auch Kohlenstoff eingespart werden. [Hu21] Die Messbarkeit von Green IT wird im nächsten Kapitel genauer erläutert.

2.1 Messbarkeit

Nachdem im letzten Kapitel der Grundbegriff Green IT näher betrachtet wurde, wird in diesem Kapitel auf die Messbarkeit der Einsparungen eingegangen. Die acht Prinzipien zeigen die verschiedenen Einsparpotenziale, die man bei nachhaltiger Softwareentwicklung beachten sollte. Im Zuge dieser Arbeit wird dabei vor allem auf die Stromeinsprung eingegangen. Seit 2020 gibt es in Deutschland mit der Einführung des „Blauen Engels für ressourcen- und energieeffiziente Software“ ein für Endnutzer verlässliches Label, das eine grüne Software auszeichnet. [Ac22] Dabei werden die Anforderungen zum Erhalt des Labels in drei Bereiche eingeteilt: Ressourcen- und Energieeffizienz, potenzielle Hardware-Nutzungsdauer und Nutzungsautonomie. Ressourcen- und Energieeffizienz gibt dabei auch nützliche Anhaltspunkte zur Messbarkeit.

Als Bezugsgrößen zur Berechnung der Ressourcen- und Energieeffizienz werden die Hardwareressourcen und der Energieverbrauch verwendet. Mit Hilfe eines Referenzsystems wird die elektrische Leistungsaufnahme im Leerlauf gemessen. Dabei werden folgende Werte im Leerlauf ermittelt und der Mittelwert berechnet: Prozessorauslastung (%), Arbeitsspeicherbelegung (Mbyte), Permanentspeicherbelegung (Mbyte/s), beanspruchte Bandbreite für Netzzugang (MBit/s) und die elektrische Leistungsaufnahme (W). Diese Daten sind über das Betriebssystem einsehbar und werden z. B. im Windows Task-Manager unter „Leistung“ visualisiert. Zusätzlich zur Leistung im Leerlauf wird für den Blauen Engel auch die Hardware-Inanspruchnahme und der Energiebedarf bei Ausführung eines Standardnutzungsszenarios beachtet. Grundlage ist dabei ein Referenzsystem, wobei nicht nur die zusätzliche Inanspruchnahme, sondern auch ein prozentualer Anteil der Grundauslastung bei der Berechnung einbezogen wird. Gemessen werden dabei Prozessorarbeit (% * s), Arbeitsspeicherarbeit (Mbyte * s), Permanentspeicherarbeit (Lesen und Schreiben) (Mbyte/s * s), Übertragene Datenmenge für Netzzugang (MBit/s * s) und der Mittelwert des Energiebedarfs (Wh). Dabei ist darauf zu achten, dass die Software keinen Einfluss auf das Energiemanagement des Systems (z. B. Standby-Modus) hat und umgekehrt die Funktionalität der Software auch nicht durch etwaige Funktionen eingeschränkt wird. [Bl20]

In Abbildung 1 wird ein beispielhafter Messaufbau erläutert, mit dem diese Messwerte ermittelt werden können.

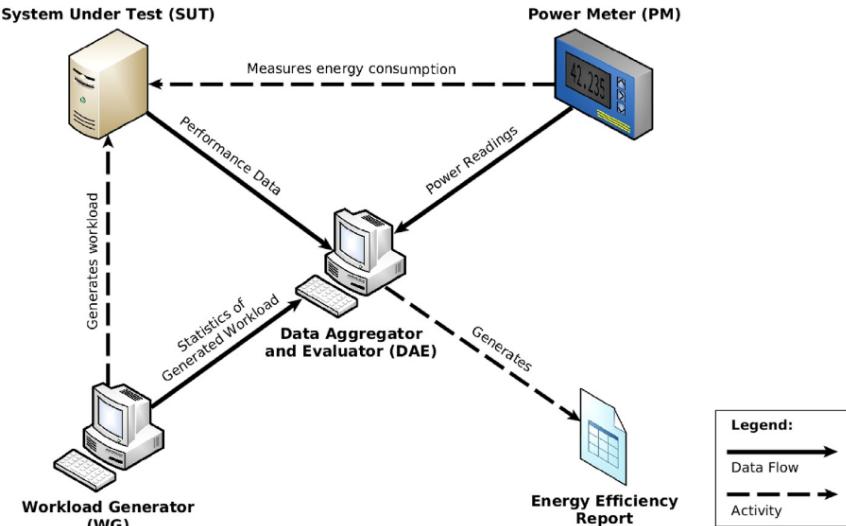


Abb. 1: Exemplarischer Aufbau zur Messung der Hardwareauslastung und des Energieverbrauchs eines Softwaresystems.[Ke18]

Wie in Abbildung 1 zu sehen, besteht der Testaufbau aus folgenden Komponenten:

- System Under Test (SUT)
- Leistungsmessgerät (PM)
- Workload Generator (WG)
- Datensammlung und -analyse (DAE)

Folgend werden die einzelnen Komponenten genauer erläutert und anschließend der Ablauf einer beispielhaften Messung erklärt:

System Under Test (SUT)

Bei einer Messung wird ein Szenario mit dem zu analysierenden Software-Produkt erstellt, welches dann auf dem SUT ausgeführt wird. Dabei besteht ein SUT z. B. aus einem PC, Server, Mobilgerät oder auch IoT-Gerät. Zudem wird es den Anforderungen entsprechend modifiziert. Während eines Durchlaufs zeichnet das SUT selbstständig die eigenen Leistungsdaten wie CPU- und RAM-Nutzung, Netzwerkauslastung etc. auf. Diese werden nach Abschluss des Tests ausgelesen und analysiert.

Leistungsmessgerät (PM)

Da das SUT seine benötigte Energie nicht selbst messen kann, wird dem Versuchsaufbau

ein „Power Meter“ hinzugefügt. Dieses misst die Leistungsaufnahme des SUT während der Ausführung des Software-Produkts. Dies ist also die Erfassung des Stromverbrauchs, der dann am Ende verglichen werden kann.

Workload Generator (WG)

Der dritte Bestandteil des Versuchsaufbaus ist der Workload Generator. Dieser erzeugt eine Last auf dem SUT. Dies ist z. B. durch Skript-Ausführungen, Aufrufe einer API, Website oder Datenbank möglich. Ein WG kann eine Desktop-Anwendung aber auch ein Tool zur Ausführung von vorher aufgezeichneten Eingaben sein, das sich durch die Programmoberfläche der Software klickt. Bei jedem Schritt wird dabei in einer Logdatei ein Start- und End-Zeitstempel gespeichert.

Datensammlung und -analyse (DAE)

Die Kernkomponente des Aufbaus ist das Modul in der Mitte von Abbildung 1: „Data Aggregator and Evaluater“. Dieser dient als Sammelpunkt für alle Daten, die während einer Szenarioausführung generiert werden. Es werden alle Informationen zur Hardwarenutzung, Leistungsaufnahme und der Logdateien aggregiert.

Ablauf

Bevor eine Messung auf einem SUT ausgeführt wird, wird das System auf einen Zustand vor Installation der Software zurückgesetzt, um eine Störung durch vorherige Tests zu vermeiden. Ist das Szenario fertig eingerichtet, wird dieses mehrere Male ausgeführt und die Werte anschließend gemittelt. Dies vermindert den Einfluss von externen Faktoren wie Background-Prozessen des Betriebssystems, die unabhängig der Software Last auf dem System erzeugen. Auch die Grundlast des Systems muss zuvor ohne die zu testende Software ermittelt werden, um anschließend von der gesamten Last abgezogen zu werden. [Ke18]

Im Anschluss an die Messungen können die gesammelten Daten analysiert und ggf. grafisch dargestellt werden. In Abbildung 2 ist zu sehen, dass „media player 2“, bei der Ausführung der gleichen Aufgaben im Durchschnitt mit 2,78 W weniger Energie als „media player 1“ verbraucht. Mittels dieser Technik lässt sich also eine simple Beurteilung der Ressourceneffizienz erstellen. [Ac22]

Im Bereich des Cloud Computings gibt es von einigen Cloud-Anbietern bereits Werkzeuge, um den Energieverbrauch und den Einfluss eines Software-Produkts auf die Umwelt ohne großen Versuchsaufbau zu messen. Eines dieser Werkzeuge ist Carbon Footprint von Google (siehe Abbildung 3). Mit Carbon Footprint können Kunden der Google Cloud die Brutto-Kohlenstoffemissionen ihrer Cloudnutzung angezeigt bekommen. Dabei ist Carbon Footprint kostenlos in die Anwendungsoberfläche integriert. Neben den kumulierten Brutto-Kohlenstoffemissionen im Laufe der Zeit können diese auch nach Produkt und nach Region gruppiert überwacht werden. IT-Teams und Entwicklern werden somit Kennzahlen bereitgestellt, die dabei helfen können, den CO2-Fußabdruck ihres Unternehmens zu reduzieren. Auch wenn die Emissionen der digitalen Infrastruktur nur

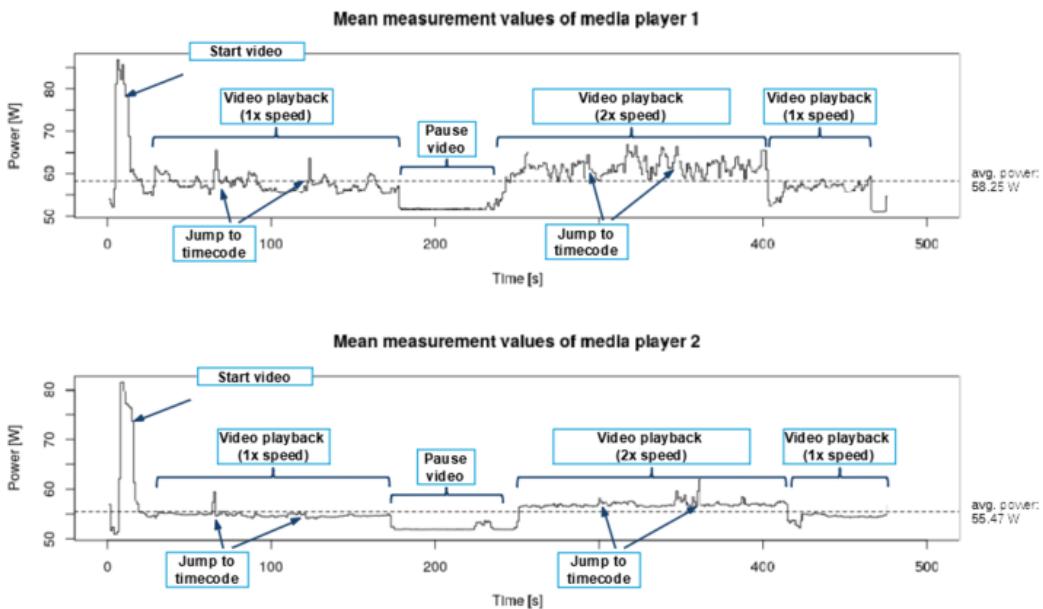


Abb. 2: Vergleich der Leistungsaufnahme von zwei Media-Playern beim Abspielen einer Videodatei. [Ac22]

einen Teil des ökologischen Fußabdrucks eines Unternehmens ausmachen, ist eine genaue Bilanzierung der IT-Emissionen notwendig, um die Fortschritte bei der Erreichung der Kohlenstoffreduzierung zu messen, die zur Abwendung der schlimmsten Folgen des Klimawandels erforderlich sind. [TC21]

Neben der Anzeige der Brutto-Kohlenstoffemissionen werden auch konkrete Handlungsempfehlungen gegeben. So erkennt Unattended Project Recommender beispielsweise Projekte, die aufgrund ihres Nutzungsverhaltens aufgegeben werden könnten und zeigt diese dem Anwender an. Google gibt an, dass so im August 2022 insgesamt über 600.000 kg CO₂ mit gefundenen sanierungsbedürftigen Projekten verbunden waren. [TC21]

So lässt sich also abschließend aussagen, dass bereits Methoden zur Messung des Verbrauchs von Ressourcen gegeben sind. So kann der Stromverbrauch eines SUT mit dem im Kapitel beschriebenen Versuchsaufbau bestimmt werden. Aber z.B. auch Google bietet mit dem Carbon Footprint schon eine Übersicht über die Brutto-Kohlenstoffemissionen einer bei ihnen gehosteter Cloud-Lösung.

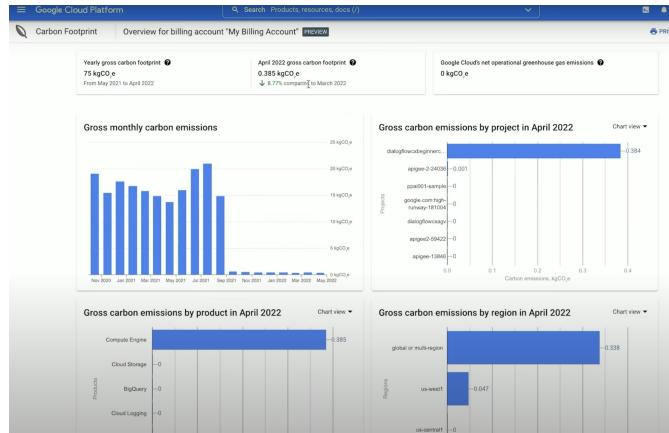


Abb. 3: Übersichtsseite von Google Carbon Footprint [Go22]

2.2 Derzeitige Entwicklungen

Im Folgenden wird auf die derzeitigen Entwicklungen im IT-Markt eingegangen und inwiefern diese einen Trend hin zur Green IT zeigen oder sogar gänzlich in die andere Richtung. Der wohl größte Trend ist aktuell wohl das wachsende Interesse an der „Cloud“. So bietet Cloud Computing potenziell einen finanziellen Vorteil, da Kunden einen großen, zentral verwalteten Pool von Speicher- und Rechenressourcen gemeinsam nutzen, statt eigene Systeme verwalten zu müssen. [D 09] Fraglich ist jedoch, ob diese zentrale Verwaltung auch auf dem Gebiet der natürlichen Ressourcen sparsamer ist. Der Artikel „Green Cloud Computing: Balancing Energy in Processing, Storage, and Transport“ beschäftigt sich mit genau dieser Frage. Die Erkenntnisse aus diesem werden nachfolgend skizziert. Der Begriff Cloud umfasst eine große Menge verschiedener Modelle und Angebote. Der Artikel beleuchtet dabei mehrere Teilgebiete der Cloud, in dieser Arbeit soll jedoch nur auf die zwei Größten davon eingegangen werden: Software as a Service (SaaS) und Storage as a Service (STaaS). SaaS meint das Anbieten einer kompletten Anwendung über die Cloud während STaaS das Outsourcen auf einen Datenspeicher meint. Dabei werden die Daten hochgeladen und müssen für jede Änderung lokal heruntergeladen werden. [Wi22] Wie groß der Energiebedarf der einzelnen Gebiete im Vergleich zur ausschließlichen lokalen Nutzung ist, soll folgend erläutert werden.

Software as a Service

Abbildung 4 zeigt ein Diagramm, das den Stromverbrauch (W) pro Nutzer angibt, gemessen an Bildern pro Sekunde (1/s). Ändern sich jede Sekunde 100% des Bildschirms entspricht dies 1 Bild pro Sekunde. Angegeben sind jeweils private und öffentliche Services mit 20 und 200 Nutzern pro Server. Privat und öffentlich bezieht sich dabei auf die Art des Transport-Netzwerks. Bei Ersterem wird ein firmeninternes Netz verwendet, in Letzterem

das öffentliche Internet. Ergänzend wird der Stromverbrauch eines leistungsschwachen und eines durchschnittlichen Laptops im Leerlauf gegenübergestellt.

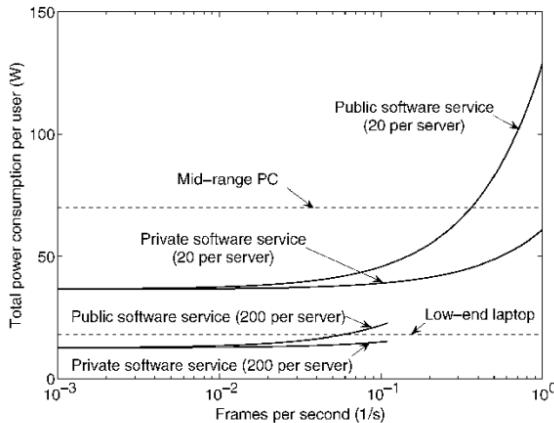


Abb. 4: Stromverbrauch pro Nutzer einer privaten und öffentlichen Cloud je Bild pro Sekunde Rate. [Ba11]

Abbildung 4 zeigt, dass vor allem bei wenigen Veränderungen der Seite einer Anwendung die Cloud-Lösungen mit 200 Nutzern deutlich stromsparender sind als eigene Rechner, da diese zum Teil im Leerlauf bereits mehr Energie verbrauchen. Bei einer hohen Anzahl an Änderungen der 20 Nutzer steigt der Energiebedarf jedoch signifikant an.

Storage as a Service

Einen ähnlichen Verlauf zeigt auch Abbildung 5. Bei einer geringen Anzahl an Downloads, die bei Änderungen notwendig sind, sind die Cloud Versionen um einiges ressourcenschöner im Vergleich zu einer Laptop-internen HDD. Erst ab ca. 9 Downloads pro Stunde wäre dies nicht mehr der Fall.

Auch für die anderen Teilgebiete wird im Artikel deutlich, dass die Cloud Variante meist stromsparender ist als eine lokale Version. Dies ist vor allem bei geringer Nutzung der Fall und trifft erst für hohe Lasten nicht mehr zu, wie in Abbildung 4 und Abbildung 5 zu sehen ist. [Ba11]

Die herkömmliche Cloud ist also nicht in den meisten Fällen bereits jetzt ressourcenschöner als der klassische, lokale PC. Zudem gibt es einen wachsenden Bedarf und Forschungsarbeiten hinsichtlich „Green Cloud Computing“, die weitere Potenziale bei der Einsparung von Ressourcen und Strom ausschöpfen wollen. So z.B. auch die Entwicklung von eigens auf die Einsparung von Energie ausgelegten Frameworks. [CS13] Weitere Umsetzungsmöglichkeiten für Green IT sollen im nächsten Kapitel erläutert werden.

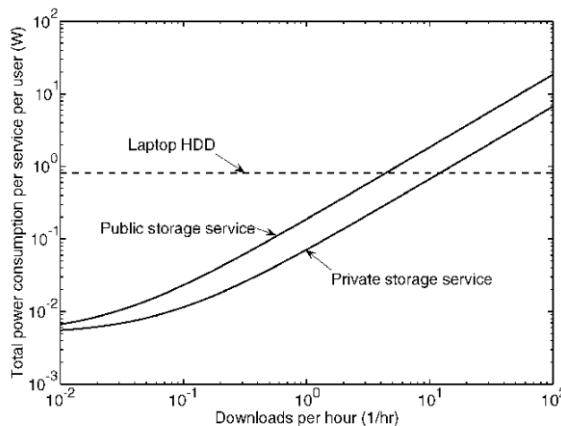


Abb. 5: Pro Nutzer Stromverbrauch je Downloadrate im Vergleich zu einer modernen Laptop HDD. Die durchschnittliche Dokumentengröße beträgt 1,25MB [Ba11]

3 Umsetzungsmöglichkeiten

Im Folgenden werden konkrete Umsetzungsmöglichkeiten des Konzepts Green IT beschrieben. Dabei wird zunächst auf Möglichkeiten durch die verwendete Softwarearchitektur eingegangen. Anschließend werden bestimmte Software-Patterns betrachtet und evaluiert, wie durch eine Effizienzsteigerung des Software-Produkts durch diese auch der natürliche Ressourcenverbrauch reduziert werden kann. Abschließend werden Vorgehensmodelle während der Entwicklung eines Software-Produkts betrachtet und deren Einfluss auf den natürlichen Ressourcenverbrauch in der IT-Branche herausgestellt.

3.1 Softwarearchitektur

Als erster Teilbereich der Umsetzungsmöglichkeiten soll in diesem Kapitel die Softwarearchitektur beleuchtet werden und mögliche Einsparpotenziale betrachtet werden. In der Literatur gibt es zurzeit noch wenig standardisierte Metriken, die eine Bewertung/Vorhersage der Umweltfreundlichkeit eines bestimmten Softwaresystems, insbesondere während der Entwurfsphase bestimmen könnten. So ist es eine große Herausforderung, insbesondere für Software-Architekten, neue Anwendung möglichst grün zu gestalten. [Me22] Auch für bekannte Muster, z. B. Monolithen und Microservices, gibt es noch keine genaueren Studien und Auswirkungen. Unter Beachtung von wenigen allgemeinen Grundsätzen kann auch die Energieeffizienz der Softwarearchitektur verbessert werden.

Data design, usage and storage

Um eine gewisse Speicherkapazität bereitzustellen, wird Hardware benötigt, die wiederum

Strom verbraucht und zuvor hergestellt werden muss. Ziel sollte also sein, möglichst wenig Daten zu speichern. Auch sollte unnötiger Datenverkehr über das Netzwerk vermieden werden, da dies auch Strom und Ressourcen verbraucht, wie im Kapitel Green IT erklärt.

Application Design

Für verschiedene Komponenten sollten Richtlinien definiert werden. So sollte nicht jede Komponente immer aktiv sein, sondern falls sie nicht benötigt wird, abgeschaltet oder zumindest in den Standby-Modus gesetzt werden. Auch ist eine parallele oder asynchrone Verarbeitung sinnvoll. So kann eine nicht zeitkritische Aufgabe während einer Zeit mit möglichst geringer Kohlenstoffintensität berechnet werden.

Platform deployments, utilization, and scaling

Ein weiterer wichtiger Punkt ist das Deployment und die Skalierung von Software. So kann bei der Konzipierung der Software auf eine gute Skalierbarkeit der Anwendung geachtet werden. Auch auf den Overhead von Host-Möglichkeiten sollte eingegangen werden. So wird z. B. eine Ausführung der Anwendung abgeschalten, das zugehörige Management-Tool bleibt jedoch aktiv.

Code efficiency

Auch durch den Quelltext selbst gibt es Einsparmöglichkeiten. Code, der aufgrund der Architektur nicht existiert, ist der schnellste Quelltext und somit auch der, der am wenigsten Energie verbraucht. Auch „unnötige“ Arbeit wie z.B. das Erstellen von Log-Dateien, könnte durch verschiedene Service-Level-Agreements für Anwendungen, die dies nicht benötigen deaktiviert werden. Solche nur teilweise benötigten Funktionalitäten können in der Architektur berücksichtigt werden. [Ei]

Zusammenfassend gibt es also zwar wenige Vergleiche zwischen konkreten Architekturen, anhand einiger Grundsätze, die auch schon im Kapitel Green IT beschrieben wurden, können jedoch mit bestimmten Design-Entscheidungen Erfolge erzielt werden.

3.2 Software-Entwurfsmuster

Auch eine Einsparung von Ressourcen durch die Verwendung oder Vermeidung bestimmter Software-Entwurfsmuster ist möglich. Softwareentwickler können durch ein besseres Verständnis der Auswirkungen von Design- und Implementierungsentscheidungen auf Anwendungsebene eine wichtige Rolle bei der Reduzierung des Energieverbrauchs der von ihnen geschriebenen Anwendungen einnehmen. [Sa12]

[Sa12] untersuchte den Energieverbrauch verschiedener Entwurfsmuster. Dabei wurde für dasselbe Programm ohne ein Entwurfsmuster und unter Verwendung eines Entwurfsmusters der jeweilige Energieverbrauch in Joule mittels eines dafür entwickelten Programms gemessen. Die Entwurfsmuster wurden dabei in die drei Kategorien Erstellungsmuster, strukturelle Muster und Verhaltensmuster eingeteilt. Im Folgenden werden die wichtigsten Erkenntnisse daraus dargestellt.

Erstellungsmuster befassen sich mit der Bereitstellung alternativer Wege zur Erstellung von Objekten. Während die Entwurfsmuster Builder (+1,19%) und Singleton (+0,42%) eine Zunahme im Energieverbrauch bewirken, kann mit Factory Method (-0,07%) eine Reduzierung bewirkt werden. Die höchste Zunahme im Energieverbrauch lässt sich dabei unter Verwendung des Abstract-Factory-Patterns feststellen (+21,55%). Unter Verwendung des Prototyp-Patterns hat sich der Energieverbrauch der Software um -0,93% reduziert.

Strukturelle Muster befassen sich mit der Klassen- und Objektkomposition. Das Entwurfsmuster Kompositum erhöht dabei den Energieverbrauch um +5,14%. Besonders signifikant ist die Erhöhung des Energieverbrauchs aber unter Verwendung des Decorator-Patterns um +712,89%. Die Entwurfsmuster Bridge (-0,24%), Flyweight (-58,08%) und Proxy (-36,47%) reduzieren den Gesamtenergieverbrauch der Software. Das Decorator-Pattern ist das Entwurfsmuster, welches in dieser Evaluation prozentual am meisten zusätzliche Energie benötigt. Als Erklärung könnte hier dienen, dass unter Verwendung des Decorator-Patterns komplexe Objekte dynamisch und ohne Vererbung erzeugt werden. Die Flexibilität, die das Decorator-Pattern bietet, bringt also gleichzeitig einen höheren Energieverbrauch mit sich, da die Struktur des Objekts nicht bereits zur Kompilierungszeit vollständig festgelegt ist.

Verhaltensmuster befassen sich mit der Kommunikation zwischen Objekten. Eine Erhöhung des Energieverbrauchs lässt sich hier nur unter Verwendung des Observer-Patterns um +62,20% feststellen. Die Entwurfsmuster Command (-1,82%), Mediator (-9,56%), Strategy (-0,18%) und Visitor (-7,49%) reduzieren jeweils den Energieverbrauch.

Zusammenfassend kann gesagt werden, dass bei einigen Entwurfsmustern die Auswirkung sehr gering, bei anderen mäßig und bei wenigen signifikant ist. Aber auch die Implementierung von Entwurfsmustern, die nur eine geringe Einsparung mit sich bringen, kann schon lohnend sein. In einer typischen Anwendung kann die Implementierung eines Entwurfsmusters millionen- oder milliardenfach ausgeführt werden. Somit kann auch ein geringer Unterschied je Iterationen einen signifikanten Unterschied im Gesamtenergieverbrauch zur Folge haben. Die Anwendung dieser Entwurfsmuster kann also den Energieverbrauch eines Software-Produkts sowohl erhöhen als auch verringern. Dabei besteht kein Zusammenhang zur Kategorie des Entwurfsmusters. [Sa12]

Trotzdem sieht man in diesem Test mit traditionellen Software-Entwurfsmustern vor allem einen Anstieg des Energieverbrauchs. [CMP21] rät deshalb für kleine Komponenten davon ab, diese Entwurfsmuster zu verwenden und schlägt stattdessen eigene, auf die Reduktion des Energieverbrauchs angepasste, Software-Entwurfsmuster vor.

Wakelock

Ein Wakelock ist ein Mechanismus, der das Ausschalten des Bildschirms oder das Aktivieren des Energiesparmodus eines Geräts unterbindet. Er kommt vor allem bei Smartphones zum Einsatz. Wenn der Wakelock nicht ordnungsgemäß freigegeben wird, wird Energie verbraucht, die eingespart werden könnte. Durch das Hinzufügen einer Freigabeanweisung lässt sich dieses Problem umgehen. [CMP21]

Recycle

Recycle beschreibt das Vorgehen, Objekte wie Sammlungen oder Datenbankverbindungen nach ihrer Verwendung wieder zu schließen. Geschieht dies nicht, bleibt z. B. die Verbindung geöffnet und andere Objekte desselben Typs können die gleichen Ressourcen nicht effizient nutzen. [CMP21]

Übermäßige Methodenaufrufe

Für jeden Methodenaufruf ist es nötig, dass alle Argumente auf den Stack gelegt, der Rückgabewert im Prozessorregister gespeichert und der Stack aufgeräumt werden muss. Wird eine Methode nun innerhalb einer Schleife mehrfach aufgerufen, wird durch diese Prozedur der Energieverbrauch der Anwendung gesteigert. Als Alternative können z. B. Methodenaufrufe ohne Parameter und auf die durch ein Objekt zugegriffen wird, das innerhalb der Schleife nicht umgewandelt wird, ersetzt werden. Eine Variable, die außerhalb der Schleife deklariert und mit dem Rückgabewert des extrahierten Methodenaufrufs initialisiert wird, kann ohne erneuten Methodenaufruf bei jedem Schleifendurchlauf integriert werden. [CMP21]

Dynamische Wiederholungsverzögerung

Wenn von einer Anwendung Daten mit verschiedenen anderen Ressourcen, wie einem Server, ausgetauscht werden, kann es vorkommen, dass die Kommunikation zwischen diesen Ressourcen fehlschlägt. Ist dies der Fall, muss ein erneuter Versuch des Datenaustauschs unternommen werden. Ist die angefragte Ressource jedoch über einen längeren Zeitraum nicht verfügbar, wird die Anwendung wiederholt versuchen, eine Verbindung herzustellen, was zu einem unerwünschten Energieverbrauch führt. Stattdessen kann nach jedem fehlgeschlagenen Verbindungsversuch die Wartezeit vor dem nächsten Versuch linear oder exponentiell erhöht werden. Nach erfolgreicher Verbindung wird die Wartezeit wieder auf den Ursprungswert gesetzt. [CMP21]

Push over Poll

Um eine Aktualisierung von Anwendungsdaten zu bewirken, werden von der Anwendung typischerweise Aktualisierungen bei einer anderen Ressource angefragt, zum Beispiel bei einem Server. Dieses kontinuierliche Anfragen von aktualisierten Daten kann jedoch dazu führen, dass einige Anfragen getätigt werden, obwohl keine aktualisierten Daten vorliegen. Als Alternative können Push-Anfragen statt Poll-Anfragen verwendet werden. Das bedeutet, dass die Ressource aktiv Benachrichtigungen an die Anwendung versendet, sobald aktualisierte Daten vorliegen. [CMP21]

Batch-Operationen

Die getrennte Ausführung von Operationen führt zu einem überflüssigen Energieverbrauch, der mit dem Ein- und Ausschalten einer bestimmten Ressource zusammenhängt - dies wird typischerweise als "tail energy consumption" bezeichnet. Indem mehrere Vorgänge in einem Einzigen kombiniert werden, wird der Energieverbrauch im Hintergrund optimiert und so Strom eingespart. [CMP21]

Caching

In vielen Anwendungen werden Daten von einem entfernten Server abgerufen. Während der Lebensdauer einer solchen Anwendung ist es möglich, dass dieselben Daten mehrmals vom Server abgerufen werden. Um die Netzwerkauslastung zu reduzieren, kann Caching zum Einsatz kommen. Dabei wird das Abrufen von Daten vom Server vermieden, indem diese im Cache vorgehalten werden und verwendet werden können. [CMP21]

Unter Verwendung dieser Entwurfsmuster kann der Energieverbrauch eines Software-Produkts reduziert werden. Aber auch Muster, die nicht auf energiebezogene Probleme abzielen, können einen erheblichen Einfluss auf den Energieverbrauch haben. Daher ist es von entscheidender Bedeutung, nicht nur die im System verwendeten Muster zu kennen, sondern auch zu wissen, wie man ihre Vorteile nutzen und gleichzeitig Nachteile für den Gesamtenergieverbrauch des Systems vermeiden kann. [CMP21]

3.3 Vorgehensmodelle

Bei der Beantwortung der Fragestellung, wie ein Softwareprodukt nachhaltiger gestaltet werden kann, müssen auch Vorgehensmodelle in der Softwareentwicklung betrachtet werden. Hierbei ist vor allem zu betrachten, wie Entwicklungsprozesse eine Auswirkung auf die Umwelt haben. Der klassische DevOps-Prozess³ fokussiert sich auf Aspekte wie die Reduzierung der Markteinführungszeit oder die Verringerung von menschlichen Fehlern, wohingegen Aspekte der Nachhaltigkeit und Green IT hier aktuell wenig bis keine Beachtung finden. [Br22]

Hierbei ist in erster Linie auf den Stromverbrauch des Software-Produkts zu achten, aber auch Emissionen, die während des Produktionsprozesses der verwendeten Hardware entstanden sind, sollten betrachtet werden.

Wie aus anderen Bereichen des Software Engineering bekannt ist es deutlich schwieriger und teurer, Änderungen im Anschluss an die Entwicklung oder sogar nach Markteinführung durchzuführen als in frühen Entwicklungsstadien. Deshalb bietet es sich an, Überlegungen zur Nachhaltigkeit so früh wie möglich in den Designprozess der Software zu integrieren. [Di13]

Hierfür ist es nötig, den Lebenszyklus von Software-Produkten zu betrachten. Dieser umfasst nach dem „Life Cycle Thinking“ folgende Phasen: Entwicklung, Vertrieb, Beschaffung, Bereitstellung, Nutzung, Wartung, Deaktivierung und Entsorgung. [DN10]

In der Entwicklungsphase sind Entwicklungsprozesse zu optimieren. Hier müssen Aspekte wie der tägliche Transportweg zur Arbeit, Dienstreisen oder der Energiekonsum von Arbeitsgeräten betrachtet werden. In den Phasen Vertrieb, Beschaffung und Bereitstellung steht vor allem der Transportweg der Software zum Kunden im Mittelpunkt. Auch bei einem reinen Download kann durch die Downloadgröße Netzwerkverkehr minimiert und so Ressourcen eingespart werden. Bisher betrachtete Möglichkeiten wie Softwarearchitektur

³ DevOps steht für Development and Operations und meint, dass ein Team gemeinschaftlich für die Entwicklung, die Qualitätssicherung und Operationen verantwortlich ist.

(siehe Unterabschnitt 3.1) oder Software-Entwurfsmuster (siehe Unterabschnitt 3.2) können vor allem in der Nutzungsphase Ressourcen einsparen. Hier sind Kriterien wie die Prozessor- und Speicherauslastung, der Netzwerkverkehr, aber auch die Update-Häufigkeit und -Größe zu betrachten. In der Wartungs- und Deaktivierungs-Phase ist die Größe der erstellten Backups ein Kriterium. Auch wie die Daten gespeichert werden, spielt eine Rolle. Die letzte Phase Entsorgung behandelt vor allem, was mit gedruckten Anleitungen, Verpackungen und dem Datenmedium geschieht. [DN10] [Jo11]

Eine Möglichkeit, das bestehende Vorgehensmodell zu optimieren, sind Prozess-Assessments. Hierbei werden Daten gesammelt und sich auf Auswirkungen konzentriert, die sich aus dem Softwareprozess selbst ergeben. Die gesammelten Daten müssen dabei umfangreich genug sein, um eine Kohlenstoff-Fußabdruckberechnung oder eine Lebenszyklusanalyse durchzuführen. Diese Aktivität sollte sehr früh begonnen werden, möglicherweise bereits in der Anfangsphase des Produktdesigns. Dabei wird sich vor allem auf die Entwicklungsphase und mögliche Auswirkungen durch diese konzentriert. Gefundene Schwachstellen können verbessert werden, um so bereits während der Entwicklung Ressourcen einzusparen. [Di13]

Indem Administratoren ressourcenschonende Standardkonfigurationen definieren oder die Benutzer zu einem nachhaltigeren Verhalten anleiten, zum Beispiel, indem nach Verlassen des Arbeitsplatzes, dieser in den Ruhezustand versetzt wird, kann während der Entwicklungsphase bereits Energie der Arbeitsgeräte eingespart werden. [Jo11]

Zusätzlich können Sustainability Reviews und Previews durchgeführt werden. Diese fokussieren sich auf die Vertriebs-, Beschaffungs-, Bereitstellungs- und Nutzungs-Phase im Vordergrund. Durch verschiedene Werkzeuge und Methoden wie Code-Reviews, Laufzeiteffizienz- und Leistungsmessungen, Energieeffizienzmessungen oder andere Metriken soll hierbei die geleistete Arbeit in Bezug auf Nachhaltigkeitsthemen überprüft werden. Indem Sustainability Reviews und Previews nach etwa zwei Dritteln einer Iteration durchgeführt werden, ist es dem Entwicklungsteam möglich, gefundene Fehler noch innerhalb derselben Iteration zu korrigieren. [Di13]

Erkenntnisse aus Sustainability Reviews und Previews, die auch für die Anwender relevant sein könnten, können diesen zur Verfügung gestellt werden. So werden sie in die Lage versetzt, die Auswirkungen auf die Nachhaltigkeit durch ihr Nutzungsverhalten zu optimieren. [Jo11]

Auch Retrospektiven zur Nachhaltigkeit sind möglich. Durch diese soll nicht das aktuelle Projekt, sondern die Nachhaltigkeit künftiger Softwareprojekte verbessert werden. Eine solche Retrospektive kann nach Abschluss des Projekts durchgeführt werden, wobei Ergebnisse in Form von gelernten Lektionen, Best Practices, Entscheidungen für zukünftige Projekte etc. dokumentiert werden. [Di13]

Diese Methoden müssen nun in das bestehende Entwicklungsvorgehen integriert werden. In vielen modernen Softwareprojekten wird Scrum als Vorgehensmodell verwendet. Das Ziel von Scrum ist es dabei, in kurzen Iterationen zu arbeiten und in jeder dieser Iterationen ein potenziell auslieferungsfähiges Inkrement der Software zu erstellen. Die bereits in Scrum

vorhandenen Bestandteile wie das Sprint Planning, Daily Scrums, Sprint Reviews und die Sprint Retrospective bleiben erhalten und werden durch die beschriebenen Möglichkeiten zur Optimierung der Nachhaltigkeit nicht beeinflusst. Das Prozess-Assessment ist dabei eine eigene Aufgabe am Anfang des Projekts, um den aktuellen Stand zu evaluieren. Sustainability Reviews und Previews werden in jedem Sprint nach etwa zwei Dritteln von diesen durchgeführt. Die Retrospektive zur Nachhaltigkeit findet einmalig kurz vor dem Ende des Projekts, allerdings noch vor dem letzten Sprint Review statt, um hier das Entwicklungsteam in die Lage zu versetzen, die kombinierten Bewertungsergebnisse an den Product Owner zu berichten. [Di13]

Um diese Vorgehensmodelle umsetzen zu können, muss das Entwicklungsteam durch bestimmte Werkzeuge unterstützt werden. Ein heutzutage viel verwendetes Konzept ist Continuous Integration. Indem die bestehende Continuous Integration Umgebung um eine automatisierte Messung der Energieeffizienz erweitert wird, erhält das Entwicklungsteam zu jedem Zeitpunkt den aktuellen Status hinsichtlich der Energieeffizienz ihres Software-Produkts. So wird es möglich, getätigte Änderungen auch im Hinblick auf Nachhaltigkeit zu bewerten. Dabei ist jedoch zu beachten, dass bereits vorhandene Tests in der Regel ungeeignet für eine Messung der Energieeffizienz sind. Für diese sind Tests nötig, die eine erhebliche Last über einen längeren Zeitraum erzeugen. Das Entwicklungsteam muss dafür optimierte Tests also vorab erstellen. [Di13]

4 Gewährleistung, dass Software nachhaltig bleibt

Durch die in im letzten Kapitel dargestellten Möglichkeiten wurden messbare Werte, wie z.B. der durch Anwendungen verbrauchte Strom idealerweise verringert. Zudem können aufwändigen Berechnungen in ein Rechenzentrum mit besserer Kohlenstoffkonzentration des Stroms verlegt werden.

Als Ziel sollte jedoch auch gelten, dass diese ressourcenschonenden Entwicklungen zukünftig nicht wieder umgekehrt werden. Es sind auch in Zukunft weitere Messungen hinsichtlich des Strombedarfs nötig, um auch nach Erweiterungen einer Anwendung nicht unverhältnismäßig mehr Energie zu verbrauchen. Es muss zudem darauf geachtet werden, dass die programmierte Anwendung auch in Zukunft nutzbar bleibt. Als Faustregel kann dabei auf die Kernfunktionalitäten geachtet werden: Diese sollten auch auf älterer Hardware verfügbar und nutzbar bleiben. [Ve21] Man sollte, wie im Abschnitt 2 Green IT schon genauer beschrieben, nicht durch eine neue Version seine alte Hardware entsorgen müssen, wodurch wieder Kohlenstoff freigesetzt werden würde.

Im Abschnitt 3 Umsetzungsmöglichkeiten wurden Regeln in Bezug auf die Architektur der Software und der Verwendung von bestimmten Entwurfsmuster und Vorgehensmodellen erläutert. Wird sich an diesen orientiert und Muster und Modelle mit Bedacht gewählt, ist dies bereits ein erster Schritt zu einem Softwareprodukt, das auch in Zukunft energiesparend arbeitet.

Auch Label wie der Blaue Engel können die Einstufung einer Applikation als „grün“ gewährleisten, da diese anhand ihrer Vergabekriterien eine umweltfreundliche Software auszeichnen. Werden diese Kriterien nicht mehr erfüllt, ist auch für die Verbraucher ersichtlich, dass die Anwendung nicht länger ausreichend energie- und ressourcensparend ist. [Bl20]

5 Bewertung

In den vorhergehenden Teilen wurden konkrete Umsetzungsmöglichkeiten vorgestellt, die das Software-Produkt nachhaltiger machen sollen. Ob diese Maßnahmen tatsächlich dazu führen, dass Software-Produkte effizienter werden und damit Ressourcen einsparen, lässt sich nicht allgemein belegen. Ohne diese Maßnahmen auszuprobieren und hier Messungen durchzuführen wird diese Frage jedoch weiterhin offen bleiben. [CMP21]

Auch wurden in dieser Arbeit lediglich Umsetzungsmöglichkeiten auf Software-Seite betrachtet. Eine vollumfängliche Green-IT-Strategie eines Unternehmens muss zwingend auch Überlegungen und Maßnahmen zur Verwendung von Hardware treffen. Im Rahmen dieser Arbeit wird dies mit Fokus auf das Thema gänzlich außen vorgelassen.

Generell birgt Green IT sowohl Vorteile als auch Nachteile. Der wohl größte Vorteil ist die Einsparung von Energie. Dies beeinflusst mehrere weitere, nicht zu vernachlässigende Faktoren. Zum einen hat ein geringerer Energieverbrauch der Software-Produkte eines Unternehmens durch Green IT zur Folge, dass die Umweltbelastung (siehe Abschnitt 2 Green IT) durch dieses Unternehmen reduziert wird.

Zum anderen sehen sich Unternehmen heutzutage mit deutlich erhöhten Energiekosten konfrontiert. Zusätzlich müssen diese ggf. mit weiteren staatlichen Abgaben rechnen, sofern sie die Umweltauswirkungen ihrer Produkte nicht berücksichtigen. Auch einige Investoren und Verbraucher haben bereits damit begonnen, beim Kauf von Produkten auf die Nachhaltigkeit der Unternehmen zu achten oder die Aktienkurse von Unternehmen abzuwerten, die die von ihnen verursachten Umweltprobleme nur unzureichend angehen. Die Menschen haben begonnen, umweltfreundliche Eigenschaften auch im Bereich der Software zu schätzen. [Sa08] Durch den Fokus auf Green IT können also sowohl direkte Kosten in Form von Energie eingespart werden, als auch Verbraucher und Investoren dazu animiert werden, sich für das Unternehmen zu entscheiden.

Da bereits eine Vielzahl an Green-IT-Kennzahlen und einige Ansätze für Kennzahlensysteme vorhanden sind, wird die Bestimmung von Ansatzpunkten für Green IT vereinfacht.

Auf der anderen Seite ist jedoch eine hohe Fachkompetenz erforderlich, um die Kennzahlen zu analysieren und geeignete Maßnahmen für Green IT abzuleiten. Aus wirtschaftlicher Sicht ist es deshalb nicht sicher, ob die Integration von Green-IT-Maßnahmen sinnvoll ist, da durch die Bestimmung und Integration geeigneter Maßnahmen ggf. höhere Kosten entstehen könnten, als sie durch den Nutzen dieser Maßnahmen eingespart werden.

Da in einem gewinnorientierten Unternehmen hauptsächlich monetäre Aspekte im Vordergrund stehen, wird der Nutzen von Green-IT-Maßnahmen oft übersehen und sich dagegen entschieden. Dabei spielen sowohl die Kosten der Integration von Maßnahmen, aber auch ggf. auftretende Leistungs-Einbußen der Software sowie Wartungskosten eine Rolle. Auch Greenwashing ist ein potenzieller Nachteil von Green IT. Einige Unternehmen haben das Potenzial erkannt, durch vermeintliche Umweltschutzkampagnen einen höheren Kundenkreis für sich gewinnen oder Investoren zu einer Investition überzeugen zu können. Greenwashing meint dabei den Versuch von Unternehmen durch Kommunikation, Marketing und Einzelmaßnahmen ein „grünes Image“ zu erlangen, ohne entsprechende Maßnahmen im operativen Geschäft systematisch verankert zu haben. [Li]

Insgesamt ist Green IT jedoch ein guter und mit Blick auf den menschengemachten Klimawandel wichtiges Konzept, das jedes Unternehmen zeitnah integrieren sollte. Dabei kann auch die Seite der Software geprüft und entsprechende Maßnahmen für diese abgeleitet werden.

6 Ausblick

Mittels der Anwendung von Konzepten der Green IT kann ein Unternehmen nicht nur mehr zum Umweltschutz generell beitragen, sondern auch durch den Betrieb von Software entstehende Kosten reduzieren. Neben hardwareseitigen Umsetzungsmöglichkeiten kann durch die Verwendung bestimmter Entwurfsmuster, die Wahl einer ressourcensparenden Architektur und des Etablieren bestimmter Vorgehensmodelle auch seitens der Software selbst dazu beigetragen werden, dass diese nachhaltiger wird.

Unter Beachtung der Vorteile von Green IT ist zu erwarten, dass zukünftig noch deutlich mehr Unternehmen dieses Thema fokussieren werden. Jedes Unternehmen muss eine ganzheitliche, umfassende Green-IT-Strategie entwickeln. Anschließend sollte es eine Green-IT-Politik entwickeln, die Ziele, Aktions- und Zeitpläne enthält. Große Unternehmen sollten auch einen Beauftragten für ökologische Nachhaltigkeit ernennen, der ihre grüne Strategie umsetzt und ihre Fortschritte und Erfolge überwacht. [Sa08]

Auch die Forschung wird sich durch die große Bedeutung in der Industrie künftig verstärkt mit Green IT und zugehörigen Maßnahmen auseinandersetzen.

Literatur

- [Ac22] Achim Guldner; Eva Kern; Sandro Kreten; Stefan Naumann: Software und Nachhaltigkeit – Wie passt das zusammen?, 2022, URL: <https://www.informatik-aktuell.de/management-und-recht/digitalisierung/software-und-nachhaltigkeit-wie-passt-das-zusammen.html>, Stand: 10.12.2022.

- [Ba11] Baliga, J.; Ayre, R. W. A.; Hinton, K.; Tucker, R. S.: Green Cloud Computing: Balancing Energy in Processing, Storage, and Transport. Proceedings of the IEEE 99/1, S. 149–167, 2011, issn: 0018-9219.
- [Bl20] Blauer Engel: Blauer Engel, Ressourcen- und energieeffiziente Softwareprodukte - Vergabekriterien, DE-UZ 215, hrsg. von Blauer Engel, 2020, URL: <https://www.blauer-engel.de/de/produktwelt/ressourcen-und-energieeffiziente-softwareprodukte>.
- [Br22] Brode, B.: 5 Steps to More Sustainable DevOps, 2022, URL: <https://devops.com/5-steps-to-more-sustainable-devops/>, Stand: 10.12.2022.
- [Bu20] Bundesumweltministerium: Green IT, Copyright: Internetseite des Bundesumweltministeriums - BMUV, 30.06.2020, URL: <https://www.bmuv.de/themen/nachhaltigkeit-digitalisierung/konsum-und-produkte/produktbereiche/green-it>, Stand: 10.12.2022.
- [Bu22] Bundeszentrale für politische Bildung: Ursachen und Folgen des Klimawandels. Bundeszentrale für politische Bildung/, 1. Mai 2022, URL: <https://www.bpb.de/shop/zeitschriften/izpb/klima-347/336195/ursachen-und-folgen-des-klimawandels/>, Stand: 10.12.2022.
- [CEA11] C. Kelly; E. Mangina; A. Ruzelli: Putting a CO₂ figure on a piece of computation. In: 11th International Conference on Electrical Power Quality and Utilisation. 11th International Conference on Electrical Power Quality and Utilisation. S. 1–7, 2011, isbn: 2150-6655.
- [CMP21] Calero, C.; Moraga, M. Á.; Piattini, M., Hrsg.: Software Sustainability. eng, Calero, Coral (HerausgeberIn) Moraga, Ma Ángeles (HerausgeberIn) Piattini, Mario (HerausgeberIn), Cham: Springer, 2021, 411 S., isbn: 9783030699703, URL: <https://ebookcentral.proquest.com/lib/kxp/detail.action?docID=6745700>.
- [CS13] Chauhan, N. S.; Saxena, A.: A Green Software Development Life Cycle for Cloud Computing. IT Professional 15/1, S. 28–34, 2013, issn: 1520-9202.
- [D 09] D. Kondo; B. Javadi; P. Malecot; F. Cappello; D. P. Anderson: Cost-benefit analysis of Cloud Computing versus desktop grids. In: 2009 IEEE International Symposium on Parallel & Distributed Processing. 2009 IEEE International Symposium on Parallel & Distributed Processing. S. 1–12, 2009, isbn: 1530-2075.
- [Di13] Dick, M.; Drangmeister, J.; Kern, E.; Naumann, S.: Green software engineering with agile methods. In (Lago, P., Hrsg.): Proceedings of the 2nd International Workshop on Green and Sustainable Software. ACM Digital Library, Association for Computing Machinery-Digital Library und ACM Special Interest Group on Software Engineering, IEEE Press, Piscataway, NJ, 2013, isbn: 9781467362672, URL: <https://dl.acm.org/doi/10.5555/2662693.2662707>, Stand: 23.10.2022.

- [DN10] Dick, M.; Naumann, S.: Enhancing Software Engineering Processes towards Sustainable Software Product Design, Umwelt-Campus Birkenfeld, Trier: Trier University of Applied Sciences, 2010.
- [Ei] Eisele, M.: 5 ways to build sustainability into your software architecture. Red Hat, Inc/, URL: <https://www.redhat.com/architect/sustainable-software-architecture>.
- [Go22] Google Cloud: Carbon Footprint | Google Cloud, 11/8/2022, URL: <https://cloud.google.com/carbon-footprint>, Stand: 30. 11. 2022.
- [Hu21] Hussain, A.: Principles of Green Software Engineering, 8/22/2021, URL: <https://principles.green/>, Stand: 19. 11. 2022.
- [Jo11] Johann, T.; Dick, M.; Kern, E.; Naumann, S.: Sustainable development, sustainable software, and sustainable software engineering: An integrated approach. In: 2011 International Symposium on Humanities, Science and Engineering Research. 2011 International Symposium on Humanities, Science and Engineering Research. S. 34–39, 2011, ISBN: 2378-9816, Stand: 23. 10. 2022.
- [Ke18] Kern, E.; Hilty, L. M.; Guldner, A.; Maksimov, Y. V.; Filler, A.; Gröger, J.; Naumann, S.: Sustainable software products—Towards assessment criteria for resource and energy efficiency. Future Generation Computer Systems 86/, PII: S0167739X17314188, S. 199–210, 2018, ISSN: 0167739X.
- [Li] Lin-Hi, P. D. N.: Definition: Greenwashing. Springer Fachmedien Wiesbaden GmbH/, URL: <https://wirtschaftslexikon.gabler.de/definition/greenwashing-51592/version-384777>, Stand: 27. 11. 2022.
- [LS] Lackes, R.; Siepermann, M.: Definition: Green IT. Springer Fachmedien Wiesbaden GmbH/, URL: <https://wirtschaftslexikon.gabler.de/definition/green-it-53166/version-276261>, Stand: 10. 12. 2022.
- [Me22] Mehra, R.; Sharma, V. S.; Kaulgud, V.; Podder, S.; Burden, A. P.: Towards a green quotient for software projects. In: 2022 IEEE/ACM 44th International Conference on Software Engineering: Software Engineering in Practice (ICSE-SEIP). 2022 IEEE/ACM 44th International Conference on Software Engineering: Software Engineering in Practice (ICSE-SEIP). S. 295–296, 2022.
- [Po20] Podder, S.; Burden, A.; Kumar Singh, S.; Maruca, R.: How Green Is Your Software?, hrsg. von Harvard Business Review, 2020, URL: <https://hbr.org/2020/09/how-green-is-your-software>, Stand: 10. 12. 2022.
- [Sa08] San Murugesan: Harnessing Green IT: Principles and Practices. IT Professional 10/, S. 26, 2008, URL: <https://ieeexplore.ieee.org/abstract/document/4446673>, Stand: 10. 12. 2022.

- [Sa12] Sahin, C.; Cayci, F.; Gutierrez, I. L. M.; Clause, J.; Kiamilev, F.; Pollock, L.; Winbladh, K.: Initial explorations on design pattern energy usage. In (Kazman, R., Hrsg.): 2012 First International Workshop on Green and Sustainable Software (GREENS 2012), Zurich, Switzerland, 3 June 2012 ; [part of the 34th International Conference on Software Engineering (ICSE). 2012 First International Workshop on Green and Sustainable Software (GREENS), Zurich, Switzerland. IEEE, Piscataway, NJ, S. 55–61, 2012, ISBN: 978-1-4673-1832-7, URL: <https://ieeexplore.ieee.org/document/6224257>.
- [TC21] Talbott, C.; Conkling, J.: New tools to measure and reduce your environmental impact, hrsg. von Google Cloud Blog, 2021, URL: <https://cloud.google.com/blog/topics/sustainability/new-tools-to-measure-and-reduce-your-environmental-impact>, Stand: 10.12.2022.
- [Um22] Umweltbundesamt: Emissionsquellen, 10. Dez. 2022, URL: <https://www.umweltbundesamt.de/themen/klima-energie/treibhausgas-emissionen/emissionsquellen#energie-stationar>, Stand: 10.12.2022.
- [Ve21] Verdecchia, R.; Lago, P.; Ebert, C.; de Vries, C.: Green IT and Green Software. IEEE Software 38/6, S. 7–15, 2021, ISSN: 0740-7459.
- [Wi22] Wikipedia, Hrsg.: Cloud Computing, de, Creative Commons Attribution-ShareAlike License Page Version ID: 228091000, 2022, URL: https://de.wikipedia.org/w/index.php?title=Cloud_Computing&oldid=228091000.
- [Wo] World Meteorological Organization: Four key climate change indicators break records in 2021, URL: <https://public.wmo.int/en/media/press-release/four-key-climate-change-indicators-break-records-2021>, Stand: 10.12.2022.
- [ZD22] ZDF: Virtuelles Schürfen immer klimaschädlicher. Digitales/, 10. Feb. 2022, URL: <https://www.zdf.de/nachrichten/digitales/bitcoins-schuerfen-kimaschaedlich-100.html>, Stand: 11.12.2022.

Autorenverzeichnis

A

Arnold, Niklas, 229

B

Bauckhage, Ingmar, 11

D

Dietrich, Alicia, 185

Dürr, Jannik, 151

Dürr, Nick, 185

E

Epple, Lukas, 61

Epple, Robin, 249

F

Freudenberger, Jülf, 37

H

Heinl, Fabian, 267

J

Joas, Dominic, 151

Jooß, Reinhold, 91

K

Kalmbach, Ruben, 151

Klimpel, Fabian, 61

Kremling, Philipp, 267

L

Liehner, Adrian, 37

N

Negrón-Martínez, Luca, 229

P

Perthel, Jan, 185

Pypenko, Vsevolod, 11

S

Sack, Raphael, 61

Schwab, Jonathan, 117

Sperling, Gabriel, 91

V

Vollert, Timo, 249

W

Weis, Jonas, 117

Wochele, Felix, 117