

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

## Workshops

### Kongresstag 1

<b>Adrian Liehner, Jülf Freudenberger</b>	
<i>Software Reuse</i> . . . . .	11
<b>Fabian Klimpel, Lukas Epple, Raphael Sack</b>	
<i>Auswirkungen von SOA</i> . . . . .	35
<b>Gabriel Sperling, Reinhold Jooß</b>	
<i>Infrastructure as Code</i> . . . . .	65

### Kongresstag 2

<b>Jonathan Schwab, Felix Wochele, Jonas Weis</b>	
<i>Intelligente Werkzeuge SWE</i> . . . . .	91
<b>Jannik Dürr, Dominic Joas, Ruben Kalmbach</b>	
<i>Hybride Applikationen</i> . . . . .	125
<b>Robin Epple, Timo Vollert</b>	
<i>TITLE</i> . . . . .	159

## Autorenverzeichnis



# Workshops





# Kongresstag 1



# Software Reuse: Überblick und Anwendung

Adrian Liehner,<sup>1</sup> Jülf Freudenberger<sup>2</sup>

**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]

---

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

<sup>2</sup> 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 Softwareprodukte 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 [Wal11]. 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 Repositories, 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 Onlinere Ressourcen 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, das 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, das 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 Performanceeinbußen 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 verschiedenen 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 Punkte 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 wiederverwendbare 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 Verfehlen 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 Budget, 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 Architekturmuster 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 Datenflusssystemen 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 unabhängigere 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 passenden 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 Verhalten 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 Aspekten 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 Quelldateien** 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 Quelldatei verteilt sein.

**Kontrollvariablen.** Auch wenn es möglich ist Kontrollvariablen von Schleifen auch außerhalb der Schleife zu deklarieren, 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 beieinanderstehen 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-Repositories, 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 Micro-services, 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 Klimpel<sup>1</sup>, Lukas Epple<sup>2</sup>, Raphael Sack<sup>3</sup>

**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

---

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

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

<sup>3</sup> 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 als 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 Dienstsyste men. 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 unumgebar.

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 werden<sup>4</sup>:

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.

<sup>4</sup> 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 anzuordnen - 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 Regeln 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.

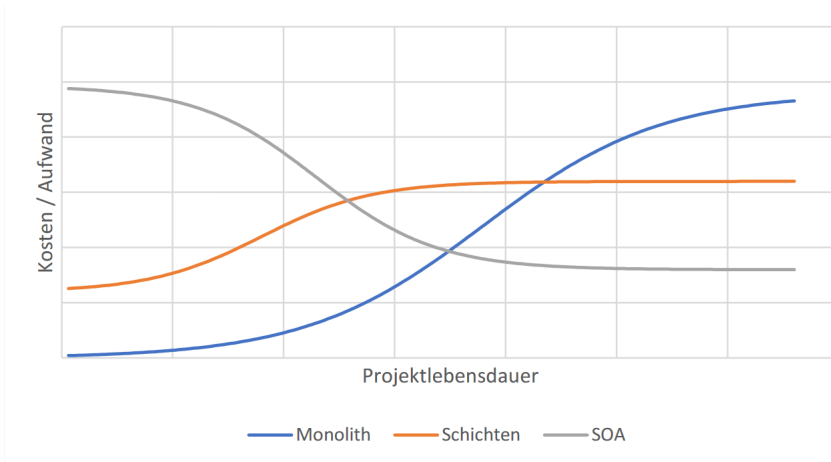


Abb. 1: Relativer Kosten-/Aufwand-Vergleich der vorgestellten Architekturen

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.

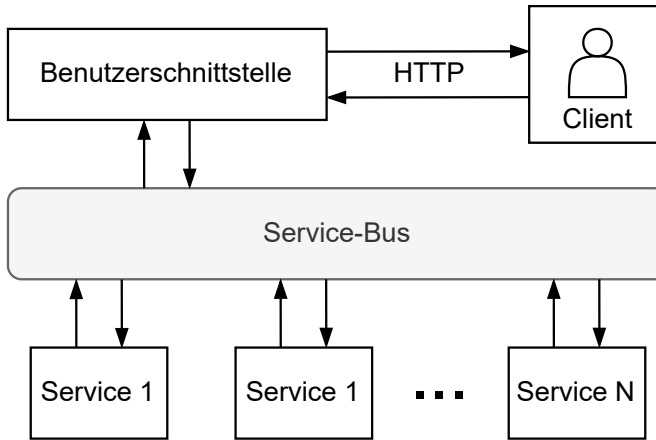


Abb. 2: SOA Aufbau

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.

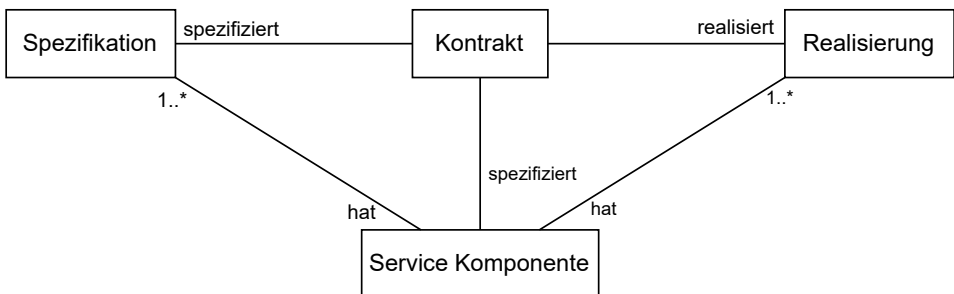


Abb. 3: Spezifikation und Realisation einer Service-Komponente [SDS04]

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]

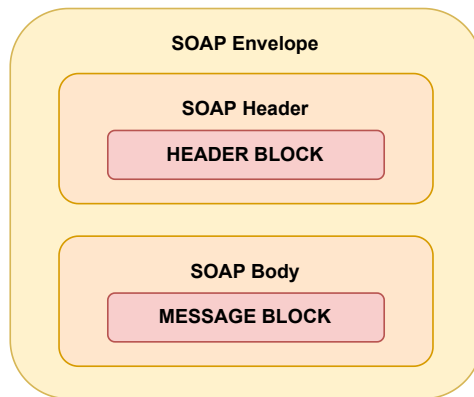


Abb. 4: SOAP Aufbau [Gi22]

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 befassen 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 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]

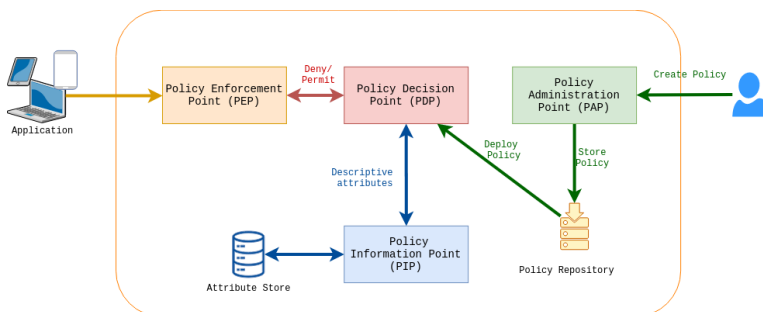


Abb. 5: XACML Flow-Diagramm

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

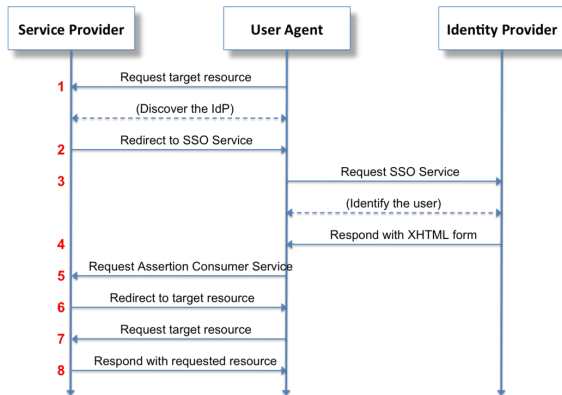


Abb. 6: SAML Sequenz-Diagramm

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 erhalten. 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 korrumpieren 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.

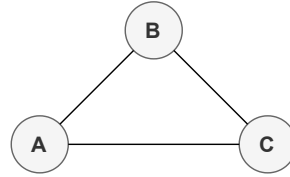
Vertikale Skalierung beschreibt in diesem Kontext die Erhöhung beziehungsweise Reduktion der verfügbaren Ressourcen eines Services. Dies kann beispielsweise bedeuten, eine Serviceinstanz durch eine andere zu ersetzen, die auf einem leistungsfähigeren Host läuft oder die zugeteilten Ressourcen der VM,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

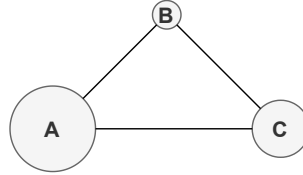


Belastung der einzelnen Services		
A	B	C
mittel	mittel	mittel



### 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

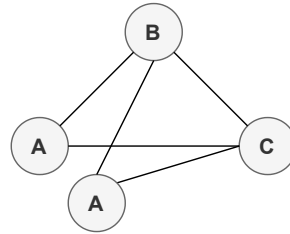


Abb. 7: Darstellung der beispielhaften Skalierungsmöglichkeiten eines Systems bestehend aus drei Services A, B und C, entsprechend bestimmter Lastszenarien

## 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 geklusterten 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.

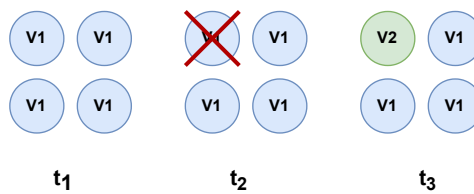


Abb. 8: Ablauf einer rollenden Aktualisierung

### **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.

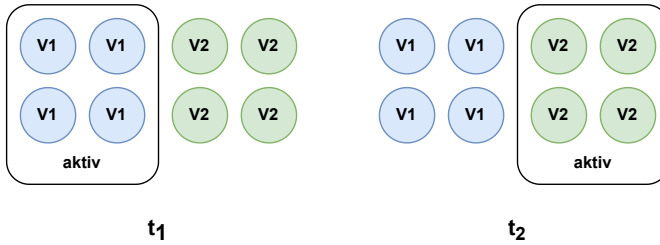


Abb. 9: Ablauf einer Blue-Green Aktualisierung

### 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.

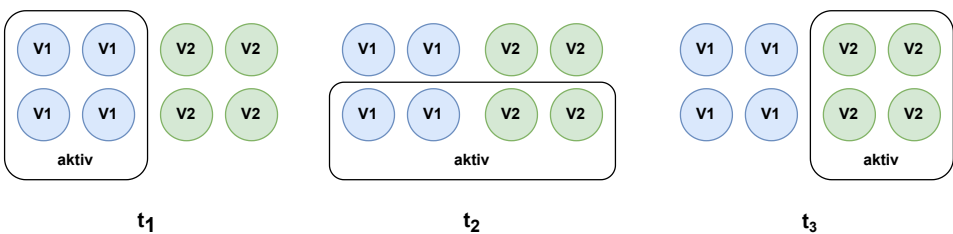


Abb. 10: Ablauf einer Canary Aktualisierung

### 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 Implementierungstechnologie 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. Darüber 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](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 Sperling,<sup>1</sup> Reinhold Jooß<sup>2</sup>

**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

---

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

<sup>2</sup> 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 kurzschrittige 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 beliebige 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.

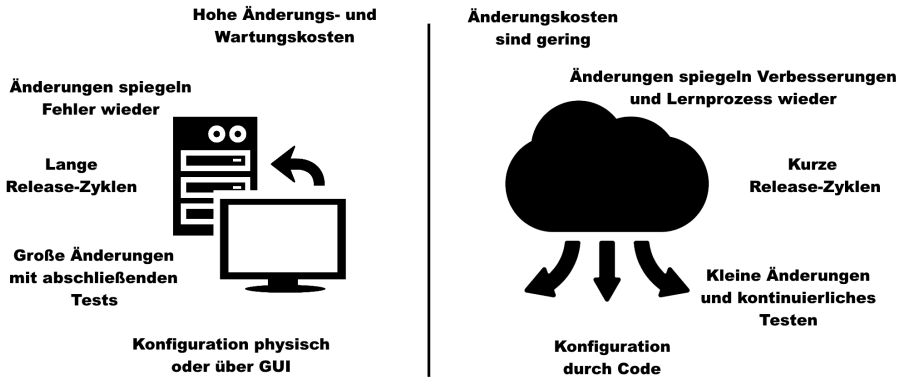


Abb. 1: Manuelle Konfiguration und Konfiguration in der Cloud Age im Vergleich [Mo20]

## 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 beachtend 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 unvorhergesehen 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 Infrastrukturteilen. 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 neuzuformen, 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 Ausführung 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 offengelegt und komplexere Strukturen werden ermöglicht.

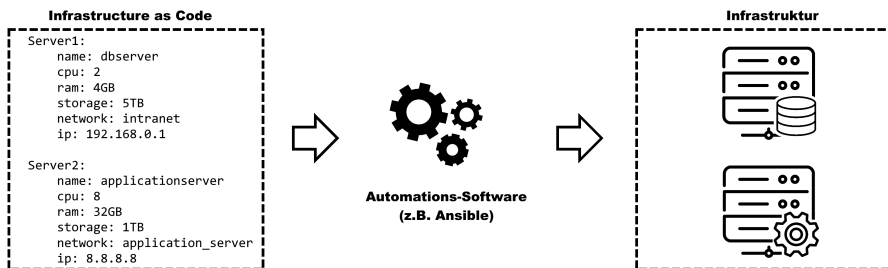


Abb. 2: IaC zur automatisierten Erstellung der Infrastruktur

### 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 Skripten 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 gewissen 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 Inkompatibilitäten 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 bereitstellen 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 Konfigurationen 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 Kontrollsystem 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



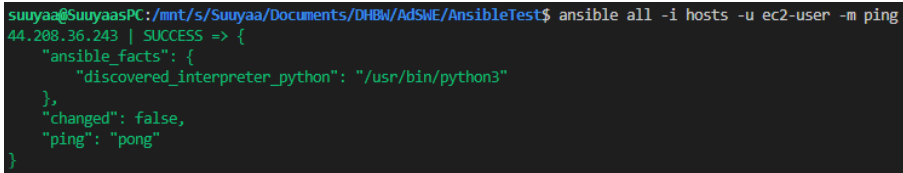
werden, und eine SSH-Verbindung zum Zielsystem muss aufgebaut werden können. Ansible wird ebenfalls durch eine aktive Community unterstützt, die viele Erweiterungen anbietet, um weitere Services anzuwenden und mehr Use Cases ermöglichen zu können.

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 Machines 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.



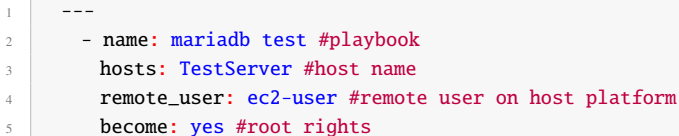
```

suuyaa@SuuyaaPC:/mnt/s/Suuyaa/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
    it will be executed first
12 # path: // Same directory, so no path is required
13 - name: installing mariadb
14   yum: name=mariadb-server state=latest
15
16 - name: create mysql configuration file
17   template: src=my.cnf.j2 dest=/etc/my.cnf
18   notify: restart mariadb #triggers at end of task and only once (multiple
    notifies possible)
19
20 - name: create mariadb log file
21   file: path=/var/log/mysqld.log state=touch owner=mysql group=mysql mode=0775
22
23 - name: start mariadb service
24   service: name=mariadb state=started enabled=yes
25
26 handlers: #handlers for notifies
27 - name: restart mariadb
28   service: name=mariadb state=restarted
29 #-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 Blocks 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 #doesn't 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.; Borovssak, 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](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.



## Kongresstag 2



# Einsatz intelligenter Werkzeuge zur Softwareentwicklung

Jonathan Schwab<sup>1</sup>, Felix Wochele<sup>2</sup>, Jonas Weis<sup>3</sup>

**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 Zeitersparnis, eine Steigerung der Codequalität oder eine bessere Nachverfolgbarkeit sein. Dies steigert die Wirtschaftlichkeit von Softwareprojekten 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.

---

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

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

<sup>3</sup> 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 Codeervollständigung

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 Codeervollständigung in Integrierten Entwicklungsumgebungen, also die Unterstützung während diesem Vorgang.

### 2.1 Allgemeine Konzepte

Automatische Codeervollständigung 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.

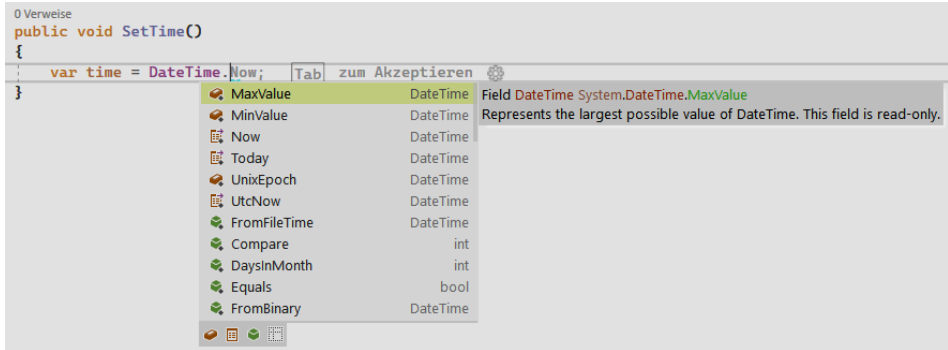


Abb. 1: Code Vervollständigung in Visual Studio 2022

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 geparkt 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].

Naheliegender 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. Gleiches 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 Rückgabetyt. 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. Bezüglich 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. Bezüglich 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.



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 Methodenname 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-Repositories, für welche Qualität sichergestellt wurde. Neue Daten finden regelmäßigeinsatz 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.

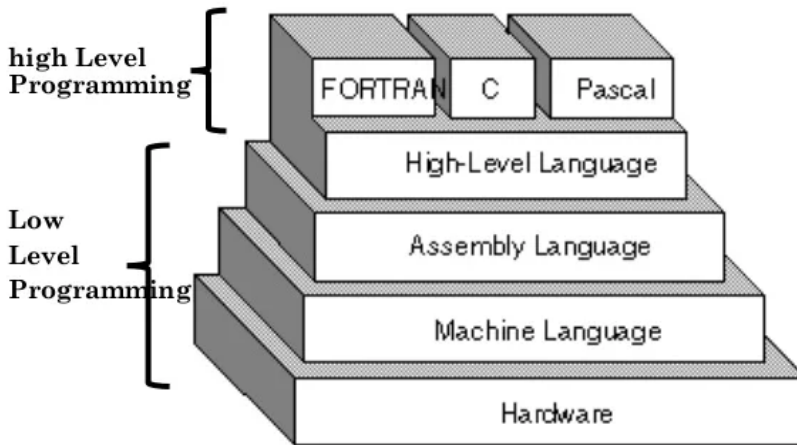


Abb. 3: Abstraktionsebenen von Programmiersprachen [Th21]

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 zählt 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 Komplettlösungen 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
2  /* Generiere eine zufaellige Zahl zwischen 0 und 10 */
3  var randomNumber = Math.floor(Math.random() * 10);
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 Zeitersparnis 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 Programmlaufes 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].

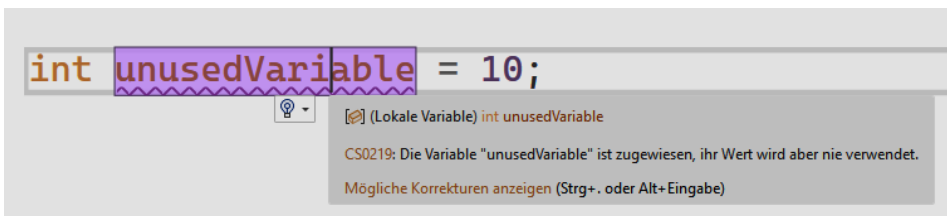


Abb. 4: Statische Codeanalyse in Visual Studio

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 Metriken 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. Intelligente Analysewerkzeuge bieten zudem die Möglichkeit einer automatischen Problembehebung. Hier wird der Quellcode automatisch refaktorisert (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 unsauberen 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 Framework-spezifischen 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 Referenzerzeugung 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 *Refaktorisierung*, 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 Ausdrucks, 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 weist 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. geparkt oder der Datentyp des Felder verändert werden. Software kann diese Lösungen automatisch vornehmen.
- (e) **Einführung von Feldern oder Variablen:** Oftmals findet eine hohe Verschachtelung 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 performanterer (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 refaktorisieren 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 Basics, 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 Refaktorisierung 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 vorgeschlagen 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].

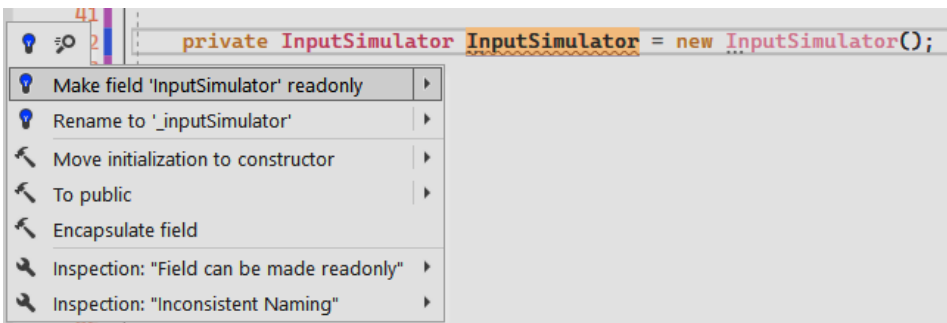


Abb. 5: Refactoring Vorschläge von ReSharper

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.*

—opensourceurvey

Eine großangelegte Umfrage von Open Source<sup>4</sup> 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

---

<sup>4</sup> 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 Arte 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 Kommentare 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 auflisten 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].

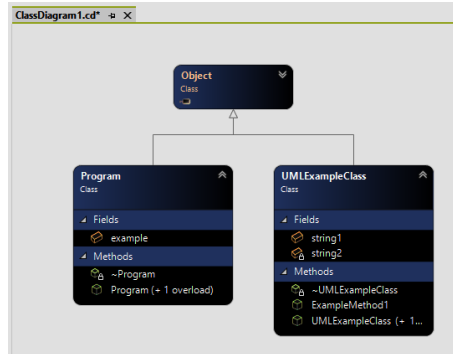


Abb. 6: Ein von Visual Studio generiertes UML-Diagramm

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 IntelliJ 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.

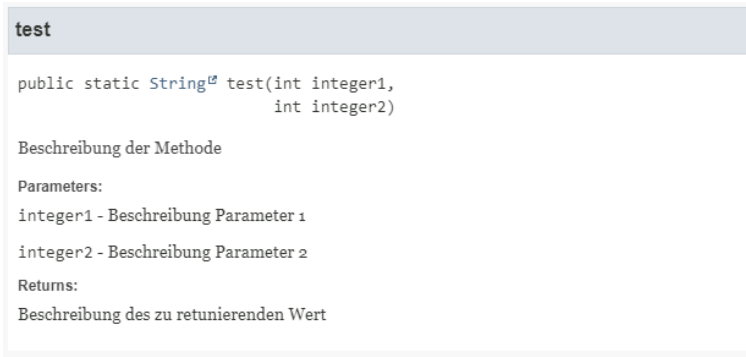


Abb. 7: Automatisch generierte Übersicht einer Java-Methode Quellcode 6

Dennoch gibt es auch in diesem Feld weitere 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 Programmierungsumgebung 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 Java<sup>5</sup>. 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.

<sup>5</sup> 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 Marktanalyse

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 dass 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 Foundation 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 GitHub<sup>6</sup> 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

---

<sup>6</sup> <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 Erkenntnisse 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 Codevervollständigung, 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.; Tillet, 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%7Bhttps://www.doxygen.nl/index.html%7D](https://www.doxygen.nl/index.html), 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/](https://opensourcesurvey.org/2017/), Stand: 02. 11. 2022.
- [GKF06] How are java software developers using the eclipse ide?, IEEE, 2006, S. 76–83.
- [gm12] gmatht: 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-cd-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](https://www.jetbrains.com/help/idea/class-diagram.html), 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](https://learn.microsoft.com/en-us/previous-versions/visualstudio/visual-studio-2015/ide/using-the-task-list?view=vs-2015&redirectedfrom=MSDN#tokenscomments), 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/](https://www.kite.com/), 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/idea/resharper/features/quick\\_fixes.html](https://www.jetbrains.com/idea/resharper/features/quick_fixes.html), Stand: 15. 11. 2022.
- [Re22b] ReShaper, J.: Refactorings, 2022, URL: [https://www.jetbrains.com/help/resharper/Refactorings\\_\\_Index.html](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](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: <https://learn.microsoft.com/en-us/visualstudio/ide/class-designer/designing-and-viewing-classes-and-types?view=vs-2022%D>, 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 Dürr,<sup>1</sup> Dominic Joas,<sup>2</sup> Ruben Kalmbach<sup>3</sup>

**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.

---

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

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

<sup>3</sup> 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 Literaturlarbeit 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.

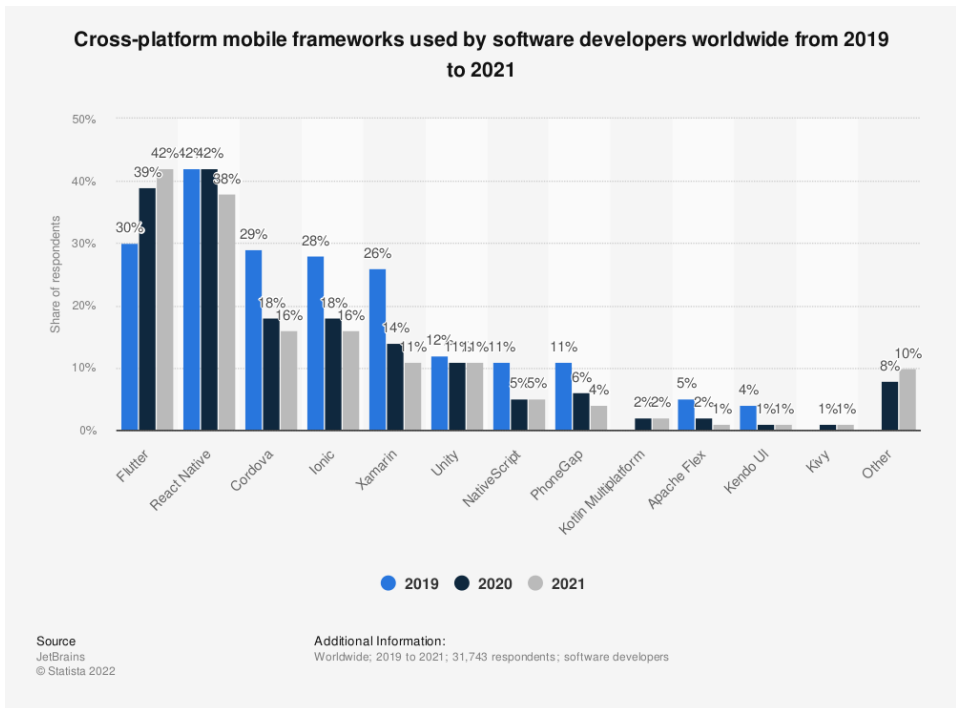


Abb. 1: Umfrage, welche hybriden Frameworks von Entwicklern verwendet werden [Statista 2021]

### 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 zwangsweise 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.

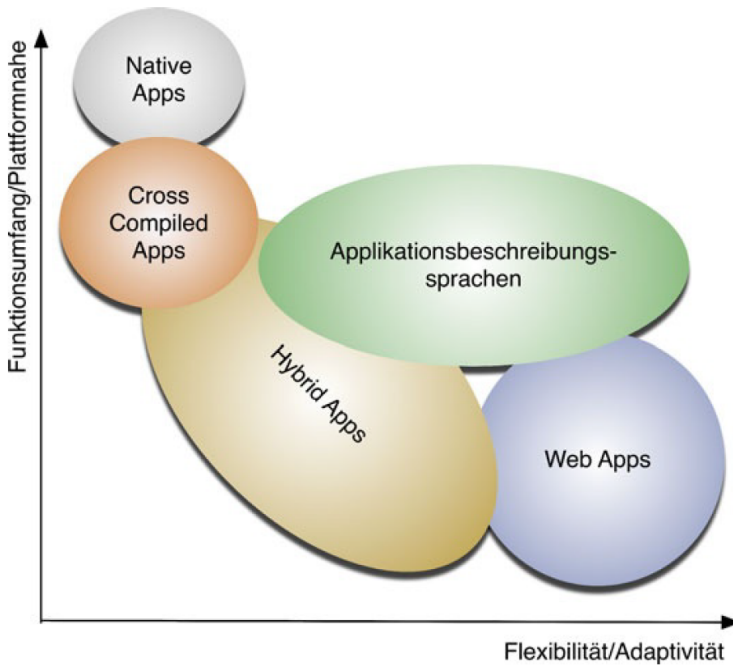


Abb. 2: Übersicht über Arten von Applikationen nach Flexibilität und Funktionsumfang [Willnecker et al. 2012, S.405]

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 [Coward 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 erschwerte 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

**Lizenz & 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

als wichtiger									Summe	%
	Plattformspezifische Funktionen	Performance	Benutzeroberfläche	Erste Schritte	Entwicklungsunterstützung	Dokumentation	Ökosystem	Lebenszyklus & öffentliches Interesse		
Plattformspezifische Funktionen		0	0	1	0	0	1	0	2	7%
Performance	1		0	1	0	1	1	0	4	14%
Benutzeroberfläche	1	1		1	1	1	1	1	7	25%
Erste Schritte	0	0	0		1	0	0	0	1	4%
Entwicklungsunterstützung	1	1	0	0		1	1	0	4	14%
Dokumentation	1	0	0	1	0		1	0	3	11%
Ökosystem	0	0	0	1	0	0		0	1	4%
Lebenszyklus & öffentliches Interesse	1	1	0	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

als wichtiger									Summe	%
	Plattformspezifische Funktionen	Performance	Benutzeroberfläche	Erste Schritte	Entwicklungsunterstützung	Dokumentation	Ökosystem	Lebenszyklus & öffentliches Interesse		
Plattformspezifische Funktionen		1	1	1	1	1	1	1	7	25%
Performance	0		1	1	0	0	1	0	3	11%
Benutzeroberfläche	0	0		0	0	0	0	0	0	0%
Erste Schritte	0	0	1		0	0	0	0	1	4%
Entwicklungsunterstützung	0	1	1	1		0	0	1	4	14%
Dokumentation	0	1	1	1	1		0	0	4	14%
Ökosystem	0	0	1	1	1	1		0	4	14%
Lebenszyklus & öffentliches Interesse	0	1	1	1	0	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	Entwicklungsunterstützung	Dokumentation	Ökosystem	Lebenszyklus & öffentliches Interesse	Summe	%
Plattformspezifische Funktionen		0	1	1	0	1	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 erkennbar, 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 online <sup>4</sup> einsehbar und steht bei Bedarf ebenfalls als Download <sup>5</sup> 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

---

<sup>4</sup> <https://docs.flutter.dev/>

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

werden können. Weitere IDEs sind z. B. Android Studio oder IntelliJ IDEA. Besonders hervorzuheben 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 Websites <sup>6</sup> 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 Speicherverwaltung 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 releasen 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

<sup>6</sup> 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 [Lestál 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 testen <sup>7</sup>.

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

<sup>7</sup> [https://www.tutorialspoint.com/react\\_native/react\\_native\\_environment\\_setup.htm](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 wiederverwendbaren, 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 optionalem 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 Lernplattform <sup>8</sup> und dem Youtube-Kanal von Microsoft und dotnet zu finden. Die Dokumentation ist online <sup>9</sup> 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

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

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

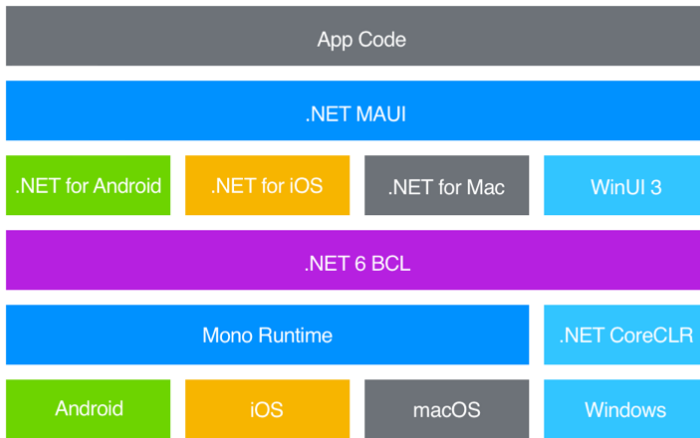


Abb. 6: Übersicht über die Architektur einer MAUI Applikation nach [Davidbritch 2022d]

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

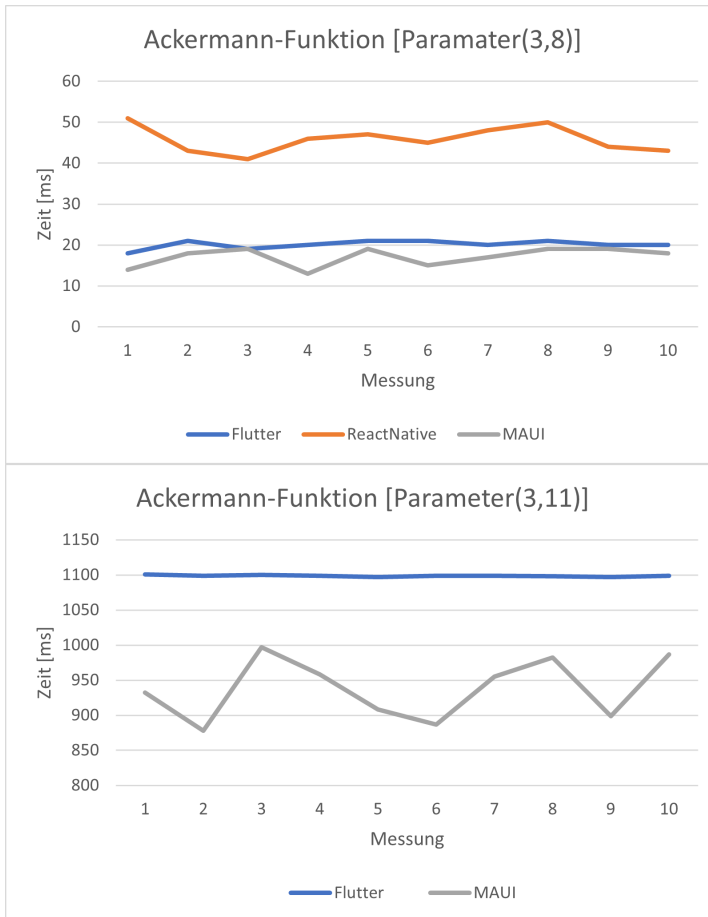


Abb. 7: Analyse der Frameworks anhand der Ackermann-Funktion

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 abgeschlagen 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

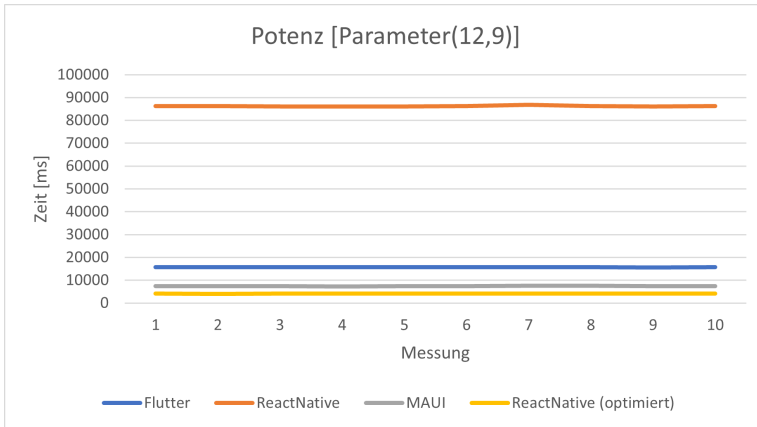


Abb. 8: Analyse der Frameworks anhand der Iterativen Potenzierung

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

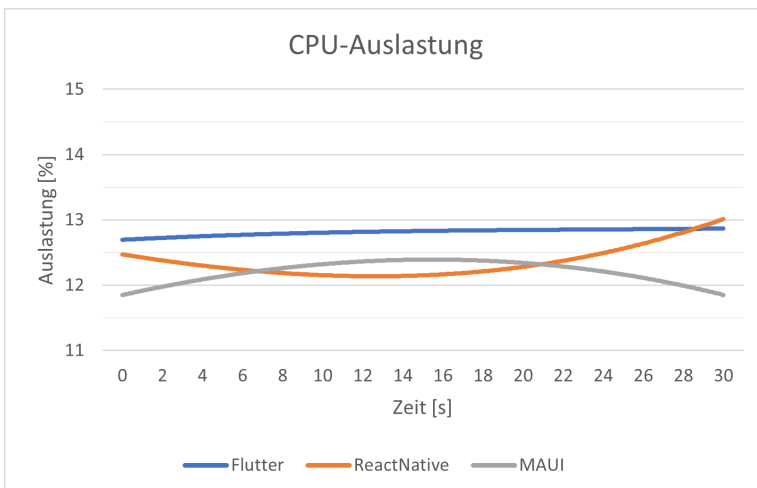


Abb. 9: CPU-Auslastung der Frameworks

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 Punktabzug 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 Punktabzug 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 Punktabzug ist notwendig, da keine eigenen, einzigartigen Widgets benutzt werden.

**Erste Schritte (3/5)** Es ist ein kleiner Punktabzug 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 Punktabzug 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 Punktabzug 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 Punktabzug notwendig, 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 Punktabzug 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 Punktabzug 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 Punktabzug, 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 Punktabzug notwendig, 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 einigem Abstand MAUI und ReactNative.

<div>Framework</div> <div>Anwendungsgebiet</div>	Flutter	MAUI	ReactNative
Formularlastig	4,32	4,00	4,07
Rechenintensiv	4,36	4,21	3,89
Sensorlastig	4,50	4,11	4,32
	<b>13,18</b>	<b>12,32</b>	<b>12,28</b>

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.
- [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-uberblick> (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.
- [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 Ullenboom, 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.
- [Wu 2018] Wenhao Wu. *React Native vs Flutter, Cross-platforms mobile application frameworks*. 2018. URL: <https://www.theseus.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.



# Resilienz in Systemarchitekturen

Robin Eppel<sup>1</sup>, Timo Vollert<sup>2</sup>

**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 Vorfalles 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 Forschungsgebiet 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 resilienter gemacht werden kann.

---

<sup>1</sup> DHBW Stuttgart Campus Horb, Studiengang Informatik, Florianstraße 15, 72160 Horb am Neckar, Deutschland  
i20010@hb.dhbw-stuttgart.de

<sup>2</sup> 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 partiellem 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 partiellem 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 Fallsicherheit 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 gemanageten 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 weist 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 kaskadieren 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]

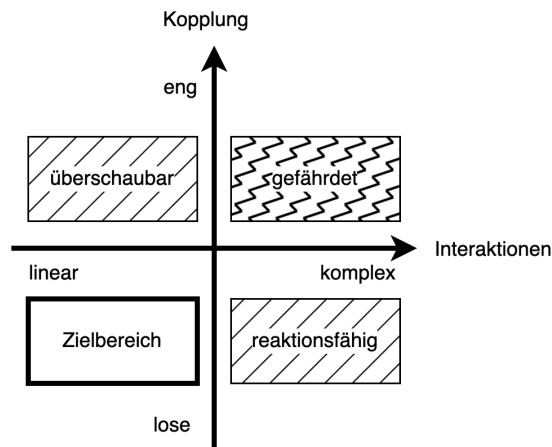


Abb. 1: Interaktionsdimensionen NAT

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 Unternehmenstätigkeiten 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.

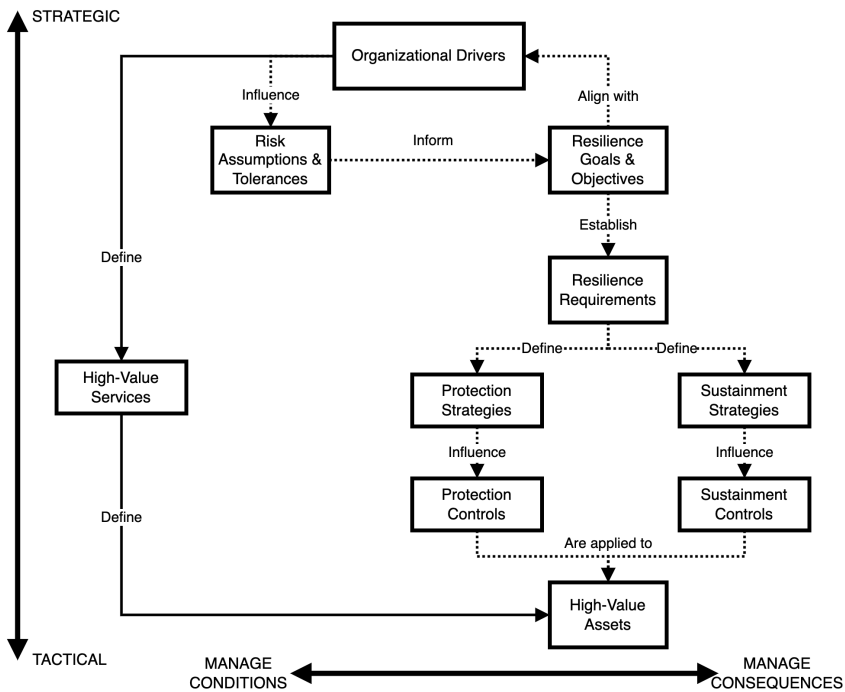


Abb. 2: Ableiten von Resilienzmaßnahmen aus Unternehmenszielen. Nachbildung von [Rich10, S. 26, Figure 9].

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 Unternehmenstätigkeit 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.

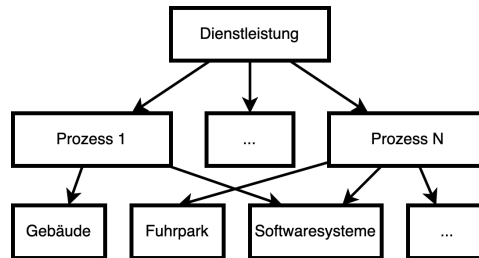


Abb. 3: Schützenswerte Ressourcen von priorisierten Dienstleistungen ableiten.

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 Störfall 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 resilienter 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.



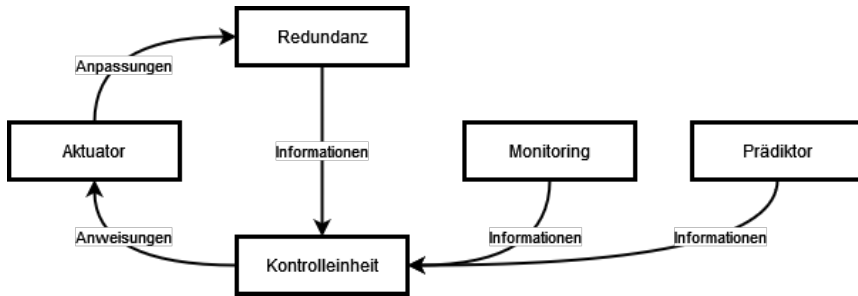


Abb. 4: Zusammenspiel der fünf Prinzipien. Adaption von [ZhLi10, Figure 5, S. 109].

**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 implementiert 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 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 eines Timeouts 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 Schiffes 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 weggefallene 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 abubrechen. 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 asynchrone Aufrufe 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 resilienter 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.





# Autorenverzeichnis

## **D**

Dürr, Jannik, 125

## **E**

Epple, Lukas, 35

Epple, Robin, 159

## **F**

Freudenberger, Jülf, 11

## **J**

Joas, Dominic, 125

Jooß, Reinhold, 65

## **K**

Kalmbach, Ruben, 125

Klimpel, Fabian, 35

## **L**

Liehner, Adrian, 11

## **S**

Sack, Raphael, 35

Schwab, Jonathan, 91

Sperling, Gabriel, 65

## **V**

Vollert, Timo, 159

## **W**

Weis, Jonas, 91

Wochele, Felix, 91