Kapitel 3 Modularisierung

albion.eu www.tectrain.ch www.accso.de

< Kapitel 2 Motivation Kapitel 4 Integration >

Kapitel 3 Modularisierung

FLEX Lehrplan

3 Modularisierung

Dauer: 120 Min Übungszeit: 30 Min

3.1 Begriffe und Konzepte

- Motivation für die Dekomposition in kleinere Systeme
- Unterschiedliche Arten von Modularisierung, Kopplung
- Systemgrenzen als Mittel für Isolation
- Hierarchische Struktur
- Anwendung, Applikation, Self-Contained System, Microservice
- Domain-Driven Design-Konzepte und „Strategic Design“, Bounded Contexts

3.2 Lernziele

3.2.1 Was sollen die Teilnehmer können?

- Teilnehmer können für eine gegebene Aufgabenstellung eine Zerlegung in einzelne Bausteine entwerfen.
- Die Teilnehmer sollen die Kommunikationsstruktur der Organisation beim Festlegen der Modulgrenzen berücksichtigen (Gesetz von Conway).
- Die Teilnehmer sollen technische Modularisierungskonzepte projektspezifisch bewerten und auswählen können.
- Die Teilnehmer sollen die Beziehungen zwischen Modulen sowie zwischen Modulen und Subdomänen veranschaulichen und analysieren können (Context Mapping).
- Die Teilnehmer können die Konsequenzen verschiedener Modularisierungsstrategien bewerten und den mit der Modularisierung verbundenen Aufwand dem zu erwartenden Nutzen gegenüberstellen.#
- Die Teilnehmer können die Auswirkungen der Modularisierungsstrategie auf die Autonomie von Bausteinen zur Entwicklungszeit und zur Laufzeit beurteilen.
- Die Teilnehmer können einen Plan zur Aufteilung eines Deployment-Monolithen in kleine Services aufstellen.
- Die Teilnehmer können ein Konzept erarbeiten, um ein System aus Services aufzubauen.
- Die Teilnehmer können eine geeignete Modularisierung und eine geeignete Granularität der Modularisierung wählen – abhängig von der Organisation und den Qualitätszielen.

3.2.2 Was sollen die Teilnehmer verstehen?

- Teilnehmer verstehen, dass jede Art von Bausteinen neben einer griffigen Bezeichnung eine Beschreibung benötigt,
  - was Bausteine dieser Art ausmacht,
  - wie ein solcher Baustein zur Laufzeit integriert wird,
  - wie ein solcher Baustein (im Sinne des Build-Systems) gebaut wird,
  - wie ein solcher Baustein deployt wird, getestet wird, skaliert wird.

- Teilnehmer verstehen, dass eine Integrationsstrategie darüber entscheidet, ob eine Abhängigkeit
  - erst zur Laufzeit entsteht,
  - zur Entwicklungszeit entsteht, oder
  - beim Deployment entsteht.
- Modularisierung hilft Ziele wie Parallelisierung der Entwicklung, unabhängiges Deployment/Austauschbarkeit zur Laufzeit, Rebuild /Reuse von Modulen und leichtere Verständlichkeit des Gesamtsystems zu erreichen.
- Daher ist beispielsweise Continuous Delivery und die Automatisierung von Test und Deployment ein wichtiger Einfluss auf die Modularisierung.
- Modularisierung bezeichnet Dekomposition eines Systems in kleinere Teile. Diese Teile nach der Dekomposition wieder zu integrieren, verursacht organisatorische und technische Aufwände. Diese Aufwände müssen durch die Vorteile, die durch die Modularisierung erreicht werden, mehr als ausgeglichen werden.
- Teilnehmer verstehen, dass zur Erreichung von höherer Autonomie der Entwicklungsteams ein Komponentenschnitt besser entlang fachlicher Grenzen anstatt entlang technischer Grenzen erfolgt.
- Je nach gewählter Modularisierungstechnologie besteht Kopplung auf unterschiedlichen Ebenen:
  - Sourcecode (Modularisierung mit Dateien, Klassen, Packages, Namensräumen etc.)
  - Kompilat (Modularisierung mit JARs, Bibliotheken, DLLs etc.)
  - Laufzeitumgebung (Betriebssystem, virtuelle Maschine oder Container)
  - Netzwerkprotokoll (Verteilung auf verschiedene Prozesse)
- Eine Kopplung auf Ebene des Sourcecodes erfordert sehr enge Kooperation sowie gemeinsames SCM. Eine Kopplung auf Ebene des Kompilats bedeutet, dass die Bausteine in der Regel gemeinsam deployt werden müssen. Nur eine Verteilung auf unterschiedliche Anwendungen/Prozesse ist praktikabel im Hinblick auf ein unabhängiges Deployment.
- Teilnehmer verstehen, dass eine vollständige Isolation zwischen Bausteinen nur durch eine Trennung in allen Phasen (Entwicklung, Deployment und Laufzeit) gewährleistet werden kann. Ist das nicht der Fall, können unerwünschte Abhängigkeiten nicht ausgeschlossen werden. Gleichzeitig verstehen die Teilnehmer auch, dass es sinnvoll sein kann, aus Gründen wie effizienter Ressourcennutzung oder Komplexitätsreduktion auf eine vollständige Isolation zu verzichten.
- Teilnehmer verstehen, dass bei der Verteilung auf unterschiedliche Prozesse manche Abhängigkeiten nicht mehr in der Implementierung existieren, sondern erst zur Laufzeit entstehen. Dadurch steigen die Anforderungen an die Überwachung dieser Schnittstellen.
- Microservices sind unabhängige Deployment-Einheiten und damit auch unabhängige Prozesse, die ihre Funktionen über leichtgewichtige Protokolle exponieren, aber auch ein UI haben können. Für die Implementierung jedes einzelnen Microservices können unterschiedliche Technologieentscheidungen getroffen werden.
- Ein Self-Contained System (SCS) stellt ein fachlich eigenständiges System dar. Es beinhaltet üblicherweise UI und Persistenz. Es kann aus mehreren Microservices bestehen. Fachlich deckt ein SCS meist einen Bounded Context ab.
- Der Modulschnitt kann entlang fachlicher oder technischer Grenzen erfolgen. In den meisten Fällen empfiehlt sich ein fachlicher Schnitt, da sich so fachliche Anforderungen klarer einem Modul zuordnen lassen und somit nicht mehrere Module für die Umsetzung einer fachlichen Anforderung angepasst werden müssen. Dabei kann jedes Modul sein eigenes Domänenmodell im Sinne eines Bounded Context und damit unterschiedliche Sichten auf ein Geschäftsobjekt mit eigenen Daten haben.
- Transaktionale Konsistenz lässt sich über Prozessgrenzen hinweg nur über zusätzliche Mechanismen erreichen. Wird ein System in mehrere Prozesse aufgeteilt, so stellt die Modulgrenze daher häufig auch die Grenze für transaktionale Konsistenz dar. Daher muss ein DDD-Aggregat in einem Modul verwaltet werden.
- Teilnehmer verstehen, welche Modularisierungskonzepte nicht nur für Transaktions-, sondern auch für Batch- und Datenfluss-orientierte Systeme genutzt werden können.
- Unterschiedliche Grade an Vorgaben für die Entwicklung eines Bausteins können sinnvoll sein. Einige Vorgaben sollten besser übergeordnet allgemein für die Integration mit anderen Bausteinen dieser Art gelten. Die übergreifenden Entscheidungen, die alle Systeme beeinflussen, können eine Makro-Architektur bilden - dazu zählen beispielsweise die Kommunikationsprotokolle oder Betriebsstandards. Mikro-Architektur kann die Architektur eines einzelnen Systems sein. Es ist weitgehend unabhängig von anderen Systemen. Zu große Einschränkungen auf Ebene der Makro-Architektur führen dazu, dass die Architektur insgesamt auf weniger Probleme angewendet werden kann.

3.2.3 Was sollen die Teilnehmer kennen?

- Die Teilnehmer sollen verschiedene technische Modularisierungsmöglichkeiten kennen: z. B. Dateien, JARs, OSGI Bundles, Prozesse, Microservices, SCS.
- Die Teilnehmer sollen verschiedene technische Modularisierungsmöglichkeiten kennen: Sourcecode-Dateien, Bibliotheken, Frameworks, Plugins, Anwendungen, Prozesse, Microservices, Self-Contained Systems.
- Die Teilnehmer sollen folgende Begriffe aus dem Domain-Driven Design kennen: Aggregate Root, Context Mapping, Bounded Contexts und Beziehungen dazwischen (z. B. Anti-Corruption Layer).
- Die Teilnehmer sollen "The Twelve-Factor App" kennen.
- Die Teilnehmer sollen das Gesetz von Conway kennen.

3.3 Referenzen

- Eric Evans: Domain-Driven Design: Tackling Complexity in the Heart of Software, AddisonWesley Professional, 2003
- http://12factor.net/
- Sam Newmann: Building Microservices: Designung Fine-Grained Systems, O'Reilly Media, 2015
- Eberhard Wolff: Microservices - Grundlagen flexibler Software Architekturen, dpunkt, 2015
- http://martinfowler.com/articles/microservices.html

Inhalte

(A) Anforderungen und Treiber für Modularisierung

1. Zielbild

Wir unterscheiden Modularisierung auf diesen zwei Ebenen:

- Trennung in fachliche Einheiten als fachliche Zerlegung (Säulen, Business Verticles)
- Trennung der technischen Zuständigkeiten innerhalb einer solchen fachlichen Einheit (Schichtentrennung)

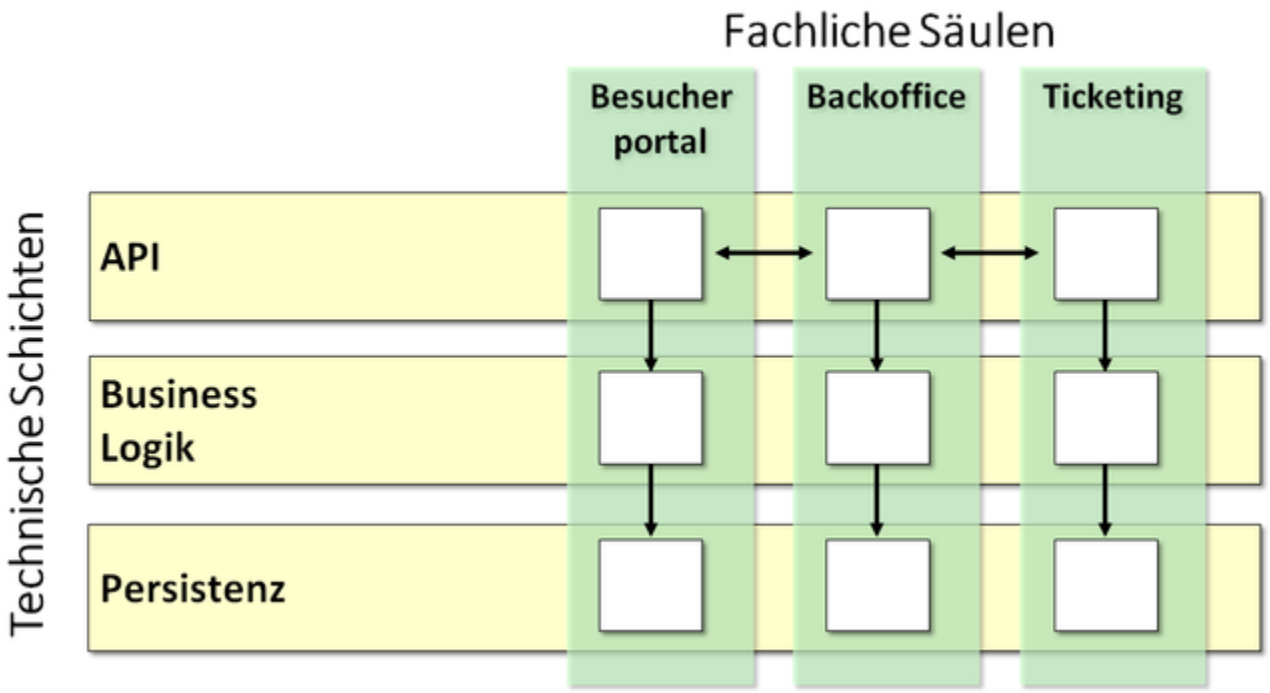


Abbildung: Fachliche Säulen - Technische Schichten der Flexinale-Case-Study, siehe Kapitel 7 und Übungen (Quelle: selbst erstellt)

2. Warum Modularisierung?

Modularisierung ist in der Softwareentwicklung aus vielerlei Gründen wichtig, v.a. hinsichtlich der Wartbarkeit: Modularisierung trägt dazu bei, dass Softwaresysteme leichter zu verstehen, zu testen und zu ändern sind, indem sie in kleinere, besser zu verwaltende Teile zerlegt werden. Dies erleichtert Eingrenzung und Behebung von Fehlern sowie die Erweiterung des Systems um neue Funktionalität.

Im Einzelnen:

1. **Wartbarkeit** bezieht sich auf die Fähigkeit, Änderungen und Aktualisierungen an der Komponente in einer effizienten und zuverlässigen Art und Weise durchführen zu können. Eine wartbare Softwarekomponente sollte so gestaltet sein, dass sie **leicht verständlich** und nachvollziehbar ist, so dass Entwickler schnell verstehen können, wie sie funktioniert und welche Teile der Komponente geändert werden müssen, um neue Funktionen hinzuzufügen oder Fehler zu beheben. Zur besseren Verständlichkeit trägt die **Strukturierung** durch die Modularisierung bei.
2. **Testbarkeit:** Die Modularisierung ermöglicht die Entwicklung von Softwaresystemen, die leichter zu testen sind. Kleinere, unabhängige Komponenten sind jeweils für sich leichter testbar, wodurch Fehler schneller eingegrenzt und behoben werden können.
3. **Deploybarkeit**: Modularisierung erleichtert die Deploybarkeit, da es möglich wird, nur einzelne Module eines Systems auszutauschen. Auch kann der Prozess des Deployments verbessert werden, wenn sich Änderungen klar lokalisieren lassen und damit auch bspw. Tests und Qualitätssicherung nur für das geänderte Modul durchgeführt werden müssen.
4. **Skalierbarkeit:** Modularisierung ermöglicht die Entwicklung von Softwaresystemen, die leichter skaliert werden können, um veränderten Anforderungen gerecht zu werden. Durch die Aufteilung eines Systems in kleinere, unabhängige Komponenten wird es einfacher, bei Bedarf Ressourcen hinzuzufügen oder zu entfernen, um größere Nutzungszahlen oder neue Funktionen zu unterstützen.
5. **Stabilität** und verbesserte **Fehlerisolierung:** Die Modularisierung von Software trägt zur Stabilität bei, da die einzelnen Komponenten unabhängig voneinander entwickelt, getestet und gewartet werden können. Dies reduziert Fehler und führt zu einem vorhersehbareren Verhalten des Gesamtsystems, da Änderungen an einer Komponente sich nicht (unbedingt bzw. direkt) auf andere Komponenten auswirken.

6. **Teamarbeit** und **organisatorische Skalierung:** Die Modularisierung ermöglicht eine Entwicklungsmethodik von mehreren, parallel arbeitenden Teams: Die Aufteilung des Systems in möglichst unabhängige Komponenten lässt Teams parallel operieren, was Geschwindigkeit und Effizienz verbessert (bei entsprechenden Kosten, siehe unten).
7. **Wiederverwendung:** Modularisierung ermöglicht die Entwicklung von wiederverwendbarem Code, der in verschiedenen Teilen eines Systems oder in verschiedenen Systemen verwendet werden kann. Dies spart Zeit und Aufwand bei der Entwicklung und kann auch die Konsistenz und Qualität des Codes verbessern.
8. **Ersetzbarkeit:** Die Ersetzbarkeit einer Softwarekomponente erlaubt es, eine Komponente durch eine andere zu ersetzen, ohne die Gesamtfunktionalität des Systems zu beeinträchtigen.

Modularisierung erhöht damit die **Flexibilität**, denn sie ermöglicht die Entwicklung von Softwaresystemen, die sich leichter an veränderte Anforderungen anpassen oder neue Technologien nutzen. Die Aufteilung in kleinere, unabhängige Komponenten macht es einfacher, einzelne Komponenten bei Bedarf zu ändern oder zu ersetzen.

Modularisierung bedeutet nicht zwingend, dass die einzelnen Komponenten auch zu eigenen Deployment-Einheiten werden, siehe Kapitel 3, Abschnitt (E).

3. Wartbarkeit, Evolvierbarkeit

Wartbarkeit als auch die Evolvierbarkeit sind wichtige Aspekte bei der Softwaregestaltung und -entwicklung. Wenn Software leicht zu verstehen, zu testen und zu ändern ist, bzw. leicht geändert werden kann, um neue Anforderungen zu erfüllen, entstehen Softwaresysteme, die im Laufe der Zeit insgesamt zuverlässiger, effizienter und kostengünstiger sind.

Die **Wartbarkeit** von Software bezieht sich auf die Fähigkeit, mit der ein Softwaresystem (leicht oder schwer) verstanden, geändert und auf dem neuesten Stand gehalten werden kann. Ein wartbares Softwaresystem ist ein System, das leicht zu verstehen, zu testen und zu ändern ist - also so konzipiert ist, dass Änderungen mit minimalen Auswirkungen auf den Rest des Systems vorgenommen werden können. Dies ist wichtig, da Softwaresysteme häufig aktualisiert oder geändert werden müssen, um Fehler zu beheben, neue Funktionen hinzuzufügen oder mit veränderten Anforderungen Schritt zu halten. Wartbarkeit zielt mehr auf den "Erhalt des Status Quo", also die Software in jeder Hinsicht (auch technisch) auf dem aktuellen Stand zu halten.

Die **Evolvierbarkeit** von Software bezieht sich auf die Fähigkeit eines Softwaresystems, sich im Laufe der Zeit anzupassen und zu verändern, zielt also stärker auf die echte Weiterentwicklung im funktionalen Sinne, den Einbau neuer Features. Die Evolvierbarkeit einer Software ist demnach optimal, wenn ein Feature-Request zu einem späten Zeitpunkt des Entwicklungsprojekts mit dem gleichen Aufwand umgesetzt werden kann, wie wenn das Feature von Anfang an gefordert gewesen wäre.

4. Test

Modulare Systeme haben Vorteile beim Testen:

1. **Testbarkeit:** Die Modularisierung ermöglicht die Entwicklung von Softwaresystemen, die sich leichter testen lassen. Durch die Aufteilung eines Systems in kleinere, unabhängige Komponenten ist es einfacher, jede Komponente einzeln zu testen, was dazu beitragen kann, Fehler schneller zu erkennen und zu beheben.
2. **Automatisierte Tests:** Die Modularisierung ermöglicht die Entwicklung von Softwaresystemen, die auch einfacher automatisiert getestet werden können. Tests für die Einzel-Komponenten sind leichter automatisierbar, was Zeit und Aufwand für manuelle Tests spart.
3. **Einfachere Testfälle:** Die Modularisierung ermöglicht es, ein System in kleinere, unabhängige Komponenten zu zerlegen. Dadurch werden die Testfälle einfacher und konzentrieren sich auf bestimmte Funktionen, was das Testen, Fehlersuche und Wartung erleichtert.

5. Deployment und Betrieb

Modulare Systeme haben verschiedene Vorteile, wenn es um **Deployment und Betrieb** geht (vgl. auch Kapitel 5 und 6):

1. **Skalierbarkeit:** Dank der Modularisierung können einzelne Komponenten eines Systems unabhängig voneinander skaliert werden, was die Skalierbarkeit eines Systems insgesamt verbessern kann. Durch die Aufteilung eines Systems in kleinere, unabhängige Komponenten ist es möglich, die Last auf mehrere Prozessoren oder Maschinen zu verteilen, was die Gesamt-Leistung und Skalierbarkeit verbessert. Außerdem können die Ressourcen effizienter genutzt werden, da verschiedene Komponenten optimiert und auf unterschiedlicher Hardware ausgeführt werden können.
2. **Flexibilität:** Modularisierung ermöglicht den Einsatz verschiedener Komponenten eines Systems auf verschiedenen Servern oder in verschiedenen Umgebungen, was zu einer größeren Flexibilität bei der Bereitstellung und dem Betrieb eines Systems führen kann.
3. **Wiederverwendung:** Modularisierung ermöglicht die Entwicklung von wiederverwendbarem Code, der in verschiedenen Teilen eines Systems oder in verschiedenen Systemen insgesamt verwendet werden kann. Dies kann die Bereitstellung und den Betrieb eines Systems erleichtern.
4. **Deployment**: Durch die Aufteilung eines Systems in kleinere, unabhängige Komponenten wird es einfacher, diese separat zu paketieren und für ein Deployment bereitzustellen: Dies verringert die Komplexität des Bereitstellungsprozesses und ermöglicht häufigere und schnellere Bereitstellungen.
5. **Versionierung:** Die Modularisierung ermöglicht i.d.R. die parallele Bereitstellung und Installation verschiedener Versionen einer Komponente, wodurch es einfacher wird, neue Versionen einer Komponente zu testen, bevor sie für das gesamte System produktiv eingeführt werden.
6. **Wartung:** Modularisierung ermöglicht eine effizientere Wartung eines Systems, da das System in kleinere, unabhängige Komponenten zerlegt werden kann. Es wird somit einfacher, Fehler zu erkennen und zu beheben und dem System neue Funktionen hinzuzufügen.
7. **Isolation von Fehlern**: Die Modularisierung ermöglicht die Isolation von Fehlern, was Auswirkungen eines Fehlers auf das gesamte System minimiert. Wenn eine Komponente eines Systems ausfällt, kann sie ersetzt werden.

Insbesondere die Punkte 1. und 7., Skalierbarkeit und Fehlerisolierung, implizieren, dass die verschiedenen Module auch separate Deployment-Units sind.

6. Replace, Rebuild, Reuse

Die **Wiederverwendung** von Modulen in einem System kann durch den Entwurf und die Entwicklung modularer, wiederverwendbarer Komponenten erreicht werden, die in verschiedene Teile des Systems integriert werden können, aber auch integriert werden müssen. Dies kann erreicht werden durch:

- Definition klarer **Schnittstellen** für jedes Modul, die angeben, wie es mit anderen Teilen des Systems interagiert
- kleine und konzentrierte Module mit einer einzigen, **klar definierten Verantwortung**
- Entwurf von Modulen, die unabhängig und in sich **geschlossen** sind, mit minimalen Abhängigkeiten von anderen Teilen des Systems
- Erstellung von **Unit-Tests** für jedes Modul, um sicherzustellen, dass es korrekt funktioniert und leicht gewartet werden kann
- **Dokumentation** der Module und ihrer Verwendung, damit andere Entwickler sie verstehen und effektiv nutzen können.
- Es ist auch wichtig, eine gute **Versionskontrolle** und **Paketverwaltung** mit entsprechenden Methoden und Tools für die Module zu etablieren und zu nutzen, um Änderungen zu verfolgen, Abhängigkeiten zu managen und die Aktualisierung der Module zu erleichtern.

Darüber hinaus ist es wichtig, ein **gutes Verständnis des Gesamtsystems** und seiner Anforderungen zu haben, um die Teile des Systems zu identifizieren, die überhaupt (sinnvoll, mit entsprechender Kosten-Nutzenrechnung) modularisiert und wiederverwendet werden können.

7. Vorteile und Kosten der Aufteilung und Zerlegung eines Softwaresystems

Die Zerlegung eines Softwaresystems in unabhängige Komponenten bringt Vorteile (siehe oben) mit sich, die erhöhte Flexibilität verursacht entsprechend **Kosten und Aufwände**:

- Erhöhter Entwicklungsaufwand: Die Erstellung und Pflege einzelner Komponenten ist i.d.R. komplexer und zeitaufwändiger als die Arbeit an einem monolithischen System (Beispiele: erschwertes Refactoring, Versionierung).
- Kommunikationsaufwand: Die Komponenten müssen miteinander zur Laufzeit kommunizieren, was die Komplexität erhöht und potenzielle Fehlerquellen schafft.
- Erhöhter Deploymentaufwand: Das Deployment jeder einzelnen Komponente ist i.d.R. schwieriger und zeitaufwändiger als Deployment eines monolithischen Systems.
- Erhöhter Testaufwand: Das Testen jeder einzelnen Komponente - vor allem im integrativen Zusammenspiel ist i.d.R. schwieriger und zeitaufwändiger als das Testen eines monolithischen Systems.

Verschiedene **Aufgaben** werden durch die Zerlegung in Komponenten nötig und verursacht, u.a.:

- Entwurf und Implementierung von Schnittstellen, über die die Komponenten miteinander kommunizieren können.
- Verwaltung von Abhängigkeiten zwischen Komponenten.
- Testen und Bereitstellen jeder Komponente separat.
- Verwaltung der Versionierung und der Updates für einzelne Komponenten.

8. Parallelisierung und Skalierung in der Entwicklung

Die Modularisierung hilft bei der **parallelen Abarbeitung** von Arbeitspaketen (in der Entwicklung), da sie es ermöglicht, verschiedene Komponenten des Systems parallel zu entwickeln, zu testen und auszuführen, ohne dass diese Teams sich gegenseitig stören oder voneinander abhängig sind. Allerdings ist trotzdem Kommunikationsaufwand zwischen den Teams notwendig, um zunächst die Komponenten zu schneiden und ihre Schnittstellen zu besprechen.

9. Organisatorische Aspekte: Conway's Law, Team Topologies

Info
http://www.melconway.com/Home/Committees_Paper.html

https://martinfowler.com/bliki/ConwaysLaw.html

Conway's Law ist ein Prinzip des SW-Engineerings, das besagt, dass "*Organisationen, die Systeme entwerfen, gezwungen sind, Entwürfe zu erstellen, die Kopien der Kommunikationsstrukturen dieser Organisationen sind*". Mit anderen Worten: Die Architektur eines Softwaresystems spiegelt die Kommunikationsstruktur der Organisation wider, die es entwickelt hat. Das bedeutet, dass ein Unternehmen mit isolierten Teams oder schlechten Kommunikationskanälen wahrscheinlich ein Softwaresystem mit geringer Kohäsion und hoher Kopplung produziert, das entsprechend schwer zu verstehen und zu warten ist.

Abbildung: Conway's Law
(Quelle: Martin Fowler: "Microservices". https://www.martinfowler.com/articles/microservices.html)

Die Umkehrung dieses Conway'schen Laws sagt aus: "*Wenn Sie die Kommunikationsstruktur Ihrer Organisation verbessern wollen, sollten Sie die Architektur Ihres Systems verbessern*". Mit anderen Worten: Wenn ein Unternehmen die Art und Weise, wie seine Teams kommunizieren und zusammenarbeiten, verbessern möchte, sollte es sich auf die Verbesserung des Designs seiner Softwaresysteme konzentrieren. Dahinter steht der Gedanke, dass durch die Gestaltung von Softwaresystemen, die einfach zu verstehen und zu warten sind, die Kommunikation und Zusammenarbeit der Teams in diesen Systemen erleichtert wird. Dies sollte zu einer besseren Kommunikation, einer besseren Abstimmung und einem besseren Endprodukt führen.

Diese Kommunikationsstruktur ist i.d.R. nicht identisch mit der **Organisationsstruktur** des Unternehmens, vgl. Organigramm.

Das **Inverse Conway Maneuver** ist eine Technik, bei der dieses Wissen genutzt wird, um die Systemarchitektur mit der Kommunikationsstruktur des Unternehmens in Einklang zu bringen, um die Zusammenarbeit zu verbessern und die Komplexität zu verringern.

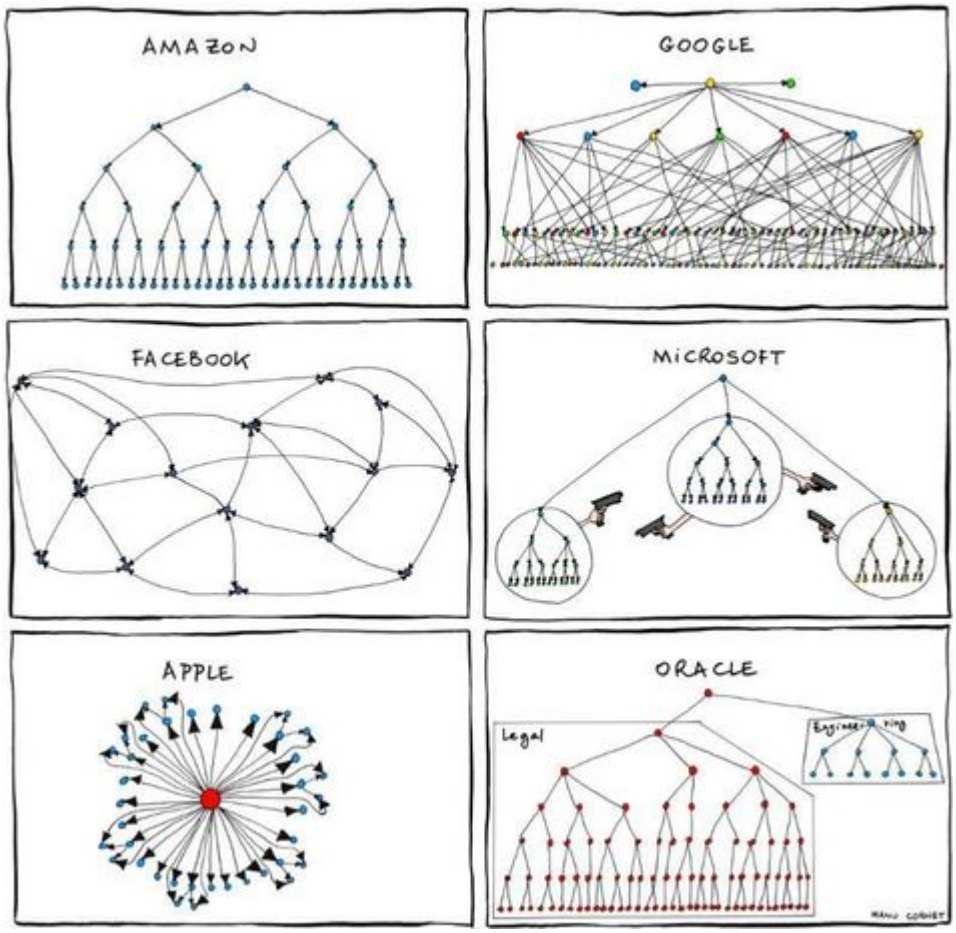


Abbildung: Organisationsstruktur in bekannten Unternehmen
(Quelle: Post auf Linkedin, https://www.linkedin.com/posts/milanmilanovic_technology-softwareengineering-softwarearchitecture-activity-7027531191008309248-nKUf/)

10. Organisatorische Aspekte: Team Topologies

Info
Matthew Skelton, Manuel Pais : "Team Topologies: Organizing Business and Technology Teams for Fast Flow Taschenbuch", IT Revolution Press, 2019

Team-Struktur und Architektur müssen also kongruent sein und haben - wechselseitig - Einfluss aufeinander. Ein Team optimal für ein(e) Software/Architektur/Projekt richtig aufzustellen, ist nicht trivial. Hilfestellung kann "Team Topologies" geben, das Zielvorgaben (z.B. für die Teamgröße ("fünf bis neun Personen")) und Strukturvorschläge macht:

- Ein "Stream-aligned Team" schafft Mehrwert entlang des Value Streams, i.d.R. angeleitet durch einen Product Owner. Es ist crossfunktional aufgesetzt, um mit seinem Kompetenzmix schnell Mehrwert zu schaffen.
- Ein "Platform Team" stellt eine zugrundeliegende Plattform für ein "Stream-aligned Team" zur Verfügung (i.d.R. für komplexe Technologien und Infrastruktur). Entscheidend ist, dass die Plattform Services anbietet, die die Komplexität vereinfacht und dem "Stream-aligned Team" entscheidend Arbeit abnimmt.
- Ein "Complicated Subsytem Team" hat den speziellen Auftrag, ein kompliziertes Teilsystem (fachliches oder technisches Thema) umzusetzen - ebenfalls, um dem "Stream-aligned Team" entscheidend Arbeit abzunehmen. Es wird nur bei Bedarf gebildet und schnell wieder aufgelöst.
- Ein "Enabling Team" unterstützt andere Teams beim Erlernen neuer Technologien, Kompetenzen, Infrastruktur. Es ist Vorreiter für neue Themen und besteht i.d.R. aus Experten für das Thema.

11. Leitlinien für den Schnitt: Wo und wie lassen sich Module aufteilen, wie finde ich die Grenzen?

Modularisierung teilt ein System in kleinere Einheiten auf. Dazu ist die Frage zu beantworten: "*Wo und wie lässt sich das System (in Module) aufteilen, wie finde ich deren Grenzen?*"

statische Kriterien:

- **Daten** - warum und wo sind die "Grenzen" kohärenter Datenmodelle (siehe z.B. DDD's Aggregate)
- **Benutzer** und **Benutzergruppen** unter dem Aspekt von Security
- **Anwendungsfälle** und **Nutzungsszenarien:** Wann, warum und wie oft werden welche Dienste und Anwendungsfälle aufgerufen?

dynamische Kriterien:

- **Mengengerüste:** Wie viele Anfragen werden gestartet? Wann, wie oft, zu welchen Peak-Zeiten?

Dieses Thema wird bei "Domain-Driven Design (DDD) vertieft, siehe Kapitel 4 (B).

12. The Twelve-Factor App (Englisch)

Info
https://12factor.net/

The **Twelve-Factor App** is a methodology for building software-as-a-service (SaaS) applications that aims to make them easy to scale and maintain. It lays out 12 principles for building robust, scalable, and maintainable applications:

- I. Codebase - One codebase tracked in revision control, many deploys
- II. Dependencies - Explicitly declare and isolate dependencies
- III. Config - Store config in the environment
- IV. Backing services - Treat backing services as attached resources
- V. Build, release, run - Strictly separate build and run stages
- VI. Processes - Execute the app as one or more stateless processes
- VII. Port binding - Export services via port binding
- VIII. Concurrency - Scale out via the process model
- IX. Disposability - Maximize robustness with fast startup and graceful shutdown
- X. Dev/prod parity - Keep development, staging, and production as similar as possible
- XI. Logs - Treat logs as event streams
- XII. Admin processes - Run admin/management tasks as one-off processes

(B) Komponentenschnitt und -bildung

1. Charakteristiken von Komponenten

Software wird in Komponenten zerlegt, um die Komplexität beherrschbar zu machen. Es gibt viele Definitionen für den Begriff "Komponente", z. B. von Clemens Szyperski, siehe:

Info
https://wiki.c2.com/?ComponentDefinition

Eine **komponentenbasierte Architektur** strukturiert ein Softwaresystem als eine Menge von Komponenten, die zu einem vollständigen System zusammengesetzt werden können.

Wichtige Konzepte zur Definition von **Komponenten** sind**:**

1. **Abstraktion:** Komponentenbildung beruht auf dem Prinzip der Abstraktion, d. h. der Trennung der wesentlichen Features von unwesentlichen Details.
2. **Kapselung:** Implementierungsdetails einer Komponente werden vor anderen Komponenten verborgen.
3. Die Kapselung wird ermöglicht durch **Schnittstellen**: Eine Komponente exportiert ihre Funktionalität über eine oder mehrere Schnittstelle, andere Komponenten / Aufrufer verwenden und kennen sie also ausschließlich über diese Schnittstelle. Sie kennen keine Implementierungsdetails.
4. **Lose Kopplung** und **Hohe Kohäsion:** Kopplung bezeichnet den Grad Abhängigkeiten untereinander. Kohässion bezeichnet den Zusammenhalts innerhalb einer Komponente. Siehe 6..

Eine **Softwarekomponente** ist somit eine in sich geschlossene Einheit von Funktionen (hohe Kohäsion), die leicht in ein größeres Softwaresystem integriert werden kann (lose Kopplung). Sie ist eine Softwareeinheit, die durch Fachlichkeit, Technik, Planung und Teamarbeit bestimmt wird. Sie kann entweder eine einzelne Klasse, eine Gruppe verwandter Klassen oder - typischer - ein komplexes Modul sein, das mehrere Klassen und andere Ressourcen umfasst. Eine Softwarekomponente hat in der Regel eine klar definierte Schnittstelle, die angibt, wie sie von anderen Komponenten im System verwendet werden kann. Eine Softwarekomponente kann als eine Blackbox betrachtet werden, die Funktionalität kapselt und bewusst interne Implementierungsdetails verbirgt. Dadurch kann die Komponente leicht(er) ersetzt oder aktualisiert werden, ohne dass der Rest des Systems beeinträchtigt wird.

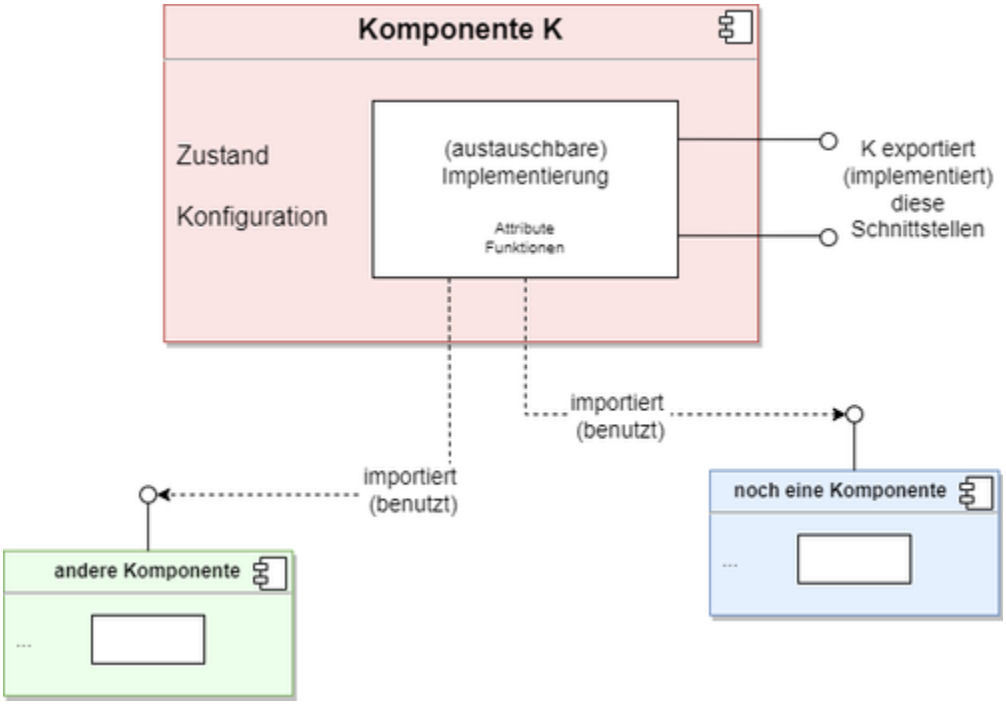


Abbildung: Komponente mit ihren Schnittstellen und ihren Abhängigkeiten
(Quelle: selbst erstellt)

Merkmale einer Softwarekomponente:

- Sie ist austauschbar, d. h. sie kann gegen eine andere Implementierung ausgetauscht werden, ohne dass der Rest des Systems beeinträchtigt wird.
- Sie ist wiederverwendbar, d. h. sie kann an mehreren Stellen und in verschiedenen Zusammenhängen eingesetzt werden.
- Sie ist modular, das heißt, sie kann unabhängig von anderen Komponenten entwickelt, getestet und eingesetzt werden.
- Sie ist komponierbar, d. h. sie kann mit anderen Komponenten kombiniert werden, um komplexere Funktionen zu schaffen.
- Sie hat eine klar definierte Schnittstelle, so dass sie leicht in andere Systeme und Anwendungen integriert werden kann.
- Ihre Abhängigkeiten (sofern vorhanden) zu anderen (Schnittstellen von) Komponenten sind klar definiert.

Eine Komponente kann, aber muss nicht separat deploybar sein.

Eine **Komponente** besteht aus:

- Name bzw. Namespace,
- Version,
- gut definierten ausgehenden (bereitgestellte) Schnittstellen,
- gut definierten eingehenden (verwendete) Schnittstellen - also Abhängigkeiten,
- Code (wobei die Implementierungsdetails hinter der Schnittstelle versteckt sind),
- ggf. Ressourcen und
- Konfiguration.

Das **Interface Segregation Principle** (ISP) hilft dabei, Schnittstellen einer Komponente zu entwerfen und insbesondere Interfaces, die zu groß sind, aufzuteilen: :

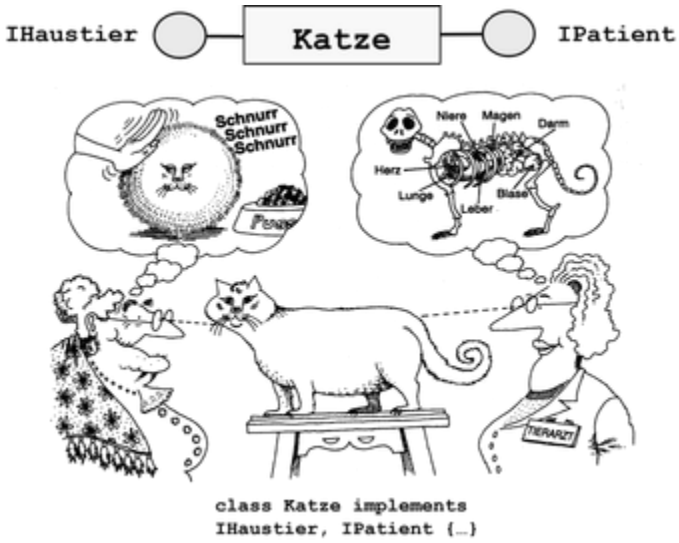


Abbildung: Interfaces als verschieden Sichten auf ein Objekt
(Quelle: Grady Booch: "Objektorientierte Analyse und Design", Addison-Wesley, 1994)

Eine Komponente kann mehrere Schnittstellen implementieren, die unterschiedlich Sichten auf die Komponente zulassen, wie in der Abbildung der Katze und ihrer Schnittstellen dargestellt ist. Dabei ist hier die Katze als einzelne Klasse dargestellt, aber das Prinzip ist für Komponenten das gleiche (und jeder Katzenkenner würde sicherlich zustimmen, dass zur Implementierung des komplexen Wesens einer Katze viel mehr als ein einzige Klasse notwendig wäre).

Ein anderes Beispiel für eine Komponente mit verschiedenen Schnittstellen ist eine Komponente zur Verwaltung von Berechtigungen. Sie wird a) eine Schnittstelle implementieren zur Abfrage von Berechtigungen (z.B. "hat User X die Berichtigung Y?"), b) eine Schnittstelle zur Administration von Berechtigungen (z.B. "Gibt User X die Berechtigng Y").

Info
Das ISP besagt: “Clients should not be forced to depend upon interfaces that they do not use.” (Robert C. Martin)

Siehe https://de.wikipedia.org/wiki/Prinzipien_objektorientierten_Designs#Interface-Segregation-Prinzip

Achtung:

1. Falls eine Komponente hier zwei (oder mehreren) Interfaces auf *ein*e Fachlichkeit anbietet, so ist das in Ordnung und bedeutet "Interface Segregation". (Im Bild: Eine Katze implementiert zwei Sichten auf sich.)

2. Falls über diese Sichten sehr verschiedene Dinge nach außen bereitgestellt werden, so widerspricht das i.d.R. dem Prinzip "Separation of Concerns". Lösung: Komponente aufteilen.
3. Für eine verteilte Anwendung (z.B. mit Microservices, siehe Kapitel 3(E)) ist eine Trennung von Sichten in verschiedene Services immer zu bevorzugen, selbst wenn das Redundanz von Daten, Funktionalität etc. bedeutet. (Hier wird DRY (= Don't repeat yourself) bewusst verletzt.)

2. Komponente und Modul

Die Begriffe "Modul" und "Komponente" sind nicht sauber von einander angegrenzt, nicht trennscharf und werden daher oft als Synonyme verwendet. Wir verwenden daher den Begriff "Komponente". Jedoch wird in der Literatur häufig "Modul" genutzt. Wo hier externe Quellen referenziert sind, die von Modulen sprechen, ist "Modul" als Synonym zu "Komponente" zu betrachten.

Eine **Komponente** ist eine Einheit für Replacebility und Reuse.

- Es handelt sich in der Regel um eine einzelne Klasse oder eine Gruppe verwandter Klassen, die über eine klar definierte Schnittstelle verfügen und leicht ersetzt oder aktualisiert werden können, ohne das übrige System zu beeinträchtigen.
- Komponenten werden durch einen Komponentenkonfigurator (vgl. z.B. Spring) zu größeren Systemen zusammengesteckt und -stellt. Eine Komponente ist in der Regel eine in sich geschlossene Funktionseinheit, die leicht in ein größeres Softwaresystem integriert werden kann.
- Eine Komponente kann auch andere Ressourcen, Konfigurationsdateien und Dokumentation enthalten.
- Komponenten können andere Komponenten enthalten, also eine Hierarchie bilden.

Ein **Modul** ist ein Synonym für eine Komponente. Mögliche weitere Abgrenzungen sind:

- Einige Modul-Frameworks wie OSGi erlauben einen **Lebenszyklus** zur Laufzeit (um Module zu starten, zu stoppen, ...) (wie es intern mit dem Plugin-System von Eclipse genutzt wird).
- Einige Modul-Frameworks wie OSGi erlauben die **parallele Nutzung** der gleichen Klasse(n) in verschiedenen **Versionen**. (Wird beispielsweise durch das Java's JPMS / Jigsaw System nicht unterstützt).

3. Komponenten: Hierarchie vs. "flach"

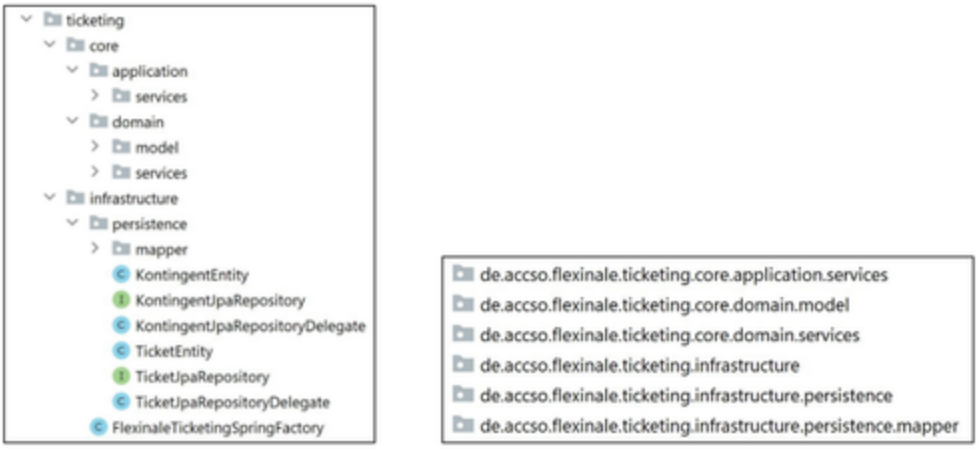


Abbildung: Darstellung von Packages in Java als Hierarchie vs. flach (ticketing-Komponente der Case-Study "Flexinale", Modulith-2) (Quelle: selbst erstellt)

Die Komponenten sind in der Regel in einer **Hierarchie** angeordnet, so dass kleinere Komponenten zu größeren kombiniert werden (ggf. rekursiv).

Dies kann ggf. nur ein Gedankenmodell sein, wenn der Komponentenrahmen dies nicht (wirklich) unterstützt, wie in diesen Beispielen:

- Java-Packages stehen nicht in einer Hierarchie. So ist das Paket "org.example.package" kein Unterpaket von "org.example". Alle Packages sind flach, werden jedoch typischerweise (in IDEs) als Hierarchie/Baum angezeigt.
- Das Gleiche gilt für OSGi-Module. Alle OSGi-Bundles sind flach im Bundle-Raum.

4. Zusammensetzung und Komposition zu größeren Einheiten

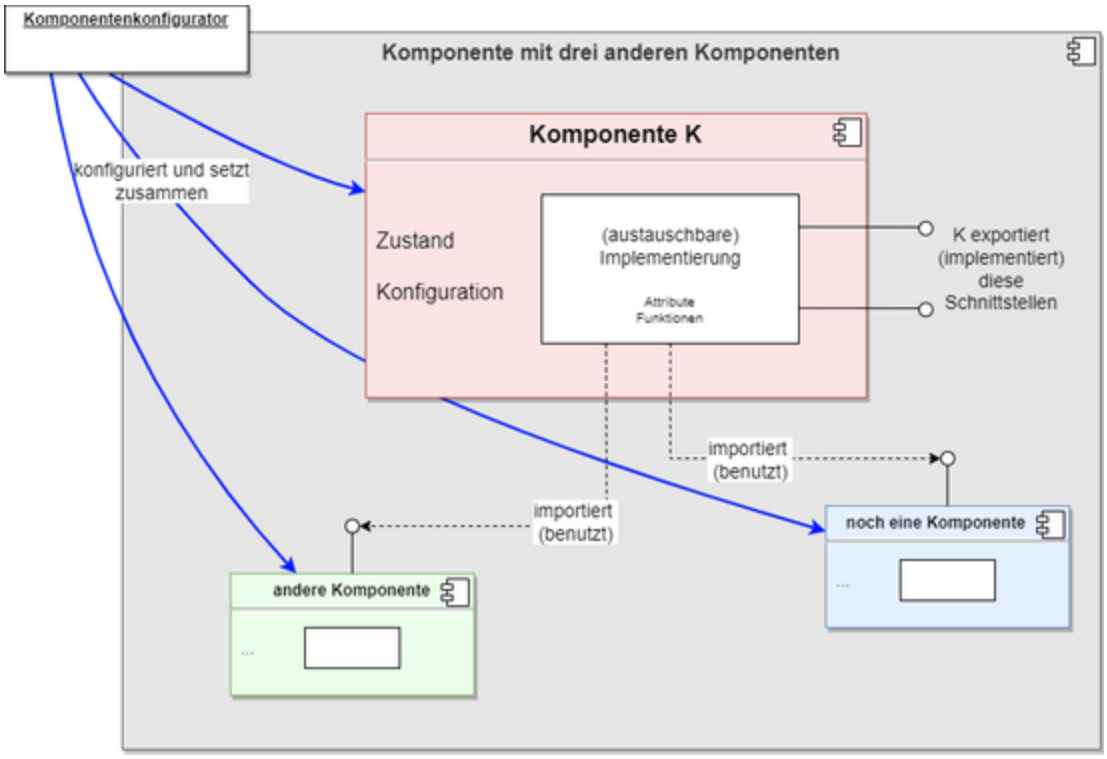


Abbildung: Konfiguration und Zusammensetzung mehrerer Komponenten (auch als Hierarchie)
(Quelle: selbst erstellt)

Die Zusammensetzung solcher Komponenten und Module zu größeren, lauffähigen Systemen, in denen alle Abhängigkeiten erfüllt sind, führt zu den **Definitionen von Systemen, Teil-/Subsystemen und Anwendungen**. Wir verwenden diese Begriffe, um ein lauffähiges, in sich geschlossenes "Ding" zu charakterisieren, das einen geschäftsorientierten Zweck erfüllt. Ein solches System kann auch - in Bezug auf die Verteilung - in einzelne Services unterteilt werden.

Damit ist die fachliche Sicht auf die Anwendung (-slandschaft) adressiert (vgl. Kapitel über DDD). Ein Bounded Context ist für eine "Fachlichkeit" zuständig und wird durch ein in sich geschlossenes System realisiert, das dann a) in einen oder mehrere Services und b) in eine Reihe von Komponenten zerfällt.

5. Arten von Komponenten-Konfiguration

Eine Softwarekomponente muss **konfiguriert** werden, um das Verhalten und die Einstellungen festzulegen, die sie haben soll, wenn sie ausgeführt wird. Es gibt verschiedene Gründe, warum eine Softwarekomponente konfiguriert werden muss, z. B:

- Anpassung an verschiedene **Umgebungen:** Eine Komponente muss ggf. anders konfiguriert werden, wenn sie in einer Entwicklungsumgebung läuft als in einer Produktionsumgebung.
- Verwendung unterschiedlicher **Ressourcen:** Eine Komponente muss ggf. so konfiguriert werden, dass sie andere Ressourcen verwendet, z. B. eine bestimmte Datenbank oder einen Web-Service.
- Ändern des **Verhaltens:** Eine Komponente muss unter Umständen so konfiguriert werden, dass sie ihr Verhalten je nach den Anforderungen des Systems oder des Benutzers ändert. Beispiel: Spracheinstellungen oder Länderspezifika.

Eine Softwarekomponente wird in der Regel zu verschiedenen Zeitpunkten ihres Lebenszyklus konfiguriert, unter anderem:

- Zur **Build-Zeit:** Die Komponente wird während des Build- oder Kompilierungsprozesses konfiguriert. Dies kann das Einrichten der Build-Umgebung, das Festlegen von Abhängigkeiten und das Konfigurieren der Komponenteneinstellungen umfassen.
- Zum Zeitpunkt des **Deployments:** Die Komponente wird bei der Installation und Deployment konfiguriert. Dies ist in der Regel für stage-spezifische Einstellungen wie Server/Connection- oder Benutzer-/Anmeldeinformationen erforderlich.
- Zum Zeitpunkt des **Starts:** Die Komponente wird konfiguriert, wenn sie gestartet oder initialisiert wird. Dies kann die Angabe der von der Komponente zu verwendenden Ressourcen, die Konfiguration der Komponenteneinstellungen und die Angabe der erforderlichen Anmeldedaten umfassen. **Dependency Injection z.B. per Spring** ist eine Technik zur Verwaltung von Abhängigkeiten zwischen Komponenten in einem System. Beispiel: Eine verwendete Komponente wird bei Tests durch einen Mock ersetzt.

- Zur **Laufzeit:** Die Komponente wird konfiguriert, während sie ausgeführt wird (ggf. auch mit dynamischen Änderungen, also Um-Konfigurationen). Dabei können die Einstellungen der Komponente geändert, neue oder aktualisierte Ressourcen bereitgestellt oder das Verhalten der Komponente geändert werden.

Es gibt verschiedene technische Umsetzungen der Konfiguration einer Softwarekomponente, darunter Properties, XML-, JSON-, YAML- usw. Dateien, Environment-Variablen, Datenbankinhalte oder auch Kommandozeilen-Argumente und auch in Kombination: Diese Konfigurationsvarianten überschreiben sich in der Regel in einer klar definierten Hierarchie.

6. Komponentenschnitt: Kopplung und Kohäsion

Kopplung und Kohäsion sind zwei wichtige Konzepte in der Softwaretechnik, die sich auf die Beziehung zwischen verschiedenen Teilen eines Softwaresystems beziehen.

Kopplung bezieht sich auf den Grad der gegenseitigen Abhängigkeit zwischen verschiedenen Teilen eines Systems. Ein System mit **hoher Kopplung** weist eine starke Beziehung zwischen den verschiedenen Teilen auf, so dass sich eine Änderung in einem Teil des Systems wahrscheinlich auf andere Teile des Systems auswirkt. Ein System mit **geringer Kopplung** weist eine schwache Beziehung zwischen den verschiedenen Teilen auf, so dass sich Änderungen in einem Teil nicht auf andere Teile auswirken. Solche lose gekoppelte Komponenten können geändert oder ersetzt werden, ohne dass dies Auswirkungen auf die anderen Komponenten des Systems hat.

Kohäsion bezieht sich auf den Grad, in dem verschiedene Teile eines Systems zusammenarbeiten, um einen einzigen, genau definierten Zweck zu erreichen. Ein System mit **hoher Kohäsion** hat Teile, die eng zusammenarbeiten, um ein bestimmtes Ziel zu erreichen. In einem System mit **niedriger Kohäsion** arbeiten dessen Teile unabhängig voneinander, verfolgen ggf. keinen klaren Zweck. Hohe Kohäsion ist der Grad, in dem die Elemente einer Komponente zusammenarbeiten, um einen einzigen, genau definierten Zweck zu erreichen. Eine hohe Kohäsion führt zu besser wartbarem und wiederverwendbarem Code.

Im Allgemeinen gilt ein Softwaresystem mit **loser, geringer Kopplung und hoher Kohäsion** als besser wartbar und leichter verständlich. Umgekehrt ist ein System mit hoher Kopplung und geringer Kohäsion schwieriger zu verstehen und zu ändern.

7. Komponentenschnitt: SRP (Single Responsible Principle) und SoC (Separation Of Concerns)

Das **SRP (Single Responsible Principle)** ist eines der SOLID-Prinzipien von Robert C. Martin. Es besagt, dass eine Komponente nur eine einzige Verantwortung oder Aufgabe haben sollte. Eine Komponente sollte nur für eine bestimmte Aufgabe geändert werden müssen.

Das SRP gibt es auch auf Klassen-Ebene. Es besagt dann analog, dass eine Klasse nur eine einzige Verantwortung haben sollte.

Eng verwandt damit ist das Prinzip **SoC (Separation Of Concerns)**". Es besagt: Verschiedene Aspekte oder "Concerns" einer Anwendung sollten in separate Abschnitte oder Komponenten aufgeteilt werden.

Die beiden Prinzipien sind verwandt, haben aber einen unterschiedlichen Fokus.

- Das SRP konzentriert sich auf die Kohäsion einzelner Komponenten (oder Klassen). Es fordert, dass jede Komponente (oder Klasse) nur eine Aufgabe hat, die klar definiert ist.
- Das SoC ist ein breiteres Konzept, das auf die Trennung verschiedener logischer oder funktionaler Aspekte einer Anwendung abzielt.

Eine andere, weitergehende Interpretation des SRP ist: Eine zusammengehörende Verantwortung oder Aufgabe soll auch nur in einer einzigen Kompnente (oder Klasse) abgebildet sein. Eine zusammengehörende Verantwortung oder Aufgabe soll also nicht über mehrere Kompnenten (oder Klassen) verteilt sein.

8. Komponentenschnitt: "Shallow Modules" vs. "Deep Modules"

Info
John Ousterhout: "A Philosophy of Software Design", ISBN-10: 1732102201

Video: https://www.youtube.com/watch?v=bmSAYlu0NcY

John Ousterhout führt eine weitere Charakterisierung über die "Tiefe" eines Moduls her (Modul ist hier als Synonym für Komponente zu verstehen, siehe 2.):

Shallow Modules verbinden große Schnittstellen und wenig Funktionalität.

Deep Modules haben kleine Schnittstellen und umfassende Funktionalität mit komplexen Operationen.

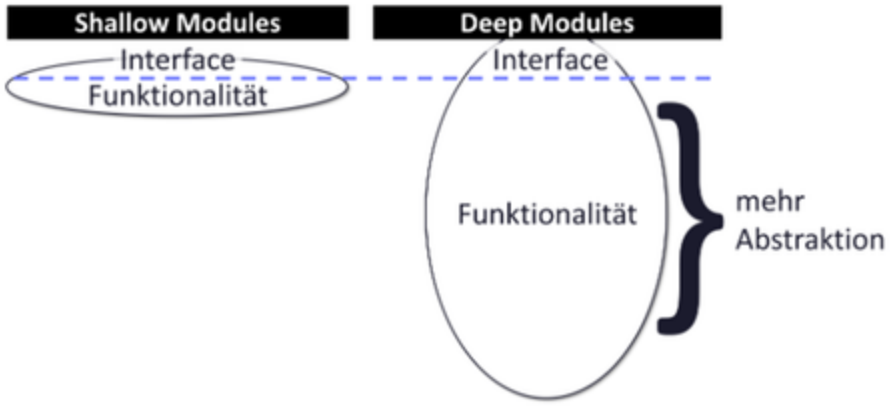


Abbildung: Shallow vs. Deep Modules (Module ist hier als Synonym für Komponente zu verstehen, siehe 2.)
(Quelle: John Ousterhout: "A Philosophy of Software Design")

Beim Komponentenschnitt sollte man auf einen angemessenen Grad der "Tiefe" anstreben, also ein angemessenes Mass an Abstraktion.

9. Komponentenschnitte v.a. an Fachlichkeit orientieren (und nicht an technischen Strukturen)

Ein System sollte auf der Grundlage **fachlicher Business-Anforderungen und -Domains** statt auf der Grundlage technischen Anforderungen modelliert werden, denn:

1. Fachlichkeit (fachliche Anforderung) ändert sich öfter und schneller als Technik. ("Business Alignment hat Priorität, es bezahlt unseren Job!")
2. Business Alignment: Die fachliche Modellierung eines Systems ermöglicht eine bessere Abstimmung zwischen dem System und den Geschäftsanforderungen. Dadurch wird es einfacher, das System zu verstehen und Entscheidungen darüber zu treffen, wie das System weiterentwickelt werden soll, um veränderten Geschäftsanforderungen gerecht zu werden. Fachlichkeit ändert sich öfter und ist schwieriger nachzuverfolgen.
3. Unabhängige Weiterentwicklung: Durch eine solche Modellierung eines Systems wird es möglich, das System unabhängig weiterzuentwickeln. Dies ermöglicht die Einführung neuer Funktionen und Fähigkeiten, ohne dass das gesamte System geändert werden muss.
4. Flexibilität: Dies ermöglicht eine größere Flexibilität bei Deployment und Betrieb. Es kann auch die Skalierung, den Einsatz und den Betrieb des Systems sowie die Erfüllung sich ändernder Geschäftsanforderungen erleichtern.
5. Domänenbasiertes Testen: Eine solche Modellierung ermöglicht die Entwicklung von Testfällen, die sich auf bestimmte Geschäftsdomänen konzentrieren, was das Test und Fehleranalyse erleichtert.
6. Beherrschung der Komplexität: Es ermöglicht die Beherrschung der Komplexität, indem das System in kleinere, besser handhabbare Teile aufgeteilt wird. Dadurch wird es einfacher, das System zu verstehen, zu testen und zu ändern.

10. Domain-driven Design, Bounded Contexts, Aggregates

Info
https://speakerdeck.com/mrtnlhmnn/ddd-als-best-practice-fur-datenmodellierung-von-microservices

https://accso.de/app/uploads/2019/12/Masterarbeit_Marcus-Franz_20191004_Domain-Driven-Design_DDD.pdf

Michael Plöd. "Hands-on Domain-driven Design - by example. Domain-driven Design practically explained with a massive case study". Leanpub, 2019

Domain-driven Design (DDD) ist ein Ansatz für die Softwareentwicklung, der sich auf die Problemdomäne oder den spezifischen Unternehmensbereich konzentriert, den die Software unterstützen soll. Das Ziel von DDD ist es, das Softwaredesign an der Geschäftsdomäne auszurichten, wodurch es sowohl für Entwickler als auch für Domänenexperten genauer und leichter verständlich wird.

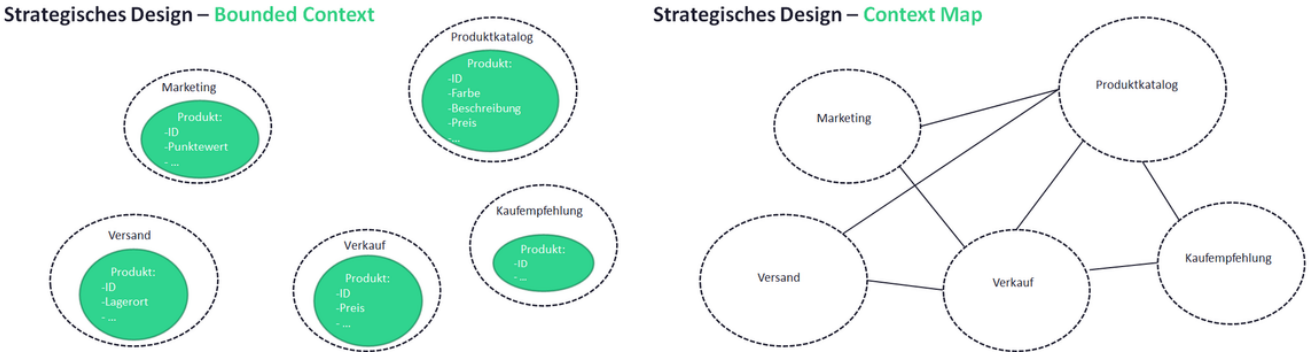
DDD besteht aus drei sich ergänzenden Techniken:

1. **Strategisches Design,** siehe unten
2. **Taktisches Design** mit Entities, Value Objects, Repositories, Aggregates etc.
3. Kollaborative **Modellierungstechniken** wie Event Storming und Domain Storytelling.

DDD ist keine Methodik oder ein Regelwerk, sondern eine Reihe von Prinzipien und Praktiken, die die Softwareentwicklung leiten und die Fachlichkeit in den Kern und Vordergrund des Designprozesses stellen.

Zu den Kernkonzepten von DDD gehören:

1. **Domänenmodell:** Dies ist das Kernkonzept von DDD und bezieht sich auf die Darstellung der Problemdomäne in der Software. Das Domänenmodell ist eine Sammlung von Objekten, die die Entitäten, Dienste und Regeln der Problemdomäne darstellen.
2. **Ubiquituous Language:** Eine Ubiquituous Language ist eine gemeinsame Sprache, die sowohl von Entwicklern als auch von Domänenexperten verwendet wird, um die Problemdomäne zu beschreiben. Es ist wichtig, eine einheitliche Sprache zu haben und die gleichen Begriffe zu nutzen, um sicherzustellen, dass das Softwaredesign die Geschäftsdomäne genau widerspiegelt und von allen leicht verstanden werden kann.
3. **Bounded Context:** Ein Bounded Context ist eine Möglichkeit, Domänenobjekte zu gruppieren, die unterschiedliche Bedeutungen, Zwecke oder Anwendungsfälle haben, und sie von anderen Domänenobjekten zu trennen, die andere Bedeutungen haben. Auf diese Weise können je nach Verwendungskontext unterschiedliche Modelle für ein und denselben Begriff verwendet werden.
   Ein Bounded Context stellt eine semantische, kontextuelle Grenze einer Ubiquitous Language dar. Jeder Begriff innerhalb eines Bounded Context hat, entsprechend der Ubiquitous Language, seine eigene Bedeutung und Struktur, was sich in einem eigenen Domänenmodell ausdrückt, welches durch den Bounded Context vor anderen Modellen isoliert wird. Ein Bounded Context definiert demnach mit fachlicher "Brille" die verschiedenen Teile des Systems, die ein eigenes und isoliertes Verständnis der Geschäftsdomäne haben. Jeder Bounded Context hat sein eigenes Vokabular und seine eigenen Modelle, die für diesen Kontext spezifisch sind.
4. Eine **Context Map** stellt eine abstrakte Sicht auf die bestehenden Bounded Contexts und deren Beziehungen untereinander dar. Eine Context Map ist aber auch gut dafür geeignet, um eine bestehende Anwendungslandschaft zu analysieren oder eine neue zu planen. Sie hilft dabei einen Überblick zu bekommen, wie ein Domänenmodell sich in der Anwendungslandschaft ausbreitet (Model Flow), welcher Bounded Context welchen anderen aufruft (Call Flow) und welcher Bounded Context Einfluss auf andere nimmt (Influence Flow).
   Eine Context Map ist demnach eine visuelle Darstellung der verschiedenen Teile eines Systems und ihrer Interaktion miteinander in Bezug auf die Geschäftsdomäne. Sie zeigt die Grenzen der verschiedenen Kontexte und die Beziehungen zwischen ihnen.
   Die Context Map hilft bei der Identifizierung der Grenzen der verschiedenen Kontexte und bei der Identifizierung von Bereichen des Systems, die möglicherweise mehr Aufmerksamkeit oder Modellierung benötigen. Sie hilft auch dabei, Bereiche des Systems zu identifizieren, die stabiler sind und sich weniger wahrscheinlich ändern, sowie Bereiche, die unbeständiger sind und sich wahrscheinlich ändern werden.



Abbildungen: Bounded Contexts und Context Map einer eCommerce-Anwendung
(Quelle: Marcus Franz, Martin Lehmann, Dr. Renato Vinga-Martins: "DDD als Best Practices für Datenmodellierung und Microservices", https://speakerdeck.com/mrtnlhmnn/ddd-als-best-practice-fur-datenmodellierung-von-microservices)

Bei der fachlichen Strukturierung und dem Finden der **fachlichen Grenzen** hilft eine Orientierung an diesen Eigenschaften:

- Welche Akteure (Nutzer, Nutzergruppen) arbeiten mit dem System?
- Welche Aktionen (Nutzungsszenarien) führen sie aus (und wieoft, wie kritisch, mit welchem Ziel, nur lesend / auch schreibend, ...)?
- Welche Daten müssen dazu erstellt / geschrieben benutzt werden (Sicht auf Entitäten, Blick auf Datenhoheit)?
- Welche Daten müssen dazu zusätzlich benutzt werden (benutzt, i.d.R. rein lesend, als Kopie von anderen Strukturen)?

Im Taktischen Design von DDD zählen insbesondere diese Patterns:

- **Entität**: Ist ein "Ding", ist über eine ID identifizierbar, hat einen Lebenszyklus (wird erzeugt - verändert - gelöscht). Beispiel: Produkt, Buch, Bankkonto
- **Value Object**: Repräsentiert einen immutablen Wert, der durch seine Eigenschaften definiert wird - aber keine eigene Identität besitzt. Beispiel: Produktkennung, ISBN, IBAN, Geldbetrag
- Unterscheide:
  - ID einer Entität: Wird verwendet, um die Identität einer Entität eindeutig zu bestimmen
  - Value Object wird durch seine Attribute definiert.
  - Entitäten haben eine eindeutige Identität (= ID), über die sie von anderen Entitäten unterscheidbar ist, selbst wenn alle (anderen, fachlichen) Attribute gleich sind
  - Dagegen sind Value Objects immer dnn gleich, wenn alle ihre Attribute gleich sind.

Beim Auffinden von Strukturen hilft auch das DDD-Pattern "**Aggregate**", mit denen Konsistenz- und Validierungsanforderungen klar(er) werden:

- Ein Aggregate ist eine fachliche "Klammer" um eine Menge von Entitäten (im Sinn eines "Containers").
- Ein Aggregate selbst ist eine Entität, hat also ID und Lebenszyklus.
- Ein Einstieg zum Lesen/Schreiben erfolgt immer über eine "Root-Entity". Beispiel eCommerce: Warenkorb und Warenkorb-Positionen.
- Ein Aggregate erlaubt es, an zentraler Stelle Fragen der Konsistenz bzw. fachliche Validierungen zu überprüfen. Beispiel eCommerce: Mindestbestellwert, Maximalbestellwert, maximale Anzahl von Produkten. Beispiel Flexinale: Ticketbundle, damit keine Ticket-Vorführungen "überlappen".

Ein Aggregate ist ein guter Indikator, um fachliche Grenzen zu finden und zu modellieren und wird sich in technischen Komponenten/Service-Strukturen (Konsistenz, Datenhaltung, Datenhoheit) niederschlagen (da i.d.R. Anforderungen an die Konsistenz nicht sinnvoll über deren Grenzen hinweg umgesetzt werden, ein Aggregate also immer *in* einer/m solchen Komponente/Service leben muss und nicht verteilt werden kann.

11. Komponentenschnitt: Quasar und AT-Trennung

Info
Siehe https://dpunkt.de/produkt/moderne-software-architektur/

Siehe https://projects.fbi.h-da.de/~b.humm/pub/Siedersleben_-_Quasar_1__sd_m_Brosch_re_.pdf

und https://projects.fbi.h-da.de/~b.humm/pub/Siedersleben_-_Quasar_2__sd_m_Brosch_re_.pdf

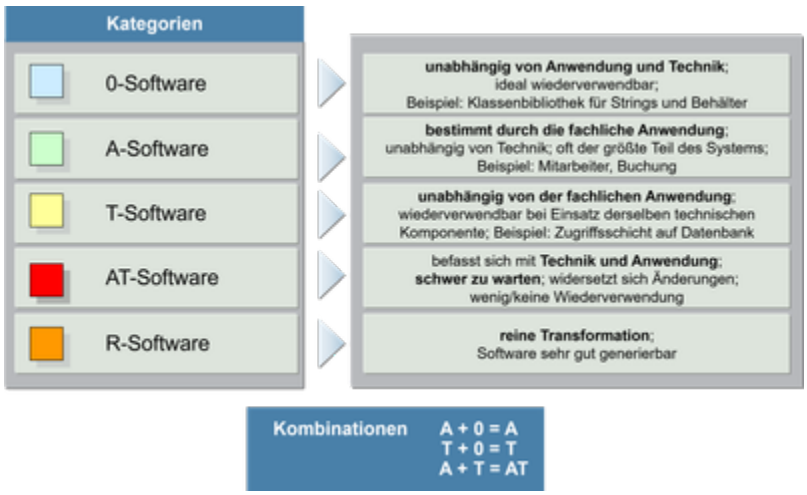


Abbildung Software-Kategorien ("Blutgruppen")
(Quelle: Johannes Siedersleben: "Moderne Software-Architektur", dpunkt.verlag, 2004)

Quasar ist ein Architekturkonzept, das federführend von Johannes Siedersleben bei sd&m entwickelt wurde. Es betont einige Aspekte von komponenten-orientierter Architektur:

a) Trennung von **Fachlichkeit** und **Technik,** Kategorisierung von Softwarebausteinen in sogenannte Blutgruppen als A-, T- und 0-Software:

- 0-Software ist unabhängig von Anwendung und Technik. Beispiele sind Klassenbibliotheken, die sich mit Strings und Behältern befassen (Beispiele: Java Collections, C++ Standard Template Library). Sie ist ideal wiederverwendbar, für sich alleine aber ohne Nutzen.
- A-Software ist bestimmt durch die Fachlichkeit der Anwendung, aber unabhängig von der Technik. (Beispiele: Entitäten wie "Kino", "Film" oder Use-Cases wie "Gutschein-Einlösen")
- T-Software ist unabhängig von der Anwendung, bestimmt durch die Technik (Beispiele: Hibernate, JPA, JDBC, Kafka).
- AT-Software ist bestimmt durch Anwendung *und* Technik: Diese Mischform ist schwer zu warten, denn sie "widersetzt" sich Änderungen. Sie hat verschiedene Änderungszyklen bzw. hat mehr als einen Grund, sich zu ändern, kann daher kaum wiederverwendet werden. AT-Softwar ist daher zu vermeiden.

b) **Komponentenbegriff:**

- Definition von Software-Komponenten als sinnvolle Einheit des Entwurfs, der Implementierung und damit der Planung.
- Definition verschiedener Typen von Schnittstellen

c) **Standard-Referenzmodell für Business-Informationssysteme** bestehend aus typischen Komponenten und Layern (im Sinne von A, T und TI inkl. entsprechender Architektursichten)

Quasar betont mit der AT-Trennung besonders die Trennung von Fachlichkeit und Technik. ImSinne von SoC (Separation of Cpncerns) heißt das: Fachlichkeit und Technik sind immer unterschiedliche "Concerns".

Auch innerhalb einer Komponente ist die AT-Trennung ein wichtiges Prinzip. Innerhalb einer fachlichen Komponente lassen sich technische Aufgaben nicht vollständig vermeiden, in dem Sinne, dass "Technik" angesprochen werden muss, z.B. eine Datenbank oder ein Eventbus. Klassen, die sich darum kümmern, sollten aber immer getrennt sein von den Klassen die sich um fachliche Aspekte kümmern. Siehe bspw. die Onion-Architektur in 3C, 2.: Für Technik ist ausschließlich der äußerste Ring zuständig.

12. Abhängigkeiten zu Build-Zeit, Startzeit, Laufzeit

In der Softwareentwicklung können **Abhängigkeiten** zwischen Komponenten eingeteilt werden bzgl. Build-/Kompilierzeit, Startzeit und Laufzeit:

1. Abhängigkeiten zur **Build-Zeit**: Dies sind Abhängigkeiten, die für die Kompilierung einer Komponente erforderlich sind. So kann eine Komponente beispielsweise von einer bestimmten Version einer Programmiersprache oder eines Frameworks oder von einer bestimmten Bibliothek oder einem bestimmten Werkzeug abhängig sein.
2. **Startzeit**-Abhängigkeiten: Dies sind Abhängigkeiten, die zum Start oder Initialisieren einer Komponente erforderlich sind. Eine Komponente kann von anderen Komponenten oder Diensten abhängig sein, die zum Startzeitpunkt vorhanden oder ausgeführt werden müssen. Beispiel: Eine Java-Komponente ist zur Build-Zeit nur von einer Logging-Abstraktion wie slf4j abhängig, die aber zur Startzeit durch eine konkrete Implementierung implementiert werden muss.
3. **Laufzeit**-Abhängigkeiten: Dies sind Abhängigkeiten, die erforderlich sind, damit eine Komponente während des normalen Betriebs korrekt läuft. So kann eine Komponente von anderen Komponenten oder Diensten abhängig sein, mit denen sie während des normalen Betriebs kommuniziert.

Vgl. auch Maven's Ebenen der Dependency-Scopes (compile, runtime, test, provided).

13. Early vs. Late Binding

Info
Beispiel für Java: https://www.geeksforgeeks.org/difference-between-early-and-late-binding-in-java/

In der Objektorientierten Programmierung werden die Begriffe **"late binding" und "early binding"** verwendet, um den Zeitpunkt zu beschreiben, zu dem die Bindung der Verweise eines Objekts an die Speicheradressen der Objekte, auf die sie verweisen, aufgelöst wird.

- **Early Binding,** auch als statisches Binding oder Compile-Time-Binding bezeichnet, bezieht sich auf den Prozess der Auflösung von Objektreferenzen zur Kompilierzeit. Bei der frühen Bindung werden die Objektreferenzen aufgelöst, wenn der Code kompiliert wird, und der daraus resultierende ausführbare Code enthält die Speicheradressen der Objekte, auf die sich die Referenzen beziehen. Dies bedeutet, dass sich die Objektreferenzen zur Laufzeit nicht ändern können und der kompilierte Code effizienter ausgeführt werden kann, da die Referenzen bereits aufgelöst wurden.
- **Late Binding,** auch dynamisches Binden oder Laufzeitbindung genannt, bezieht sich auf den Prozess der Auflösung von Objektreferenzen zur Laufzeit. Bei der späten Bindung werden die Objektreferenzen erst bei der Ausführung des Codes aufgelöst. Das bedeutet, dass sich die Referenzen zur Laufzeit ändern können und der Code auf eine allgemeinere Weise ausgeführt werden muss, um diese Möglichkeit zu berücksichtigen. Die späte Bindung wird auch als dynamische Bindung oder Laufzeitbindung bezeichnet, da sie die referenzierten Objekte zur Laufzeit ändern kann.

Im Allgemeinen ist die frühe Bindung effizienter, weil die Referenzen zur Kompilierzeit aufgelöst werden, aber weniger flexibel, weil sich die Referenzen zur Laufzeit nicht ändern können. Die späte Bindung ist umgekehrt flexibler, weil sich die Referenzen zur Laufzeit ändern können, aber weniger effizient, weil die Referenzen zur Laufzeit aufgelöst werden müssen.

14. Law of Leaky Abstraction

Info
https://www.joelonsoftware.com/2002/11/11/the-law-of-leaky-abstractions/

Das **Law of Leaky Abstraction** ist ein Programmierkonzept von Joel Spolsky, das besagt, dass "*alle nicht-trivialen Abstraktionen bis zu einem gewissen Grad 'durchlecken' und undicht sind*". Wenn also ein Programmierer die Abstraktionsebene in seinem Code erhöht, beginnen die Implementierungsdetails dieser Abstraktion durch die Schnittstelle "durchzusickern". Dies gilt insbesondere, wenn sich das Verhalten ändert.

Das bedeutet, dass ein Programmierer bei der Verwendung von Abstraktionen auf höherer Ebene immer noch die zugrundeliegenden Implementierungsdetails verstehen und mit ihnen arbeiten muss, um den Code ordnungsgemäß verwenden und Fehler beheben zu können. Das Gesetz unterstreicht, dass Abstraktionen nie perfekt sind und dass immer ein gewisser Grad an Komplexität durchsickern wird. Es ist wichtig, sich beim Schreiben und Verwenden von Abstraktionen des Gesetzes des Durchsickerns von Komplexität bewusst zu sein, da es dazu beitragen kann, unerwartetes Verhalten und Fehler im Code zu vermeiden.

Beispiele:

- Eine Netzwerkschicht versucht, eine vollständige Abstraktion eines zugrunde liegenden unzuverlässigen Netzwerks zu bieten, aber manchmal "leckt"das Netzwerk durch die Abstraktion hindurch und man "spürt die Dinge", die die Abstraktion nicht ganz abschirmen kann. Eine Netzwerkkommunikation, die über einen Proxy oder Stub erfolgt, fühlt sich an und sieht so aus, als würde man mit einem lokalen Objekt oder Dienst sprechen, ist es aber tatsächlich nicht: Die Remote-Kommunikation bringt verschiedene Probleme mit sich, wie z.B. Latenzzeiten, Netzwerkfehler, Probleme mit Zustellungsgarantien (Nachrichten gehen verloren, werden dupliziert usw.) und die Performance/Dauer der Anfrage selbst.
- Ein Web-Framework kann eine High-Level-Schnittstelle für die Bearbeitung von HTTP-Anfragen bieten, aber der Entwickler muss trotzdem zugrundeliegende Details des HTTP-Protokolls verstehen, um das Framework richtig zu nutzen.
- Im Flexinale-Code wird die EventBus-Schnittstelle durch eine lokale Variante ("InMemorySyncEventBus") und ("KafkaAsyncEventBus") implementiert. Diese sehen hinter ihrer gemeinsamen Schnittstelle gleich aus, funktionieren aber sehr unterschiedlich: a) wegen der lokalen ! = Remote-Kommunikation, siehe oben und b) wegen der synchronen vs. asynchronen Aufrufe.

15. Dokumentation

Info
https://arc42.org/

https://github.com/feststelltaste/software-component-canvas

https://canvas.arc42.org

Der De-facto-Standard für **Architekturdokumentation** ist **arc42** - ein weit verbreiteter, umfassender Dokumentationsstandard für Softwarearchitektur. arc42 bietet eine Vorlage für Organisation und Strukturierung von Architekturdokumentation, die verschiedene Aspekte der Softwarearchitektur abdeckt, wie z. B. Stakeholder, Anforderungen und technische Entscheidungen.

Abbildung: arc42-Logo

- Komponenten werden normalerweise im Abschnitt "Building Blocks" dokumentiert, der Teil des Kapitels "Solution Strategy" ist. Dieser Abschnitt gibt einen Überblick über die Hauptkomponenten der Software und ihre Beziehungen, einschließlich ihrer Verantwortlichkeiten, Schnittstellen und Abhängigkeiten.
- Er dokumentiert auch alle relevanten Einschränkungen und Annahmen zu den Komponenten sowie alle technischen Entscheidungen, die zu deren Design und Implementierung getroffen wurden.
- Vgl. auch weitere Abschnitte 6 ("Laufzeitsicht") und 7 ("Deployment-Sicht").

Eine Softwarekomponente kann durch einen "**Component Canvas**" dokumentiert werden. Dieser Canvas gibt einen guten Überblick über die einzelnen Komponenten. Als Werkzeug wird es gerne verwendet, um die Hauptkomponenten eines Softwaresystems zu dokumentieren. Es handelt sich dabei um eine Abwandlung des "Business Model Canvas" der Unternehmensplanung und -strategie. Das Software Component Canvas ist eine Vorlage, die dabei hilft, die Dokumentation über die Komponenten eines Softwaresystems zu organisieren und zu strukturieren. Der Software-Component-Canvas ist demnach nützlich für Dokumentation, Kommunikation und Analyse von Software-Architektur und hilft sicherzustellen, dass alle relevanten Informationen über eine Komponente in einer strukturierten und konsistenten Weise erfasst werden.

Ein solcher Canvas ist in 9 Abschnitte unterteilt, die einen Überblick über die Hauptkomponenten der Software, ihre Beziehungen und den Kontext, in dem sie arbeiten, geben:

1. **Komponente**: Name der Komponente, ggf. Synonyme
2. **Zweck**: Hauptzweck und Verantwortlichkeiten der Komponente
3. **Schnittstellen**: die für extern bereitgestellten Schnittstellen und APIs der Komponente
4. **Abhängigkeiten**: andere Komponenten, von denen die Komponente abhängt (ggf. weitere Details wie Art, Zeitpunkt)
5. **Einschränkungen**: alle Einschränkungen der Komponente
6. **Annahmen**: alle Annahmen, die über die Komponente getroffen wurden
7. **Technische Entscheidungen**: alle technischen Entscheidungen, die über den Entwurf und die Implementierung der Komponente getroffen wurden
8. **Qualitätsattribute**: alle nicht-funktionalen Anforderungen, die die Komponente erfüllen muss
9. **Risiken**: alle mit der Komponente verbundenen Risiken

Neben der Architektur selber dokumentiert man auch wichtige **Architekturentscheidungen**. Dabei soll nicht nur die Entscheidung selber dokumentiert werden, sondern auch die Begründung für die Entscheidung. Außerdem werden auch die untersuchten Alternativen sowie die Gründe, warum sie verworfen wurden, dokumentiert. Das hilft nicht nur beim Verständnis, sondern verhindert auch, das immer wieder über Alternativen diskutiert wird, die längst verworfen wurden.

Für die Dokumentation einer Architekturentscheidung hat sich das **Architecture Decision Record (ADR)** etabliert. Ein ADR enthält in der Regel

1. **Überschrift:** Eine kurze, prägnante Überschrift, die die Architekturentscheidung beschreibt.
2. **Kontext:** Eine Beschreibung des Kontexts oder des Problems, das die Notwendigkeit für diese Architekturentscheidung ausgelöst hat. Dies kann technische Anforderungen, Geschäftsanforderungen oder andere relevante Faktoren einschließen.
3. **Entscheidung:** Die eigentliche Architekturentscheidung und ihre Begründung. Hier wird erläutert, warum die gewählte Architekturlösung die beste Option ist.
4. **Untersuchte Alternativen**: Welche Alternativen wurden untersucht und warum wurden sie verworfen.
5. **Status:** Der Status des ADRs, der angibt, ob die Entscheidung bereits umgesetzt wurde, noch aussteht oder ob sie überholt ist.
6. **Konsequenzen:** Eine Beschreibung der erwarteten Auswirkungen der getroffenen Entscheidung auf das System, auf andere Komponenten oder auf die Entwicklung des Projekts.
7. **Verweise:** Hier können Verweise auf andere ADRs, externe Dokumentation oder Ressourcen angegeben werden, die zur Unterstützung der Entscheidung herangezogen wurden.

Für die Architekturentscheidungen gibt es auch ein eigenes Kapitel im arc-42-Standard.

(C) Modularisierung in der Innensicht: Design & Mikro-Ebene

1. Patterns

Es gibt verschiedene Modularisierungspatterns, die für den Entwurf und die Entwicklung von Softwaresystemen verwendet werden können. Einige gängige Beispiele sind:

- **Layered** Pattern: Bei diesem Pattern wird das System in eine Reihe von Layern gegliedert, wobei jedes Layer für einen bestimmten Funktionsumfang zuständig ist (der typischerweise technisch motiviert ist). Die Schichten sind in der Regel auf der Grundlage des Abstraktionsgrads organisiert, wobei niedrigere Schichten einfachere Funktionen und höhere Schichten komplexere Funktionen bieten.
- **Plug-in**-Pattern: Mit diesem Pattern können neue Funktionen zu einem System hinzugefügt werden, indem neue "Plug-ins" erstellt und dem System hinzugefügt werden. Diese Plug-ins sind in der Regel so konzipiert, dass sie unabhängig und austauschbar sind, so dass sie dem System je nach Bedarf hinzugefügt oder entfernt werden können.
- **Mikrokernel-Pattern:** Bei diesem Pattern wird das System um einen kleinen, zentralen Kernel herum organisiert, der grundlegende Funktionen bereitstellt und es ermöglicht, zusätzliche Funktionen durch die Verwendung von "Modulen", die in den Kernel eingesteckt werden, hinzuzufügen.
- **Service-orientiertes** Pattern: Bei diesem Pattern wird das System als eine Sammlung unabhängiger Dienste organisiert, die über genau definierte APIs miteinander kommunizieren. Jeder Dienst ist für eine bestimmte Geschäftsfähigkeit oder Funktion verantwortlich und kann unabhängig von den anderen Diensten entwickelt, bereitgestellt und skaliert werden.
- **Event-driven** Pattern: Bei diesem Pattern wird das System um eine Reihe von Ereignissen herum organisiert, wobei die verschiedenen Teile des Systems bestimmte Ereignisse "konsumieren", auf sie reagieren und diese verarbeiten. Dieses Pattern ist Basis ereignisgesteuerter Architekturen und Systeme, die asynchrone Vorgänge verarbeiten müssen.

Von der Schichten-Architektur zur Onion-Architektur

In einer "klassischen" Schichten-Architektur (Layered Pattern) wird nach technischen Schichten geschnitten. Ein Request, bzw. der Workflow beginnt in der API-Schicht, geht durch die Schichten der Business Logik (in der Abbildung: Servces Layer und Domain Layer) bis zur Datenbank und den umgekehrten Weg zurück, s. Abbildung, linke Seite. Die Abbildung rechts zeigt eine andere Darstellung, hier sind die Schichten entlang des Workflows dargestellt. In dieser Darstellung sind die Schichten aber vermischt, gerade technische (Infrastruktur-) Schichten und fachliche. Insbesondere steht hier die Datenbank als ein Stück Infrastruktur (unschön) im Zentrum.

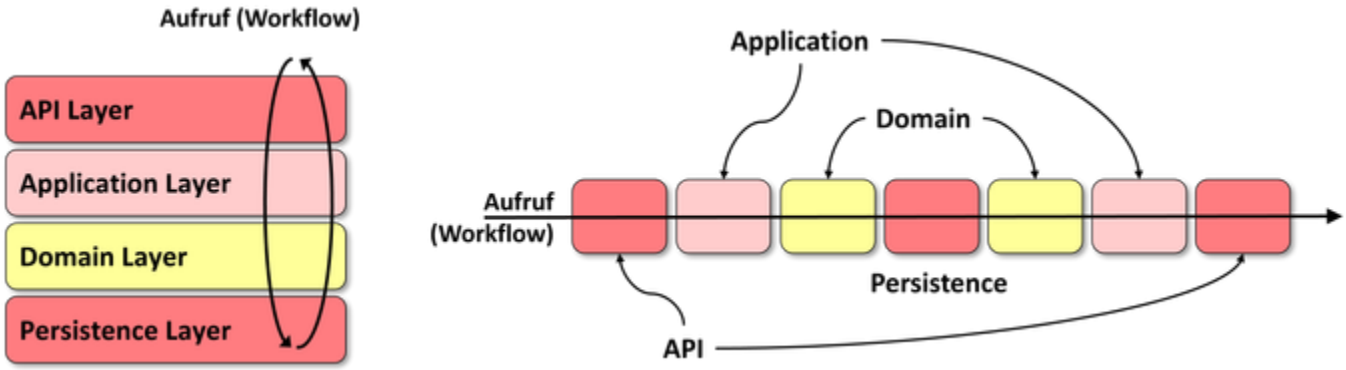


Abbildung: Schichten und Workflow
(Quelle: Selbst erstellt. Nach Scott Wlaschin: Domain Modeling Made Functional: Tackle Software Complexity with Domain-Driven Design and F#, Kapitel 3)

Man möchte die Domänen-Logik jedoch ins Zentrum verschieben und die Infrastruktur an den Rand. So ergibt sich eine domämemzentrierte Struktur, eine Onion-Architektur:

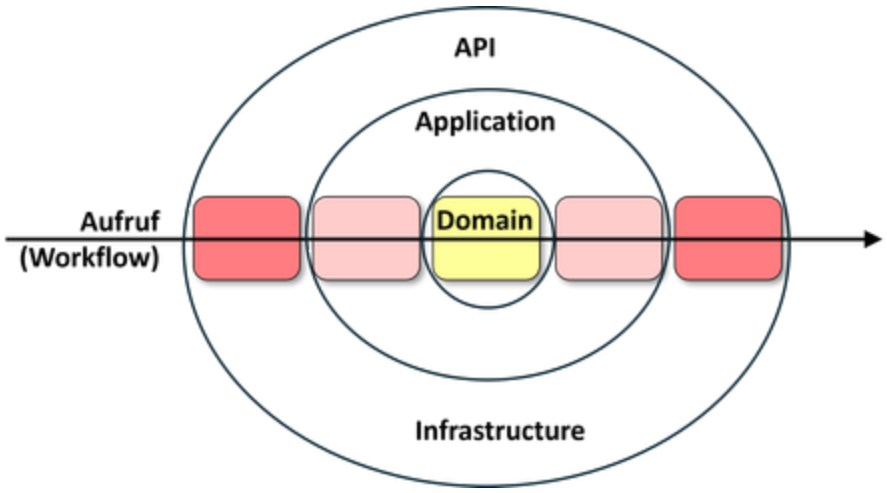


Abbildung: Domänenlogik im Zentrum: Onion-Architektur
(Quelle: Selbst erstellt. Nach Scott Wlaschin: Domain Modeling Made Functional: Tackle Software Complexity with Domain-Driven Design and F#, Kapitel 3)

2. Onion Architektur

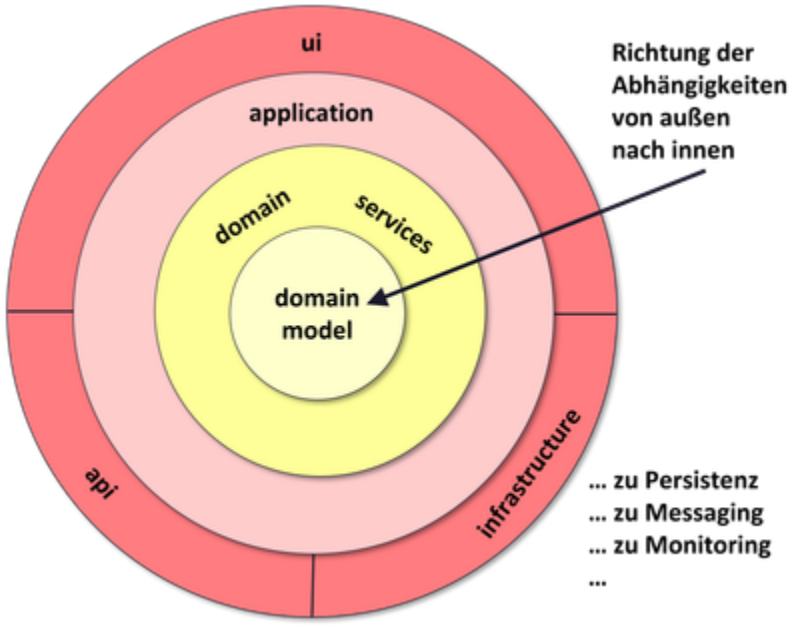


Abbildung: Onion-Architektur - Innensicht einer Komponente
(Quelle: selbst erstellt)

Info
https://speakerdeck.com/mrtnlhmnn/automatisierte-architekturtests-und-statische-code-analyse-mit-archunit

https://medium.com/expedia-group-tech/onion-architecture-deed8a554423

Christian Stettler, "DDD mit Onion Architecture & Stereotypes. Die Applikationsarchitektur für Domain-driven Design", Talk bei der JUG Switzerland, 13. März 2019, https://www.jug.ch/html/events/2019/ddd_mit_onion_architecture_und_stereotypes_be.html

Scott Wlaschin: Domain Modeling Made Functional: Tackle Software Complexity with Domain-Driven Design and F#, O'Reilly Media

Die **Onion-Architektur** ermöglicht eine flexible, nachhaltige und übertragbare Architektur innerhalb einer Komponente. Sie teilt eine Komponente in mehrere, konzentrisch angeordnete Layer / (Zwiebel)Schichten, die in Richtung des Kerns (der Domäne) miteinander interagieren. Die

Architektur hat die Datenhaltung nicht "unten" / "innen", wie bei einer traditionellen dreistufigen Architektur, sondern hat ein echtes Domänenmodell im Kern. Kernidee ist es, externe Abhängigkeiten so weit wie möglich nach außen zu verlagern und den Kern von externen Abhängigkeiten unabhängig zu halten.

- Die Domänenlogik steht mit Domain Entities und Domain Services im Zentrum.
- In application-Ring liegen Use Cases bzw. Abläufe, die die Domänen-Logik zusammenstecken. Außerdem liegen hier technische Aspekte wie Transaktionen, Authentifizierung, Logging, etc.
- Der äußere Ring ist in mehrere Sektoren zerteilt. In infrastructure liegen Zugriffe auf Infrastruktur wie Persistenz, Monitoring, Messaging. Dort befindet sich auch die API, die GUI u.ä.

Diese Regeln gelten:

- Die **Richtung** der (statischen) Abhängigkeiten ist von **außen nach innen**: Äußere Ringe dürfen von inneren abhängen, andersherum jedoch nicht. Insbesondere ist die Domänen-Logik frei von Abhängigkeiten zur Infrastruktur.
- Die Ringe sind per **Dependency Inversion Principle** entkoppelt.
- Jeder Ring hat seine eigene Sicht auf die Daten.

Hinweis: Der äußerste Ring besteht aus unabhängigen Sektoren, die Regeln müssen entsprechend auf die einzelnen Sektoren angewendet werden.

Einige **Details** zur Onion-Architektur und den Regeln:

Überspringen von Ringen: Strikte Auslegungen besagen, dass Ringe nicht übersprungen werden dürfen. Das heißt, dass ein Ring nur vom nächsten benachbarten Ring weiter innen abhängen darf, z.B. api nur von application. Das führt aber dazu, dass wirklich jeder Ring eine eigene Repräsentation der Daten benötigt und das führt zu sehr viel Mapping-Code. Gerade in application stellt sich in der Praxis schnell die Frage, welchen Mehrwert eine eigene Repräsentation der Daten bringt. Daher geht man hier oft pragmatisch vor. Beispielsweise ist es in Ordnung, wenn das domain model in der api bekannt ist, zumindest als Rückgabewerte von Services, die in application angesiedelt sind. Diese dürfen Klassen des domain models als Rückgabewert an die api haben. Die api ist dann dafür verantwortlich, diese Daten in die passenden DTOs (Data Transfer Objects) für den Aufrufer zu mappen. Allerdings dürfen Objekte des domain models in api nur lesend verwendet werden. Daher sollte application nur "immutable" Daten oder Kopien von Objekten des domain models zurückgeben.

Auch in der Persistenz, die Teil von infrastructure ist, stellt sich die gleiche Frage. Auch hier ist es pragmatisch, wenn das domain model genutzt werden darf. Aber auch hier ausschließlich, um die aus der Persistenz, z.B. Datenbank, gelesenen Entitäten in das domain model zu mappen, bzw. andersherum für die Persistierung.

Transaktionskontrolle immer in application: Transaktionskontrolle (also öffnen und schließen einer Transaktion) wird häufig "so weit außen wie möglich" in einer Anwendung verortet. Das wäre im Fall der Onion-Architektur die api, bspw. die Controller in der api. Die Begründung für das Vorgehen ist, dass sämtliche Änderungen an Daten, die in der Anwendungslogik (also hier der application und domain) durchgeführt werden, auch wirklich innerhalb der Transaktionskontrolle (und im Java- und Spring/Hibernate-Stack die Session) liegen und von diesen erkannt werden. Ein anderer technischer Grund ist das Lazy Loading von Referenzen, was nur innerhalb einer geöffneten Hibernate-Session korrekt funktioniert.

Dagegen spricht jedoch einiges: Es kann viele Schnittstellen geben, Controller sind nur eine Möglichkeit. Wenn sich die Transaktionslogik im Controller befindet, müssen andere Wege der Interaktion mit der Anwendung diese Logik duplizieren. Wenn die Transaktionen dagegen in der application liegen, dann ist sichergestellt, dass alle Wege von außen (ob über Web-Controller, REST-APIs, Messaging oder andere Schnittstellen) das gleiche Transaktionsverhalten verwenden. Außerdem ist es einfacher, Tests für die Domänenlogik zu schreiben. Sie benötigen insbesondere keine Web- oder sonstige Infrastruktur.
Ein anderes Argument ist die Laufzeit: Beispielsweise Controller sind Teil der Präsentationsschicht und darauf fokussiert, Benutzeranfragen so schnell wie möglich zu bearbeiten. Transaktionen können Ressourcen (wie Datenbankzeilen) sperren und so die Verarbeitung verlangsamen.

Wenn man sorgfältig darauf achtet, die Objekte des domain models höchstens lesend in der api verwendet werden, treten die oben geschilderten Probleme, weshalb man die Transaktion schon im Controller öffnen möchte, nicht auf.

Vorteile der Onion-Architektur:

- Onion verbessert die Testbarkeit des gesamten Codes, da Unit-Tests für einzelne Schichten erstellt werden können, ohne andere Module zu beeinträchtigen.
- Insbesondere ist die Domänenlogik besser testbar, da sie keine Infrastrutkur-Logik enthält.
- Frameworks/Technologien können leichter geändert werden, ohne den fachlichen Domänenkern zu beeinträchtigen.
- Die Zwiebel-Schichten sind nicht eng gekoppelt, im Sinne der Separation of Concerns.

Kosten:

- Die Separierung der Ringe führt häufig zu zusätzlichem Mapping, insbesondere der Daten.
- Es gibt zusätzliche Indirektionen durch das Dependency Inversion Principle, welche nicht immer intuitiv verständlich sind.
- Bei Anwendungen ohne oder mit sehr wenig Domänenlogik (CRUD-artigen Anwendungen) sollte man abwägen, ob sich dieser zusätzliche Aufwand wirklich lohnt.

3. Onion vs. Ports&Adapters/Hexagonal vs. Clean Architecture

Info
https://www.maibornwolff.de/know-how/ddd-architekturen-im-vergleich/ von Stephan Schneider

https://herbertograca.com/2017/11/16/explicit-architecture-01-ddd-hexagonal-onion-clean-cqrs-how-i-put-it-all-together/

Gulmariyam Yerzhanova: "Vergleich von verschiedenen Architekturen im Kontext des Domain-Driven Designs auf Basis von praktischen Implementierungen im Backend", Bachelorarbeit, Heinrich Heine Universität Düsseldorf, August 2023, PDF

Gulmariyam Yerzhanova, Marcus Franz: "Von Hexagonal- über Onion- bis Clean-Architektur – Unterschiede und Gemeinsamkeiten", betterCode() Clean Architecture, März 2024

Onion: Siehe oben und Zitat aus dem Artikel von Stephan Schneider:

> "*2008 präsentiert Jeffrey Palermo seine Onion-Architektur, ebenfalls als Verbesserung des Schichtenmodells [...]. Rückblickend lässt sich Cockburns Core Logic [...] in Palermos Architektur als Application Core wiederfinden [...] . Dieser untergliedert sich, ähnlich wie in o.g. hypothetischer Core Logic bei Cockburn [...] in zwei weitere Schichten (Application Services und Domain Services, zusammenfassend die Business Logic) mit dem Domain Model in der Mitte. Palermo unterteilt seine Business Logic jedoch nicht in primär und sekundär, sondern ordnet sie konzentrisch an.*"

Ports-and-Adapters-Architektur, Zitat aus dem Artikel von Stephan Schneider:

> "*Mit seiner Ports-and-Adapters-Architektur (2005) rückt Alistair Cockburn erstmals die Fachlichkeit einer Anwendung in den Mittelpunkt des Softwareentwurfs [...]. Er legt die Logik-Schicht des Schichtenmodells [...] in das Zentrum und nennt diese Core Logic [...]. Die äußeren Schichten (Presentation und Data) nennt er Adapters und entkoppelt sie von der Core Logic durch jeweils eine weitere Schicht, die Ports. Einen vom Frontend getriebenen Port und dessen Adapter nennt er primär und einen die Datenbank treibenden Port und dessen Adapter dementsprechend sekundär.*"

Clean Architecture, Zitat aus dem Artikel von Stephan Schneider:

> "*In seiner Clean Architecture (2012) nennt Robert C. Martin [...] Business Logic Business Rules, und unterteilt sie in application-specific und enterprise-wide [...], jedoch ohne expliziten Logik-Kern, der Palermos Domain Model entsprechen würde [...]. Dementsprechend gehören zu Martins Enterprise-Wide Business Rules sowohl Werte-Objekte (Business Objects) als auch Entities und Repository-Interfaces. Zu den Application-Specific Business Rules gehören Use Cases [...] und DTOs [...]. Letztere werden von Controllern und Presentern in der nächst-äußeren Schicht, den Interface Adapters verwendet bzw. implementiert. In der äußersten Schicht liegen die Frameworks & Drivers, die zum Beispiel die Repository-Interfaces der Enterprise-Wide Business Rules implementieren."*

4. Dependeny Inversion Principle

Info
https://en.wikipedia.org/wiki/Dependency_inversion_principle

https://martinfowler.com/bliki/InversionOfControl.html

Das **Dependeny Inversion Principle** (DIP) ist eines der **SOLID-Prinzipien** von Robert C. Martin. Es besagt, dass Module/Implementierungen auf hoher Ebene nicht von Modulen/Implementierungen auf niedrigerer Ebene abhängen sollten. Abstraktionen sollen nicht von Details abhängen, sondern umgekehrt.

In der Onion-Architektur wird das DIP verwendet, um Abhängigkeiten zu verwalten und diese von außen nach innen zu orientieren. (Siehe auch Hollywood Principle: "Don't Call Us, We'll Call You.")

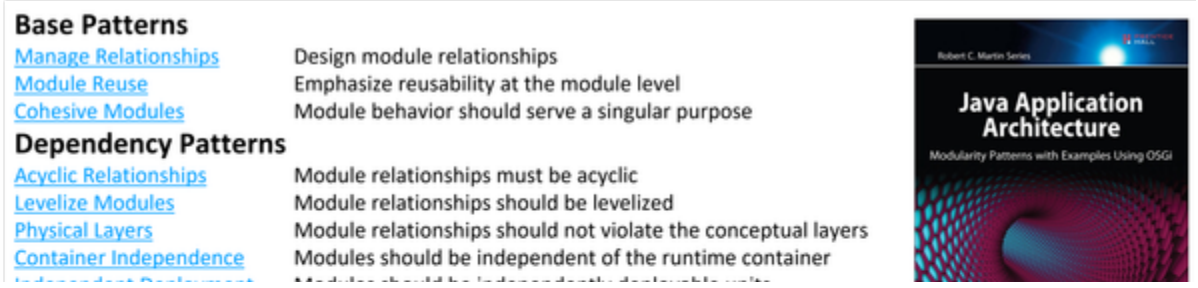
5. Modularity Patterns von Kirk Knoernschild (Englisch)

Info
Kirk Knoernschild: Java Application Architecture. Modularity Patterns with Examples Using OSGi, https://web.archive.org/web/20190504043055/https://www.kirkk.com/modularity/pattern-catalog/
siehe auch Vortrag M. Lehmann, K. Schaal, https://speakerdeck.com/mrtnlhmnn/jigsaw

John Ousterhout: "A Philosophy of Software Design",

siehe auch Youtube, siehe auch SE-Radio

und siehe auch Vortrag M. Lehmann, K. Schaal https://speakerdeck.com/mrtnlhmnn/jigsaw ("Deep Modules")



Independent Deployment	Modules should be independently deployable units
Usability Patterns	
Published Interface	Make a module's published interface well known
External Configuration	Modules should be externally configurable
Default Implementation	Provide modules with a default implementation
Module Facade	Facade as a coarse-grained entry point to ... fine-grained underlying implementation
Extensibility Patterns	
Abstract Modules	Depend upon the abstract elements of a module
Implementation Factory	Use factories to create a module's implementation classes
Separate Abstractions	Place abstractions and the classes that implement them in separate modules
Utility Patterns	
Collocate Exceptions	Exceptions should be close to the classes that throw them
Levelized Build	Execute the build in accordance with module levelization
Test Module	Each module should have a ... test module that validates it's behavior and illustrates it's usage

Abbildung: Kirk Knoernschilds Katalog der Modularity Patterns

5. Modularity Maturity Index (MMI)

Carola Lilienthal, Henning Schwendtner: "Domain-Driven Transformation", dpunkt-Verlag

Mit dem Modularity Maturity Index (MMI) wird ausgedrückt (und ein Stück messbar), wie gut der Sourcecode einer Anwendung auf eine fachliche Zerlegung vorbereitet ist, als Kombination dieser drei Kategorien:

- "Modularität" wird qualitativ beurteilt und drückt aus, ob hohe Kohäsion (im Inneren) und lose Koppelung (zu anderen Bausteinen) vorhanden ist.
- "Hierarchie" wird quantitativ beurteilt und drückt aus, wie stark das System einem Big-Ball-of-Mud ähnelt (mit Fragen wie: Zyklen vorhanden? Wieviele? Wieviele Klassen/Packages betroffen? Verletzungen in technischen Schichten? Wieviele? etc.)
- "Musterkonsistenz" wird qualitativ beurteilt und drückt aus, ob Architektur-/Entwurfsmuster eingehalten sind.

Diese drei Kategorien werden einzeln angeschaut und jeweils bzw. in Summe als quantitative Messgröße bewertet. Somit lässt sich insgesamt der MMI für ein System angeben.

(D) Modularisierung von Java und Spring-Boot-Anwendungen

1. Java: Packages und Interfaces

Die Modularität in Java-Anwendungen kann direkt die eingebauten Spracheigenschaften von Java nutzen:

- Strukturierung über Java-Packages
- Verwendung der Sichtbarkeiten public, protected, private und package-protected
- Java-Packages sind dabei keine Hierarchie (auch wenn die Namenskonvention der Packages etwas anderes vermuten lässt).

Man kann und sollte weitere Konventionen verwenden, z. B. alle extern sichtbaren Klassen (die ausgehende Schnittstelle) in ein "oberes /äußeres" Paket einsortieren und den Rest in Pakete weiter unten in der Hierarchie ("impl" o.Ä.). Ohne weitere statische Codeanalysewerkzeuge helfen die eingebauten Sichtbarkeitsprüfungen dann aber nicht (viel).

Java-Interfaces und abstrakte Klassen helfen, die Deklaration von Funktionalität von ihrer Implementierung zu trennen. Weitere Unterstützung kommt mit Java 17 z.B. in Form von Sealed Classes.

2. Sourcecode-Strukturen für Java (z.B. mit Git, Maven, Intellij)

Man kann Sourcecode-Strukturen in verschiedene Packages (siehe oben) aufteilen, verschiedene Intellij-Projekte/Module und/oder Maven-Module und/oder Git-Projekte.

Es ist Best-Practice, das gleiche Level an Aufteilung und Namensgebung/-konventionen in allen benutzten Tools gleich zu halten (so dass: Maven module == IntelliJ module == Git Projekt).

3. Java Deployment Artefakte: JAR, WAR/EAR, OSGi

Die kompilierten Artefakte des Java-Codes sind JAR-Dateien (Java-Archive). Für JEE-Anwendungsdienste kann ein WAR (Web Archive) oder ein EAR (Enterprise Archive) erstellt werden.

Für OSGi-Projekte werden OSGi-Bundles erstellt.

Alle diese Artefakte haben die gleiche Grundidee eines Meta-Deskriptors (wie MANIFEST.MF).

4. Jigsaw: JPMS für Java Module

Info
https://speakerdeck.com/mrtnlhmnn/jigsaw

https://speakerdeck.com/mrtnlhmnn/komponentendesign-und-modularity-patterns-mit-dem-java-modulsystem-jigsaw

https://speakerdeck.com/mrtnlhmnn/workshop-java9-das-neue-modulsystem-jigsaw

https://openjdk.org/projects/jigsaw/

Seit Java 9 gliedert Java seine interne Struktur in sogenannte Module im **Java Platform Module System (JPMS)**. Diese Module wurden als Teil von JSR376 als "Jigsaw" eingeführt. Java-Module haben ein restriktives Konzept der Sichtbarkeit und Zugänglichkeit mit - sowohl zur Build- als auch zur Laufzeit. Ihr Meta-Deskriptor heißt module-info.java/.class.

Das JDK selbst ist in solche JPMS-Module gegliedert (siehe JMOD-Dateien in $JDK/jmods). Zur Laufzeit ersetzt ein Modulpfad den Klassenpfad.

Um Abwärtskompatibilität zu unterstützen, wird ein (mehr oder weniger) transparentes Migrations- und Transformationskonzept zwischen dem "alten" Klassenpfad und dem neuen Modulpfad bereitgestellt (z.B. per Automatic Modules).

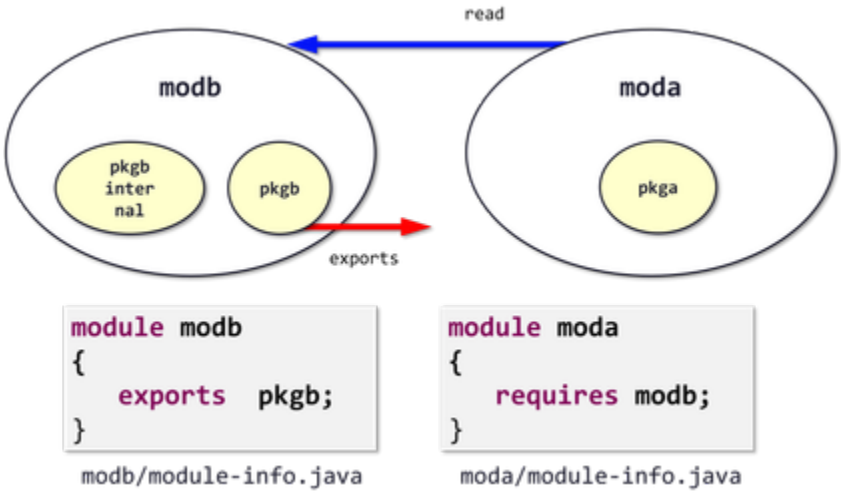


Abbildung: Java JPMS, Jigsaw Module
(Quelle: selbst erstellt)

5. Spring Modulith, jMolecules (Englisch)

Info
https://spring.io/projects/spring-modulith

https://github.com/xmolecules/jmolecules

"*Spring **Modulith** allows developers to build well-structured Spring Boot applications and guides developers in finding and working with application modules driven by the domain. It supports the verification of such modular arrangements, integration testing individual modules, observing the application's behavior on the module level and creating documentation snippets based on the arrangement created.*"

"[jMolecules is a] *... set of libraries to help developers implement domain models in distraction-free, plain old Java. Ideas behind jMolecules [are]: Explicitly express architectural concepts for easier code reading and writing. Keep domain-specific code free from technical dependencies. Reduce boilerplate code. Automatically generate documentation and validate implementation structures and your architecture*"

6. ArchUnit

Abbildung: ArchUnit-Logo

Info
https://speakerdeck.com/mrtnlhmnn/automatisierte-architekturtests-und-statische-code-analyse-mit-archunit

ArchUnit ist ein Werkzeug zur **statischen Codeanalyse,** das in Form von JUnit-basierten Tests vordefinierte Architekturregeln überprüft. Der Build schlägt fehl, wenn die architektonischen Regeln in der Architektur nicht befolgt werden. Vorteile:

- Diese architektonischen Regeln sind leicht zu pflegen und benötigen kein separates Toolset (wie Sonar).
- Sie können direkt in der IDE ausgeführt werden.
- Sie können in der gleichen (Java-basierten) Sprache geschrieben werden wie der Code selbst. (Es gibt eine Portierung für C# / DOT.NET)
- Das Refactoring des Codes führt (in den meisten Fällen) auch zu einem Refactoring der ArchUnit-Regeln.

Zudem kann ArchUnit auch als **statisches Code-Analyse-Tool** für eigene Zwecke eingesetzt werden (z.B. wenn Migrationen anstehen und alle relevanten Abhängigkeiten im Code gefunden werden müssen, um eine Kosten- und Aufwandsabschätzung zu erhalten).

7. Test-Tools and -Libraries

Info
https://dzone.com/articles/7-awesome-libraries-for-java-unit-amp-integration

Beliebte Integrations- und automatisierter **Testbibliotheken** für Java- und Spring Boot-Anwendungen:

- **JUnit** ist ein weit verbreitetes Unit-Testing-Framework für Java-Anwendungen. Es ermöglicht Entwicklern, wiederholbare Tests für einzelne Codeeinheiten zu schreiben und auszuführen.
- **Spring Test** ist Teil des Spring Ökosystems, das Unterstützung für das Testen von Spring-Anwendungen bietet. Es umfasst Funktionen wie das Laden der Testkonfiguration, Caching und Unterstützung für Integrationstests von Spring-Komponenten.
- **TestNG** ist ein Test-Framework für Java-Anwendungen, das JUnit ähnelt, aber zusätzliche Funktionen wie Unterstützung für datengesteuerte Tests und Testkonfiguration über XML-Dateien bietet.
- **Mockito** ist ein beliebtes Mocking-Framework für Java-Anwendungen. Es ermöglicht Entwicklern die Erstellung von Mock-Objekten für den Einsatz beim Testen und enthält Funktionen wie Argument-Matcher, Spies und Stubbing.
- **Hamcrest** ist eine Bibliothek zum Schreiben von Matcher-Objekten, die zur Durchführung flexibler und lesbarer Anpassungen in Test-Assertions verwendet werden können.
- **WireMock** ist eine Bibliothek zur Erstellung von HTTP-basierten Mock-Diensten. Sie ermöglicht es Entwicklern, das Verhalten eines entfernten HTTP-Dienstes zu simulieren, indem sie einen lokalen WireMock-Server erstellen, der so konfiguriert werden kann, dass er bestimmte Antworten auf HTTP-Anfragen zurückgibt.
- **REST-assured** ist eine Bibliothek zum Testen von RESTful-Webdiensten. Sie ermöglicht es Entwicklern, Tests in einem flüssigen, leicht zu lesenden Stil zu schreiben, und bietet Unterstützung für JSON- und XML-Parsing.
- **Cucumber** ist ein Tool für Behavior-Driven Development (BDD). Es ermöglicht Entwicklern, Tests in einem natürlichsprachlichen Format zu schreiben, das auch für nicht-technische Beteiligte leicht zu verstehen ist.
- **ArchUnit:** siehe oben.

Siehe auch Kapitel 4 und 5, Abschnitte zu Tests.

(E) Modularisierung auf Makro-Ebene mit Microservices und Self-Contained Systems

1. Monolith

Ein **Monolith** ist eine Softwarearchitektur, bei der eine einzige große Anwendung die gesamte Funktionalität des Systems übernimmt.

- Eine monolithische Anwendung wird in der Regel als eine einzige, zusammenhängende Einheit erstellt (sowohl was Quellcode- als auch Artefakt/Deployment-Ebene betrifft). Dies bedeutet, dass alle Komponenten des Systems, wie die Benutzeroberfläche, die Geschäftslogik und die Datenspeicherung, eng in dieselbe Codebasis integriert sind und oft mit derselben Programmiersprache und demselben Framework erstellt werden.
- Eine monolithische Architektur ist für kleine Anwendungen einfach zu entwickeln und zu verstehen, aber wenn eine Anwendung wächst und komplexer wird, wird es zunehmend schwieriger, sie zu warten, zu testen und (technisch wie organisatorisch) zu skalieren. Da es sich bei der Anwendung um eine einzige Einheit handelt, wird es immer schwieriger, neue Funktionen hinzuzufügen oder Änderungen an bestehenden Funktionen vorzunehmen, ohne das gesamte System zu beeinträchtigen.
- Der Begriff "Monolith" kann sich auf einen schlecht strukturierten "Big Ball of Mud" beziehen - wenn dies nicht der Fall ist, ist der Begriff "Modulith" besser geeignet, siehe unten.
- Der Begriff kann sich auch auf eine monolithisch bereitgestellte Anwendung beziehen, einen so genannten "Deployment Monolithen" (siehe unten).

Eine monolithische Architektur ist i.d.R. weniger flexibel und weniger skalierbar als eine Service-Architektur. Dennoch ist sie aufgrund der Einfachheit des Entwicklungsprozesses und der geringeren Kosten weit verbreitet, insbesondere bei kleinen bis mittelgroßen Anwendungen - und hat dort damit (je nach Qualitätsanforderungen) auch ihre Berechtigung.

Wenn ein Unternehmen seine monolithische Anwendung skalieren möchte, ist der Prozess der Aufteilung in (Micro-)Services komplex und kostspielig und erfordert i.d.R. einen erheblichen Aufwand für die Neugestaltung der Architektur.

Umsetzung in Flexinale
In der Case-Study "Flexinale" starten wir mit einem **Monolithen** und strukturieren und zerlegen ihn schrittweise - zunächst als **Modulith** mit Onion, dann mit getrennten Komponenten, abschließend als **Verteiltes, service-basiertes System** mit Event-driven Architecture.

2. Deployment Monolith

Ein **Deployment Monolith** ist eine Softwarearchitektur, bei der die gesamte Anwendung als eine einzige, in sich geschlossene Einheit bereitgestellt wird. In dieser Architektur sind alle Komponenten der Anwendung, wie die Benutzeroberfläche, die Geschäftslogik und die Datenzugriffsebenen, eng miteinander verbunden und werden in einem einzigen Prozess(-raum) ausgeführt.

Dazu werden alle Module gemeinsam installiert und in die einzige laufende Anwendung deployt. Diese Module nutzen denselben Technologie-Stack. Tritt in einem der Module ein Fehler auf (z. B. ein Speicherleck), ist der gesamte Prozess betroffen, da es im Falle von Fehlern in der Regel keine oder keine ausreichende Isolierung zur Laufzeit gibt. Siehe auch "Resilience" in Kapitel 4 (E).

3. Modulith

In einem **Modulith** ist die monolithische Software in der inneren Struktur gut organisiert und in kleine, unabhängige und austauschbare Module bzw. Komponenten unterteilt. Jede dieser Komponenten hat dabei eine einzige Verantwortung und ist lose mit anderen Komponenten gekoppelt (siehe dazu Kapitel 3).

Idealerweise sind die Komponenten auch intern wohlgeformt, z.B. per Onion-Architektur. Dieser Ansatz ermöglicht die flexible und effiziente Entwicklung, Prüfung und Deployment der Software.

Ein Modulith verbessert gezielt Flexibilität und Skalierbarkeit bei der Softwareentwicklung sowie schnellere Entwicklungs- und Bereitstellungszyklen, somit Wartbarkeit und Evolvierbarkeit von Software.

Durch die Aufteilung der Software in kleinere, besser handhabbare Teile ist ein Modulith leichter zu verstehen, zu testen und zu ändern als ein Big-Ball-of-Mud-Monolith. Zudem lässt sich die Software leichter skalieren, um steigenden Anforderungen gerecht zu werden, sowie für sich ändernde Anforderungen weiterentwickeln.

4. Microservices

Info
https://martinfowler.com/articles/microservices.html

https://www.innoq.com/de/articles/2019/12/monolith-zu-microservices-migration/

Artikelserie von Martin Lehmann und Renato Vinga-Martins in Informatik Aktuell, 2018:

- https://www.informatik-aktuell.de/entwicklung/methoden/microservices-fachliche-entscheidungshilfen-fuer-den-einsatz.html
- https://www.informatik-aktuell.de/entwicklung/methoden/microservices-nur-bei-ausreichender-komplexitaet.html
- https://www.informatik-aktuell.de/entwicklung/methoden/wie-gross-darf-ein-microservice-sein-und-ist-das-ueberhaupt-wichtig.html
- https://www.informatik-aktuell.de/entwicklung/methoden/der-fachliche-schnitt-von-microservices-was-ist-das-was.html
- https://www.informatik-aktuell.de/entwicklung/methoden/microservices-sind-verteilte-systeme-asynchronitaet-und-eventual-consistency.html
- https://www.informatik-aktuell.de/entwicklung/methoden/wann-und-fuer-wen-eignen-sich-microservices.html

Marcus Franz, Martin Lehmann, Dr. Renato Vinga-Martins: "DDD als Best-Practice für Datenmodellierung von Microservices", JUG Augsburg, Oktober 2022, https://speakerdeck.com/mrtnlhmnn/ddd-als-best-practice-fur-datenmodellierung-von-microservices

Ein **Microservice** ist ein Software-Architekturstil, bei dem eine große Anwendung als Sammlung kleiner, unabhängiger Services aufgebaut ist, die über genau definierte APIs miteinander kommunizieren. Jeder Microservice ist für eine bestimmte Geschäftsfähigkeit oder Funktion verantwortlich und wird unabhängig von den anderen Services entwickelt, bereitgestellt und skaliert.

Die Microservices-Architektur zielt darauf ab, die Skalierbarkeit, Wartbarkeit und Evolvierbarkeit von Software zu verbessern, indem eine monolithische Anwendung in kleinere, besser verwaltbare Service aufgeteilt wird - idealerweise anhand von fachlichen Grenzen. Dieser Ansatz ermöglicht schnellere Entwicklungs- und Bereitstellungszyklen sowie eine größere Flexibilität bei der Skalierung einzelner Services, um veränderten Anforderungen gerecht zu werden.

Im Gegensatz zu einem Modul in einem Modulith läuft jeder Microservice als eigener Prozess und kommuniziert mit anderen Microservices über leichtgewichtige Mechanismen, z.B. über synchrone HTTP/REST-APIs/Calls. So können verschiedene Microservices in unterschiedlichen Programmiersprachen geschrieben und unabhängig voneinander bereitgestellt und skaliert werden. Dieser Ansatz ermöglicht es auch, dass verschiedene Teams gleichzeitig an verschiedenen Microservices arbeiten, was die Flexibilität und Geschwindigkeit der Entwicklung erhöht.

Allerdings bringen Microservices auch Herausforderungen mit sich, wie die Kommunikation zwischen den Diensten, die Erkennung von Diensten, Sicherheit, Lastausgleich, Überwachung und Tests, siehe Kapitel zu Integration.

5. Grenzen und Sizing eines Microservice

Info
Eberhard Wolff: "Microservices. Grundlagen flexibler Softwarearchitekturen", 2. Auflage, dpunkt-Verlag

Die **Grenzen** eines Microservices sollte anhand verschiedener Kriterien festgelegt werden, darunter:

1. Fachlichkeit und Geschäftsfähigkeiten: Microservices sollten auf eine bestimmte fachliche Geschäftsfähigkeit oder -funktionalität ausgerichtet sein, wie z. B. Kundenmanagement, Bestandsmanagement oder Auftragsabwicklung.
2. Bounded Context: Ein Microservices sollte folglich um einen bestimmten Bounded Context herum entwickelt werden, der die Grenzen des Microservices in Bezug auf die Daten und Funktionen, für die er verantwortlich ist, definiert.
3. Transaktionsgrenzen: Geschäftslogik, die gemeinsam geändert und konsistent gehalten werden soll, also innerhalb einer Transaktion, sollte innerhalb eines Microservice liegen. Anderfalls wäre eine verteilte Transaktion notwendig.
4. Eigenständigkeit: Ein Microservice sollte autonom und in sich geschlossen sein, mit minimalen Abhängigkeiten von anderen Services. Dies ermöglicht eine größere Flexibilität in Bezug auf die Bereitstellung, Skalierung und den Betrieb des Dienstes.
5. Datenhoheit: Jeder Microservice sollte seine eigenen Daten besitzen und für das Management seiner eigenen Datenintegrität, -konsistenz und -verfügbarkeit verantwortlich sein. Daten, die von außen konsumiert werden, werden idealerweise in ein internes Datenmodell überführt.
6. Technologie-Stack: Jeder Microservice sollte so konzipiert sein, dass er einen bestimmten Technologiestack verwendet, der für seine Funktionalität und die Daten, über die er die Hoheit hat, geeignet ist.
7. Skalierung: Ein Microservice sollte so konzipiert sein, dass er unabhängig skaliert und die erwartete Last für seine Funktionalität bewältigen kann.
8. Sicherheit: Bei der Entwicklung von Microservices sollte querschnittlich auf Sicherheitsaspekte geachtet werden (Authentifizierung und Autorisierung), um geeignete Sicherheitsmaßnahmen zum Schutz der Daten und Funktionen des Dienstes zu treffen.
9. Schnittstellen: Ein Microservice sollte über eine einfache und klar definierte Schnittstelle verfügen, die stabil und leicht zu bedienen sein sollte.

Eberhard Wolff beschreibt in seinem Buch (siehe dort, S. 37) die **Einflussfaktoren,** über die ein typisches und ideales Sizing eines Microservice erreicht wird. Ein Microservice ist nicht zu groß und nicht zu klein zu wählen. Als obere Grenze nennt er das Sizing des Microservice über die Teamgröße: Ein Team soll einen Microservice unabhängig von andereren Teams weiterentwickeln und produktiv nehmen können. Hier wird als Daumenwert oft das "One-Pizza-Team" genannt: Ein Team sollte nicht größer sein als dass es von einer Party-Pizza satt wird, das entspricht einer Größe von 6-8 Personen. Untere Grenze ist die Infrastruktur und die Verteilung: Ist es aufwändiger, Infrastruktur bereitzustellen bzw. wird

zu viel Remote-Kommunikation zu ineffizient (langsam, schlechte Latenz), so ist der Microservice zu klein. Fragen nach Konsistenz/Datenhoheit /Transaktionalität geben fachlich motivierte Größen vor.

Ein wesentliches Kriterium für das Sizing von Microservices ist **fachlicher Natur**:

- Eine **obere Grenze** für einen Microservice ist gegeben durch einen Bounded Context. Ein Bounded Context ist eine fachliche Grenze - daher sollte ein Microservice nicht größer als ein Bounded Context sein.
- Eine **untere Grenze** für einen Microservice ist durch die die Größe eines Aggregates gegeben: Ein Aggregate solle immer vollständig innerhalb eines Microservice liegen, denn über einen Aggregate lassen sich Transaktionsgrenzen ableiten. Ein Aggregate fasst Entitäten und Wertobjekte zusammen, die in einer engen fachlichen Beziehung stehen. Diese teilen somit zusammenhängende Geschäftslogik und sollten nur gemeinsam geändert werden, damit die Konsistenzregeln eingehalten werden. Daraus folgt insbesondere, dass eine Veränderung eines Aggregates innerhalb einer Transaktion durchgeführt werden muss.

Alle diese Faktoren sind für ein gutes Sizing einzubeziehen. Fachlichkeit muss dabei der Haupttreiber sein. Anders als der Begriff "Micro" suggeriert, sind Microservices i.d.R. gar nicht so klein. Wie groß oder klein ein solcher Service ist, ist zweitrangig (und ein sinnvolle Metrik fehlt letztlich auch, da Lines-of-Code o.Ä. diese Diskussion ad absurdum führen). Klein oder groß sind keine Werte an sich. Stattdessen: Ein Microservices muss fachlich *passend* geschnitten sein und technisch-organisatorische Grenzen berücksichtigen.

6. Self-Contained Systems

Info
https://scs-architecture.org/

Ein **Self-Contained System (SCS)** ist eine Art von Softwarearchitektur, bei der jede Komponente des Systems völlig unabhängig und autark ist. Das bedeutet, dass jede Komponente über alles verfügt, was sie zum korrekten Funktionieren benötigt, einschließlich ihrer eigenen Datenspeicherung, Benutzeroberfläche und Geschäftslogik.

Die Webseite scs-architecture.org gibt als Kriterien an (Zitat, auf Englisch):

1. *Each SCS is an autonomous web application. For the SCS's domain, all data, the logic to process that data and all code to render the web interface is contained within the SCS. An SCS can fulfill its primary use cases on its own, without having to rely on other systems being available.*
2. *Each SCS is owned by one team. This does not necessarily mean that only one team can change the code, but the owning team has the final say on what goes into the code base, for example by merging pull-requests.*
3. *Communication with other SCSs or 3rd party systems is asynchronous wherever possible. Specifically, other SCSs or external systems should not be accessed synchronously within the SCS's own request/response cycle. This decouples the systems, reduces the effects of failure, and thus supports autonomy. The goal is decoupling concerning time: An SCS should work even if other SCSs are temporarily offline. This can be achieved even if the communication on the technical level is synchronous, e.g. by replicating data or buffering requests.*
4. *An SCS can have an optional service API. Because the SCS has its own web UI, it can interact with the user — without going through a UI service. However, an API for mobile clients or for other SCSs might still be useful.*
5. *Each SCS must include data and logic. To really implement any meaningful features both are needed. An SCS should implement features by itself and must therefore include both.*
6. *An SCS should make its features usable to end-users via its own UI. Therefore the SCS should have no shared UI with other SCSs. SCSs might still have links to each other. However, asynchronous integration means that the SCS should still work even if the UI of another SCS is not available.*
7. *To avoid tight coupling an SCS should share no business code with other SCSs. It might be fine to create a pull-request for an SCS or use common libraries, e.g. database drivers or oAuth clients.*
8. *To make SCSs more robust and improve decoupling shared infrastructure can be minimized. E.g. a shared database make fail safeness and scalability of the SCSs depend on the central database. However, due to e.g. costs a shared database with separate schemas or data models per SCS can be a valid alternative.*

7. Begriffstrennung: Microservice vs. Self-Contained System?

Info
https://scs-architecture.org/vs-ms.html

Eberhard Wolff: "Microservices. Grundlagen flexibler Softwarearchitekturen", 2. Auflage, dpunkt-Verlag

Artikelserie von Martin Lehmann und Renato Vinga-Martins in Informatik Aktuell, 2018, v.a. Teile 2 und 3:

- https://www.informatik-aktuell.de/entwicklung/methoden/wie-gross-darf-ein-microservice-sein-und-ist-das-ueberhaupt-wichtig.html
- https://www.informatik-aktuell.de/entwicklung/methoden/der-fachliche-schnitt-von-microservices-was-ist-das-was.html

SCS ähnelt der Microservices-Architektur, jedoch mit leicht anderem Scope. Zitat von der Webseite, auf Englisch:

> *The SCS approach shares a lot of concepts with microservices, including ideas about enforcing isolation via independently deployable units, the alignment of organizational and architectural boundaries, support for diversity in terms of technology choices, and the lack of centralized infrastructure. If you're willing to see this as the core of microservices, you could view SCSs as a specialization. But compared to some other aspects seen by many people as key attributes of microservices, there are some important differences:*

- ***A microservice is probably smaller than an SCS**. An SCS might be large enough to keep a team busy and provide more meaningful business value.*
- *There are usually fewer SCSs than microservices. A logical system such as an e-commerce shop might have 5 to 25 SCSs i.e. for billing, order processing etc. An e-commerce shop might have 100s of microservices.*
- *SCSs should ideally not communicate with each other while this is fine for microservices.*
- *SCSs have a UI, while microservices might separate the UI from the logic in its own service. However, some definitions of microservices include the UI in the microservice, too.*
- *SCSs should favor integration at the UI layer. Microservices typically focus on integration at the logic layer. (Again, some definitions of microservices also allow for integration at the UI layer.)*
- *Of course it is possible to split an SCS even further so it consists of microservices — in particular for the business logic. In this case, this can be seen as a particular micro-architecture approach.*

  SCS clearly focus on larger projects and a split into multiple teams. Microservices can be used for other purposes: Often small teams or even single developers use them e.g. to use Continuous Delivery more easy, to build more robust systems or to scale each microservice independently. So microservices are more versatile while SCS solve problems specifically with the architecture and organisation of large projects.

Die Grenzen zwischen den Definitionen von Microservices und Self-Contained Systems (sowie ähnlichen Begriffen wie Nanoservices) sind fließend. Eberhard Wolff definiert in seinem Buch (siehe dort, S. 55), Zitat:

> "*SCS [...] ist etwas, was ein Team bearbeitet und eine fachliche Einheit darstellt. [...] Ein alternativer Name ist eine Vertikale [...] teilt die Architektur in Fachlichkeit auf. [...]* "

Und weiter:

> "*Ein Microservice ist ein Teil eines SCS. Es ist eine technische Einheit und kann unabhängig deployt werden.*"

Diese Definition betont den technischen Charakter eines Microservice, den fachlichen eines SCS. Damit verbunden sind die feingranulare(re) Struktur eines Microservice im Vergleich zu der grobgranulare(re)n Struktur eines SCS. Andere Definitionen betonen bereits bei einem Microservices dessen fachlichen Schnitt und das damit fachliche eigenständige Einheiten abzubilden sind. Die Übergänge dieser verschiedenen Definitionen sind letztlich fließend. Entscheidend ist immer: Fachlichkeit und fachliche Strukturen sollen der Haupttreiber für den Schnitt sein. Mittel aus DDD wie Bounded Context und Aggregates helfen, solche fachlichen Grenzen zu identifizieren. Wo genau ein "Service" aufhört und ein "System" beginnt, diese Frage wird damit ein Stück weit akademisch. Relevant und entscheidend sind in erster Linie fachliche, dazu in zweiter Linie auch technische wie organisatorische Kriterien für den Schnitt.

8. Begriffstrennung: SOA vs. Microservices?

Eine **Serviceorientierte Architektur (SOA)** ist ein unternehmensweiter Ansatz für die Softwareentwicklung von Anwendungskomponenten, der sich wiederverwendbare Softwarekomponenten oder Dienste zunutze macht. In der SOA-Softwarearchitektur besteht jeder Dienst aus dem Code und den Datenintegrationen, die für die Ausführung einer bestimmten Geschäftsfunktion erforderlich sind - z. B. die Überprüfung der Kreditwürdigkeit eines Kunden, die Anmeldung auf einer Website oder die Bearbeitung eines Hypothekenantrags.

Wie bei SOA bestehen auch **Microservices-Architekturen** aus lose gekoppelten, wiederverwendbaren und spezialisierten Komponenten, die oft unabhängig voneinander arbeiten. Microservices verwenden ein hohes Maß an fachlicher Kohäsion, siehe DDD / Bounded Context. Anstatt unternehmensweit eingeführt zu werden, kommunizieren Microservices in der Regel über Anwendungsprogrammierschnittstellen (APIs), um einzelne Anwendungen anzubieten, die eine bestimmte Geschäftsfunktion (oder Funktionen für bestimmte Geschäftsbereiche) so ausführen, die sie flexibler, skalierbarer und widerstandsfähiger macht.

Der Hauptunterschied zwischen den beiden Ansätzen liegt im **Scope:** Vereinfacht gesagt, hat die **serviceorientierte Architektur (SOA) einen Enterprise Scope**, während die **Microservices-Architektur einen Application Scope** hat. In einem SOA-Modell werden Dienste oder Module gemeinsam genutzt und unternehmensweit wiederverwendet, während eine Microservice-Architektur auf einzelnen Diensten aufbaut, die unabhängig voneinander funktionieren.

Einige Konsequenzen sind u.a.:

- Bei SOA ist die **Wiederverwendbarkeit** von Integrationen das primäre Ziel, und auf Unternehmensebene ist das Streben nach einem gewissen Grad an Wiederverwendung unerlässlich. Wiederverwendbarkeit und gemeinsame Nutzung von Komponenten in einer SOA-Architektur erhöhen die Skalierbarkeit und Effizienz.
- **Kommunikation:** In einer Microservices-Architektur wird jeder Dienst unabhängig entwickelt und verfügt über sein eigenes Kommunikationsprotokoll. Bei SOA nutzt i.d.R. jeder Dienst einen (genauer: den einen, gemeinsamen) Kommunikationsmechanismus verpflichtend, einen Enterprise Service Bus (ESB). SOA verwaltet und koordiniert die Dienste, die sie über den ESB bereitstellt. Der ESB wird damit meist zu einem Single Point of Failure für das gesamte Unternehmen.
- **Governance:** Die Art der SOA, die gemeinsame Ressourcen einschließt, ermöglicht die Umsetzung gemeinsamer Datenverwaltungsstandards für alle Dienste. Die unabhängige Natur von Microservices ermöglicht keine einheitliche Data Governance, ja vermeidet diese bewusst. Dies bietet eine größere Flexibilität für jeden Service.

9. Mono-/Modulith oder Microservice?

Vorteile von **Mono-/Modulithen:**

- Einfach: Ein Monolith ist (bei guter interner Struktur!) relativ einfach zu verstehen und zu warten, da sich die gesamte Funktionalität i.d.R. in einer einzigen Codebasis befindet und gemeinsam deployt wird.
- Kosten: Die Entwicklung und Deployment eines Monolithen wird i.d.R. somit kostengünstiger sein.
- Einfachere Tests und Betrieb: Ein Monolith ist i.d.R. einfacher zu testen und zu betreiben, da die gesamte Funktionalität zentral vorliegt (keine Ende2Ende-Tests über Prozessgrenzen hinweg).
- Die Code-Basis des Monolithen liegt in einem Mono-Repo. Das erlaubt es, einfach größere interne strukturelle Änderungen / Refactorings durchzuführen, ohne auf abhängige externe Systeme Rücksicht nehmen zu müssen (Versionierung, technische + org. Abhängigkeiten).

Nachteile:

- Skalierbarkeit: Monolithen reichen i.d.R. dann nicht aus, wenn das System wächst bzw. seine Mengengerüste steigen, und es ist dann zu schwierig oder nicht möglich, bestimmte Teile des Systems zu skalieren.
- Entwicklung: Monolithen lassen sich im Laufe der Zeit nur schwer weiterentwickeln und ändern, da die Änderung eines Teils des Systems - bei schlechter interner Struktur - Auswirkungen auf das gesamte System haben kann.
- Deployment: Das Deployment eines Monolithen kann schwierig sein, da das gesamte System auf einmal bereitgestellt werden muss.

Vorteile von **Microservices** / **SCS:**

- Skalierbarkeit: Microservices / SCS können unabhängig voneinander skaliert werden, was die Bewältigung steigender Lasten und Nutzerzahlen erleichtert.
- Entwicklung: Microservices / SCS können unabhängig voneinander weiterentwickelt und geändert werden, was das Hinzufügen neuer oder das Ändern bestehender Funktionen erleichtert.
- Deployment: Microservices / SCS können unabhängig voneinander bereitgestellt werden, was die Bereitstellung und Aktualisierung des Systems erleichtert.
- Flexibilität: Jeder Microservice / SCS kann mit verschiedenen Technologien entwickelt und bereitgestellt werden, die den jeweiligen Anforderungen am besten entsprechen.
- Ausfallsicherheit: Microservices / SCS können so konzipiert werden, dass sie ein hohes Maß an Fehlertoleranz aufweisen, was zu geringeren Ausfallzeiten führen kann.

Nachteile:

- Komplexität: Microservices / SCS können aufgrund der größeren Anzahl von einzelnen Komponenten und deren Abhängigkeiten komplexer in der Entwicklung, Deployment und generellem Handling sein als eine monolithische Architektur.
- Inter-Service-Kommunikation: Die Kommunikation und der Datentransfer zwischen mehreren Services ist komplex, siehe Kapitel 4 zu Integration.
- Testen: Das Testen von Microservices/ SCS kann komplexer sein, da jeder Dienst einzeln getestet werden muss - und auch in kostspieligen Integrationstests.
- Monitoring und Logging: Die Monitoring und Logging von Microservices/ SCS ist komplexer als bei einer monolithischen Anwendung.
- Security: Die Security mehrerer Dienste ist komplexer als die Absicherung einer einzelnen monolithischen Anwendung. Umgekehrt kann man kritische Services aber auch gezielt(er) absichern.

Nachstehende Abbildung verdeutlicht noch einmal, wo **Mono-/Modulithen** (blaue Fläche) und **Microservices** (grüne Fläche) jeweils Stärken und Schwächen haben.

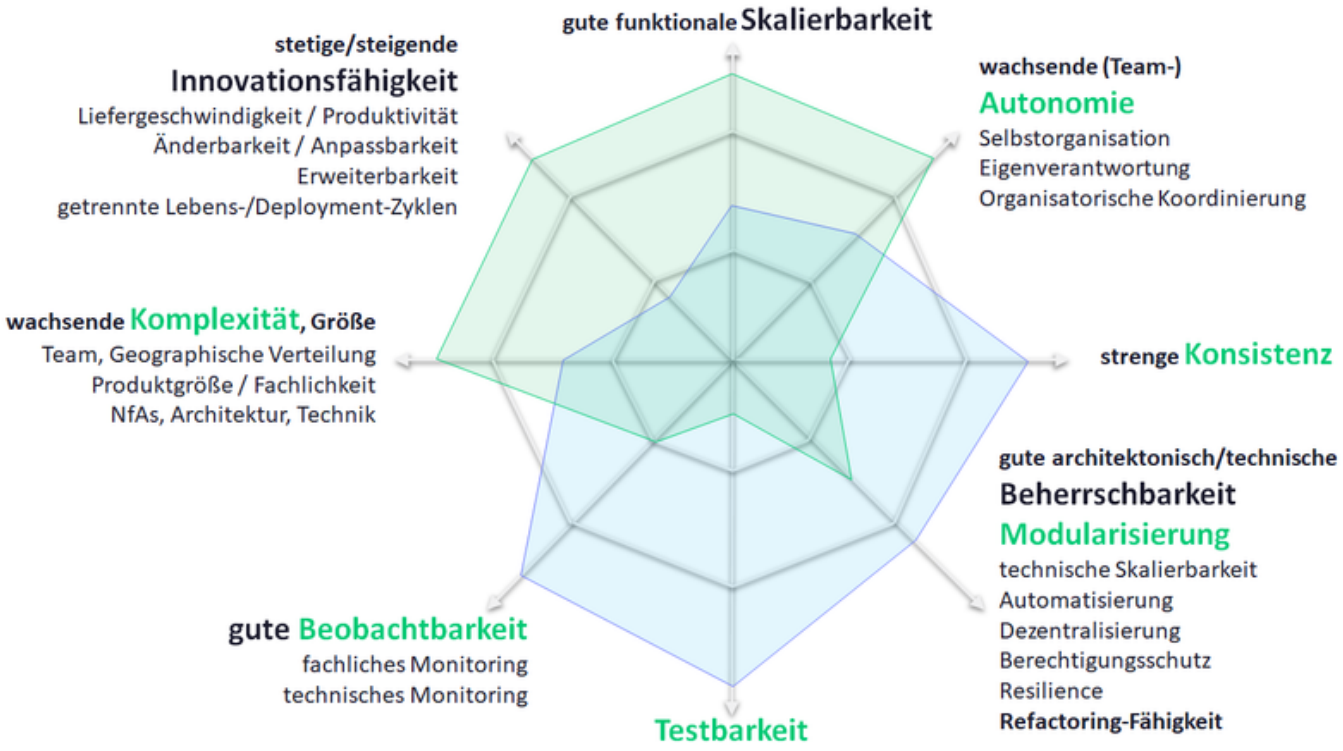


Abbildung: Stärken und Schwächen von Mono-/Modulithen und Microservices
(*Quelle: Marcus Franz, Martin Lehmann, Dr. Renato Vinga-Martins: "DDD als Best Practices für Datenmodellierung und Microservices", https://speakerdeck.com/mrtnlhmnn/ddd-als-best-practice-fur-datenmodellierung-von-microservices*)

10. Verteilte Systeme, Distributed Systems

Info
https://en.wikipedia.org/wiki/Fallacies_of_distributed_computing

Verteilte Systeme unterteilen die Laufzeit-Anwendung in separate Dienste und unterstützen somit die Isolation zur Laufzeit. Dies wird in der Regel auch getan, um die Skalierbarkeit zu unterstützen.

Ein solches verteiltes System hat jedoch eine Reihe von Implikationen, mit denen man umgehen muss. Die Kommunikation zwischen den Diensten erfolgt remote und das hat entscheidende Konsequenzen:

- Das bedeutet, dass man sich mit Versionierung der Schnittstelle auseinandersetzen muss.
- ... dito auch mit Sicherheit
- ... und auch mit unterschiedlichen (viel, viel) langsamerer Request-Performance, stark erhöhten Latenzen
- ... sowie unterschiedlichsten Arten von Fehlerszenarien (vgl. Delivery-Garantien).
- Außerdem wird der Code nicht mehr in einem Prozessraum ausgeführt, mit Auswirkungen auf Transaktionskontrolle (sofern man keine 2PC implementieren möchte, wovon abzuraten ist). Damit ergibt sich die Notwendigkeit, sich mit transaktionalen Commits und Rollbacks auf der Domänenebene zu befassen ("Fachliche Kompensationen").

Der Betrieb eines solchen Systems ist dabei wesentlich schwieriger und erfordert erweiterte Techniken für Protokollierung, Monitoring usw. Vgl. auch die "Fallacies of Distributed Computing", die schon vor 30 Jahren Trugschlüsse im Betrieb und Design verteilter Systeme benannt haben (Beispiele: "Das Netzwerk ist ausfallsicher" oder "Der Datendurchsatz ist unbegrenzt.")

Viel mehr dazu in Abschnitt 4 (E).

11. Kategorisierung von Systemen: Transaktional, Batch-orientiert, Daten-orientiert, Streaming-orientiert

Ein **transaktionales System** ist darauf ausgelegt, kleine Datenmengen performant zu verarbeiten und dabei ein hohes Maß an Konsistenz, Verfügbarkeit und Dauerhaftigkeit zu gewährleisten. Transaktionssysteme werden in der Regel für Aufgaben wie Online-Transaktionsverarbeitung (OLTP), elektronischen Handel und andere Systeme verwendet, die sofortige Aktualisierungen und ein hohes Maß an Datenkonsistenz erfordern. Für ein transaktionales Systems sind v.a. geringe Latenz, die Datenkonsistenz und hohe Verfügbarkeit entscheidend. Merkmale:

1. Performante Verarbeitung: Transaktionale Systeme verarbeiten Daten performant, idealerweise in Echtzeit und bieten sofortige Aktualisierungen und Antworten auf Benutzeranfragen.
2. Konsistenzorientiert: Transaktionale Systeme erfordern ein hohes Maß an Datenkonsistenz, d. h. sie müssen so konzipiert sein, dass die Datenintegrität und -konsistenz gewahrt bleibt. Dies kann den Einsatz von Transaktionen, Sperren und anderen Techniken zur Gewährleistung der Datenkonsistenz beinhalten. Siehe dazu auch Kapitel 8.
3. Geringe Latenz: Transaktionale Systeme müssen schnell auf Benutzeranfragen reagieren, daher müssen sie für geringe Latenzen ausgelegt sein.
4. Hohe Verfügbarkeit: Transaktionale Systeme müssen hochverfügbar sein, d. h. sie müssen so konzipiert sein, dass sie Ausfälle bewältigen und sich nach Ausfällen schnell wiederherstellen lassen. Siehe auch Abschnitt zu Restart in Kapitel 6.
5. Benutzerorientiert: Transaktionale Systeme sind in der Regel für die Endbenutzer zugänglich, und ihre Verfügbarkeit und Leistung sind für die Benutzererfahrung entscheidend.
6. Nebenläufigkeit der Zugriffe: Transaktionale Systeme sind für den gleichzeitigen Zugriff ausgelegt, d. h. sie müssen so konzipiert sein, dass mehrere Benutzer gleichzeitig auf das System zugreifen können.
7. Kurzlebige Transaktionen: Transaktionale Systeme verarbeiten kurzlebige Transaktionen, d. h., sie sind für kleine und schnelle Transaktionen ausgelegt.

Batch-orientierte Systeme sind nicht darauf ausgelegt, große Datenmengen in Echtzeit zu verarbeiten. Diese Systeme sind stattdessen auf Durchsatz statt auf niedrige Latenzzeiten optimiert. Sie werden in der Regel für Backend-Aufgaben und Aufgaben mit großen Datenmengen eingesetzt. Merkmale:

1. Nicht-Echtzeit-Verarbeitung: Batch-Systeme verarbeiten Daten nicht in Echtzeit, sondern in großen Batches. Dadurch können sie große Datenmengen effizienter und mit weniger Overhead als Echtzeitsysteme verarbeiten.
2. Leistungsorientiert: Batch-Systeme sind auf Leistung und Effizienz hinsichtlich Durchsatz optimiert und nicht auf geringe Latenzzeiten. Sie sind darauf ausgelegt, große Datenmengen schnell zu verarbeiten und große Datenmengen zu bewältigen.
3. Datenintensiv: Batch-Systeme werden in der Regel für die Verarbeitung großer Datenmengen eingesetzt, z. B. für Data Warehousing, Data Mining, Datenintegration und Berichterstattung. Sie sind für die Verarbeitung großer Datenmengen und für die horizontale Skalierung ausgelegt. Batch-Systeme sollten in der Lage sein, große Datenmengen zu verarbeiten, und sie sollten für eine horizontale Skalierung ausgelegt sein.
4. Backend-orientiert: Batch-Systeme konzentrieren sich in der Regel auf die Backend-Verarbeitung und sind in der Regel nicht für Endbenutzer zugänglich, haben also i.d.R. keine UI. Sie werden häufig so geplant, dass sie per Scheduler zu bestimmten Zeiten oder als Reaktion auf bestimmte eigehende Ereignisse ausgeführt werden.
5. Toleranz gegenüber Dateninkonsistenzen: Batch-Systeme können einige Dateninkonsistenzen tolerieren, da das Hauptziel in der Verarbeitung großer Datenmengen besteht und nicht in der Gewährleistung der Konsistenz jedes einzelnen Datensatzes.
6. Hoher Durchsatz: Batch-Systemesind darauf ausgelegt, große Datenmengen schnell zu verarbeiten und große Datenmengen zu bewältigen. Sie können Daten parallel verarbeiten und nutzen verteilte Systeme, um die Leistung zu verbessern.

7. Geplant: Batch-Systeme werden in der Regel so geplant, dass sie zu bestimmten Zeiten oder als Reaktion auf bestimmte Ereignisse, z. B. das Eintreffen neuer Daten, ausgeführt werden.

Ein **daten- (oder datenfluss-) orientiertes System** ist darauf ausgelegt, große Datenmengen, in der Regel strukturierte Daten, zu verwalten und zu verarbeiten und sie für verschiedene Arten von Abfragen, Analysen und Berichten verfügbar zu machen. Diese Systeme sind für die Speicherung, den Abruf und die Analyse großer Datenmengen sowie für einen effizienten und flexiblen Zugriff auf die Daten optimiert. Sie werden in der Regel für Aufgaben wie Data Warehousing, Data Mining, Datenintegration und Berichterstellung eingesetzt. Merkmale:

1. Datenintensiv: Daten- und datenflussorientierte Systeme sind darauf ausgelegt, große Datenmengen zu verarbeiten und eine breite Palette von Abfragen und Analysen zu unterstützen.
2. Datenmodellierung: Daten- und datenflussorientierte Systeme erfordern ein gut definiertes Datenmodell, das für die spezifischen Abfrage- und Analysetypen, die durchgeführt werden sollen, optimiert ist. Dies kann Denormalisierung, Partitionierung und andere Techniken zur Verbesserung der Abfrageleistung beinhalten.
3. Datenzugriff: Daten- und datenflussorientierte Systeme müssen einen effizienten und flexiblen Zugang zu den Daten bieten, d. h. sie müssen so konzipiert sein, dass sie eine Vielzahl von Abfragen und Analysen unterstützen. Dies kann die Verwendung von Indizes, materialisierten Ansichten und anderen Techniken zur Verbesserung der Abfrageleistung beinhalten.
4. Skalierbarkeit: Daten- und datenflussorientierte Systeme müssen in der Lage sein, große Datenmengen zu verarbeiten, und sollten für eine horizontale Skalierung ausgelegt sein.
5. Datenflussorientiert: Daten- und datenflussorientierte Systeme sind für den effizienten Fluss großer Datenmengen zwischen den verschiedenen Phasen einer Pipeline konzipiert, z. B. Dateneingabe, -umwandlung, -speicherung und -analyse.
6. Datenverwaltung: Daten- und datenflussorientierte Systeme sollten über klar definierte Data-Governance-Richtlinien und -Prozesse verfügen, um Datenqualität, Sicherheit und Compliance zu gewährleisten.
7. Datenspeicherung: Daten- und datenflussorientierte Systeme benötigen eine große Speicherkapazität, um große Datenmengen zu speichern, und sollten in der Lage sein, Daten in verschiedenen Formaten und Strukturen zu speichern.

Ein **streaming-orientiertes System** ist ein Systemtyp, der darauf ausgelegt ist, Daten in Echtzeit zu verarbeiten und zu analysieren, während sie erzeugt werden. Zu den Merkmalen eines strömungsorientierten Systems gehören:

1. Verarbeitung in Echtzeit: Streaming-orientierte Systeme sind darauf ausgelegt, Daten in Echtzeit zu verarbeiten und zu analysieren, während sie generiert werden; dies ermöglicht eine schnelle Reaktion auf Ereignisse und ein unmittelbares Feedback.
2. Hoher Durchsatz: Streaming-orientierte Systeme sind darauf ausgelegt, ein hohes Datenvolumen und einen hohen Durchsatz zu verarbeiten, was die Verarbeitung einer großen Datenmenge in kurzer Zeit ermöglicht. Das Verständnis des Datenflusses durch das System ist der Schlüssel zum Aufbau eines Systems, das einen hohen Durchsatz und geringe Latenzzeiten bewältigen kann.
3. Skalierbarkeit: Streaming-orientierte Systeme sind skalierbar, d. h. das System kann im Laufe der Zeit eine wachsende Datenmenge verarbeiten.
4. Fehlertoleranz: Streaming-orientierte Systeme sind so konzipiert, dass sie fehlertolerant sind, d. h., das System kann auch bei einem Ausfall weiterarbeiten - z.B. sollte es über mehrere Datenkopien verfügen, und es sollte ausfallsicher ausgelegt sein, d. h. es sollte über ein Backup-System verfügen, das im Falle eines Ausfalls einspringen kann.
5. Geringe Latenzzeit: Streaming-orientierte Systeme sind so konzipiert, dass sie eine geringe Latenzzeit haben, was eine schnelle Reaktion auf Ereignisse und eine sofortige Rückmeldung ermöglicht.
6. Ereignisgesteuert: Streaming-orientierte Systeme sind so konzipiert, dass sie ereignisgesteuert sind, d. h. das System kann auf Ereignisse reagieren, sobald sie eintreten, und Daten in Echtzeit verarbeiten.
7. Verteilt: Streaming-orientierte Systeme sind so konzipiert, dass sie verteilt sind, so dass das System eine große Datenmenge verarbeiten und horizontal skalieren kann.
8. Datenverarbeitungspipelines: Streaming-orientierte Systeme verwenden häufig Datenverarbeitungspipelines. Dies ermöglicht die Erstellung einer Reihe von Stufen, die die Daten durchlaufen, bevor sie verarbeitet oder analysiert werden. Dies ermöglicht Flexibilität bei der Verarbeitung und Analyse von Daten.

12. Microservices Patterns

Info
https://microservices.io/patterns/index.html

Die Webseite microservices.io liefert eine größere Pattern-Sammlung für Microservices und kategorisiert sie in

- Application Patterns: Decomposition, Data, Database, Testing, UI
- Application Infrastructure Patterns: Cross-cutting concerns, Security, Communication Style, Reliability, API, Observability
- Infrastructure Patterns: Deployment, Discovery

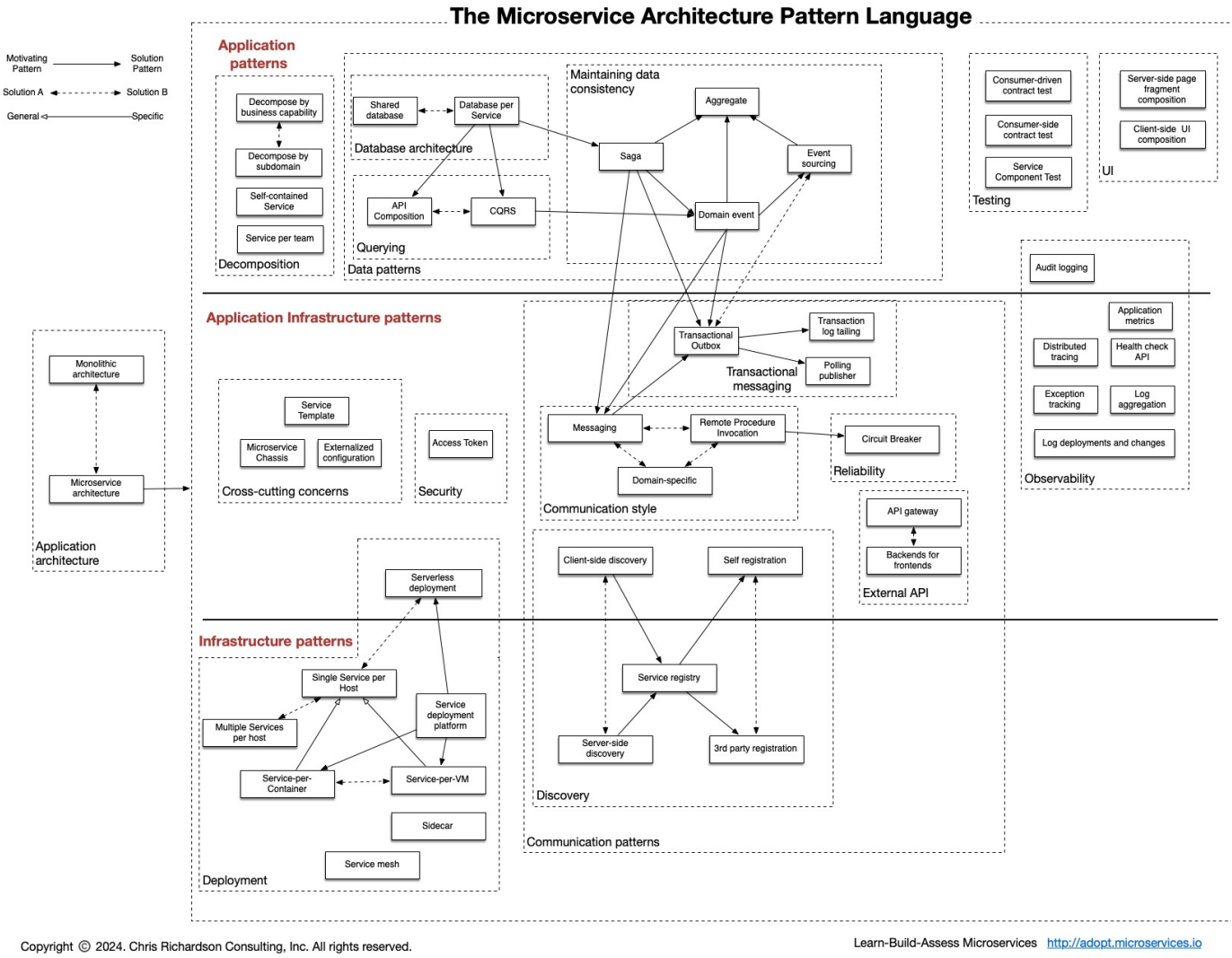


Abbildung: Microservices Architecture Pattern Language
(Quelle: https://microservices.io/i/MicroservicePatternLanguage.jpg)