Architektur von Informationssystemen

Hochschule für angewandte Wissenschaften

Sommersemester 2018

Nils Löwe / nils@loewe.io / @NilsLoewe

Was ist Softwarearchitektur?

Geschichte und Trends

Sichten auf Architekturen

Qualiät und andere nichtfunktionale Anforderungen

Architekturmuster

Dokumentation von Architekturen

Technologien und Frameworks

Wiederholung
UML für Softwarearchitekten

UML für Softwarearchitekten

Welches UML Diagramm für welche Sicht?

Baustein-Sicht

- Gute Namen wählen!
- Rollennamen anzugeben, Navigationsrichtung vorschreiben und Multiplizitäten festlegen
- Verwenden Sie nur eine Art von Schnittstellendarstellung
- Nutzen Sie Stereotypes für verschiedene Arten von fachlichen und technischen Klassen und Komponenten

UML für Softwarearchitekten

Baustein-Sicht

- Paketdiagramm
- Komponentendiagramm
- Klassendiagramm
- Aktivitätsdiagramm
- Zustandsdiagramm

UML für Softwarearchitekten

Verteilungs-Sicht

- Hauptelemente: Knoten und Kanäle zwischen den Knoten
- Knoten sind Standorte, z.B. Cluster, Rechner, Chips, ...
- Kanäle sind die physikalischen Übertragungswege, z.B. Kabeln, Bluetooth, Wireless, ...

UML für Softwarearchitekten

Verteilungs-Sicht

- Verteilungsdiagramm
- Kontextdiagramm

UML für Softwarearchitekten

Laufzeitsicht

- Elemente der Laufzeitsicht sind immer um Instanzen von Bausteinen, die in der Bausteinsicht enthalten sind, also um Objekte zu den Klassen oder um instanziierte Komponenten.

UML für Softwarearchitekten

Laufzeitsicht

- Objektdiagramm
- Kompositionsstrukturdiagramm
- Sequenzdiagramm
- Laufzeitkontextdiagramm
- Kommunikationsdiagramm
- Interaktionsdiagramm

UML für Softwarearchitekten

Warum UML?

- UML hat die Kästchen und Striche für uns standardisiert
- Die Bausteine der Architektur lassen sich auf verschiedenen Abstraktionsebenen miteinander in Beziehung setzen
- Die Zusammenarbeit wird effektiver, wenn alle hinter den Kästchen und Strichen das Gleiche verstehen

UML für Softwarearchitekten

Praxisrelevanz?

Wiederholung
Qualiät und andere nichtfunktionale Anforderungen

Was ist Qualität?

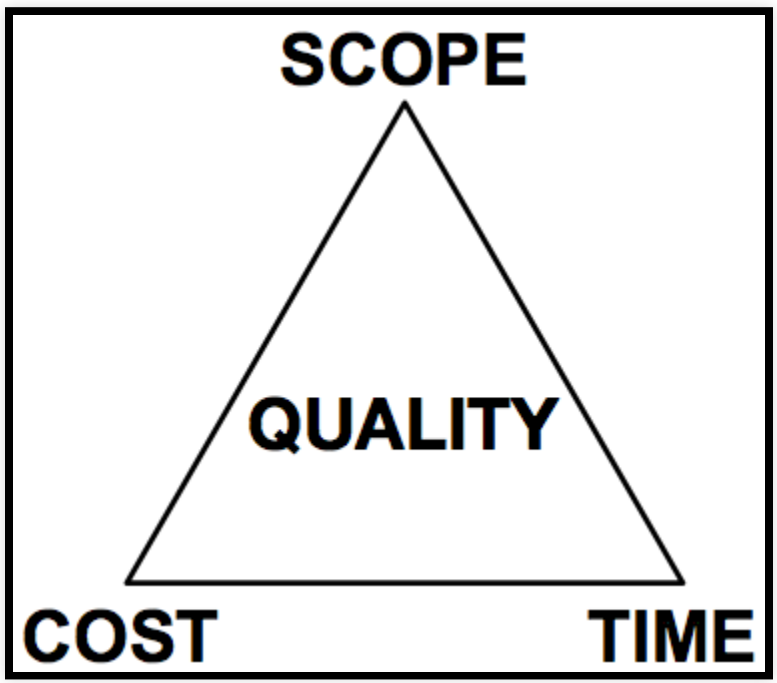
Was ist Qualität?

Duden: Qualität=„Beschaffenheit, Güte, Wert“

Was ist Qualität?

Die Qualität stimmt, wenn der Kunde wiederkommt und nicht das Produkt

Was ist Qualität?



Quelle: http://pm-blog.com/

Qualität ist ein wichtiges Ziel für Software-Architekten

Probleme von Qualität

- Qualität ist nur indirekt messbar
- Qualität ist relativ (jeweils anders für: Anwender, Projektleiter, Betreiber, ...)
- Die Qualität der Architektur korreliert nicht notwendigerweise mit der Codequalität
- Erfüllung aller funktionalen Anforderungen lässt keinerlei Aussage über die Erreichung der Qualitätsanforderungen zu

Qualitätsmerkmale nach DIN/ISO 9126

Funktionalität

Zuverlässigkeit

Benutzbarkeit

Effizienz

Änderbarkeit

Übertragbarkeit

Die weiteren Details zu Qualität und nichtfunktionalen Anforderungen sind Bestandteil der kommenden Kapitel

Wiederholung
Architekturmuster

Was sind Architekturmuster?

A pattern for software architexture describes a particular recurring design problem that arises in specific design contexts, and presents a well-proven generic scheme for its solution. The solution scheme is specified by describing its constituent components, theis relationships, and the ways in which they collaborate.

(1996 / Pattern Oriented Software Architectute)

Ein Architekturmuster beschreibt eine bewährte Lösung für ein wiederholt auftretendes Entwurfsproblem

(Effektive Softwarearchitekturen)

Ein Architekturmuster definiert den Kontext für die Anwendbarkeit der Lösung

(Effektive Softwarearchitekturen)

Warum Architekturmuster?

Erfolg kommt von Weisheit.

Weisheit kommt von Erfahrung.

Erfahrung kommt von Fehlern.

Aus Fehlern kann man hervorragend lernen.

Leider akzeptiert kaum ein Kunde Fehler, nur weil Sie Ihre Erfahrung als Software-Architekt sammeln.

In dieser Situation helfen Heuristiken.

Heuristiken kodifizieren Erfahrungen anderer Architekten und Projekte, auch aus anderen Bereichen der Systemarchitektur.

Heuristiken sind nicht-analytische Abstraktionen von Erfahrung

Es sind Regeln zur Behandlung komplexer Probleme, für die es meist beliebig viele Lösungsalternativen gibt. Heuristiken können helfen, Komplexität zu reduzieren.

Andere Begriffe für Heuristiken sind auch „Regeln", „Muster" oder „Prinzipien".

Es geht immer um Verallgemeinerungen und Abstraktionen von konkreten Situationen.

Heuristiken bieten Orientierung im Sinne von Wegweisern, Straßenmarkierungen und Warnschildern.

Sie geben allerdings lediglich Hinweise und garantieren nichts. Es bleibt in Ihrer Verantwortung, die passen- den Heuristiken für eine bestimmte Situation auszuwählen:

Die Kunst der Architektur liegt nicht in der Weisheit der Heuristiken, sondern in der Weisheit, a priori die passenden Heuristiken für das aktuelle Projekt auszuwählen.

Anwendung auf Software-Architekturen:

klassische Architekturmuster

Horizontale Zerlegung: „In Scheiben schneiden"

Vertikale Zerlegung: „In Stücke schneiden"

weitere Architekturmuster

Alles ist möglich...

Horizontale Zerlegung

Jede Schicht stellt einige klar definierte Schnittstellen zur Verfügung und nutzt Dienste von darunter liegenden Schichten.

Vertikale Zerlegung

Jeder Teil übernimmt eine bestimmte fachliche oder technische Funktion.

Prinzipien zur Zerlegung

Kapselung (information hiding)

- Kapseln von Komplexität in Komponenten.
- Betrachtung der Komponenten als „black box“,
- Definition klarer Schnittstellen
- Ohne Kapselung erschwert eine Zerlegung das Problem, statt es zu vereinfachen (was bekannt ist, wird auch ausgenutzt!)

Prinzipien zur Zerlegung

Wiederverwendung

- wiederverwendbarkeit verringert den Wartungsaufwand
- Achtung: Nur Dinge wiederverwenden, bei denen es sinnvoll ist

Prinzipien zur Zerlegung

Iterativer Entwurf

- Überprüfung eines Entwurfs mit Prototypen oder Durchstichen
- Evaluation der Stärken und Schwächen eines Entwurfes
- Explizite Bewertung und Analyse dieser Versuche

Prinzipien zur Zerlegung

Unabhängigkeit der Elemente

- Geringe Abhängigkeiten erhöhen die Wartbarkeit und Flexibilität des Systems
- Komponenten sollen keine Annahmen über die Struktur anderer Komponenten machen

Fragen?

Praktikumsaufgabe

2. Praktikumsaufgabe

Praktikumsaufgabe

REST

Representational State Transfer (REST) bezeichnet ein Programmierparadigma für verteilte Systeme, insbesondere für Webservices. REST ist eine Abstraktion der Struktur und des Verhaltens des World Wide Web.

REST hat das Ziel, einen Architekturstil zu schaffen, der die Anforderungen des modernen Web besser darstellt.

Praktikumsaufgabe

REST

- Der Zweck von REST liegt schwerpunktmäßig auf der Maschine-zu-Maschine-Kommunikation.
- REST kodiert keine Methodeninformation in den URI, der URI gibt Ort und Namen der Ressource an
- Eine Ressource kann über verschiedene Medientypen dargestellt werden, auch Repräsentation der Ressource genannt.

Praktikumsaufgabe

REST Prinzipien

- Client-Server
- Zustandslosigkeit
- Caching
- Einheitliche Schnittstelle
- Mehrschichtige Systeme
- Code on Demand

Praktikumsaufgabe

REST Einheitliche Schnittstelle

- Adressierbarkeit von Ressourcen
- Repräsentationen zur Veränderung von Ressourcen
- Selbstbeschreibende Nachrichten (Verwendung von Standardmethoden wie Http)
- „Hypermedia as the Engine of Application State“ (HATEOAS)

Praktikumsaufgabe

HTTP

Das Hypertext Transfer Protocol (HTTP) ist ein zustandsloses Protokoll zur Übertragung von Daten auf der Anwendungsschicht über ein Rechnernetz. Es wird hauptsächlich eingesetzt, um Webseiten (Hypertext-Dokumente) aus dem World Wide Web (WWW) in einen Webbrowser zu laden.

Es ist jedoch nicht prinzipiell darauf beschränkt und auch als allgemeines Dateiübertragungsprotokoll sehr verbreitet.

Praktikumsaufgabe

HTTP Verben

- GET fordert die angegebene Ressource vom Server an.
- POST fügt eine neue (Sub-)Ressource unterhalb der angegebenen Ressource ein.
- PUT legt die angegebene Ressource an. Wenn die Ressource bereits existiert, wird sie geändert.
- PATCH ändert einen Teil der angegebenen Ressource.
- DELETE löscht die angegebene Ressource.

Praktikumsaufgabe

Ein kleines Beispiel

swagger.io

Was ist Softwarearchitektur?

Geschichte und Trends

Sichten auf Architekturen

Qualiät und andere nichtfunktionale Anforderungen

Architekturmuster

Dokumentation von Architekturen

Technologien und Frameworks

Überblick über Architekturmuster

Arten von Architekturmustern?

Chaos zu Struktur / Mud-to-structure

Verteilte Systeme

Interaktive Systeme

Adaptive Systeme

Domain-spezifische Architektur

Chaos zu Struktur / Mud-to-structure

- Organisation der Komponenten und Objekte eines Softwaresystems
- Die Funktionalität des Gesamtsystems wird in kooperierende Subsysteme aufgeteilt
- Zu Beginn des Softwareentwurfs werden Anforderungen analysiert und spezifiziert
- Integrierbarkeit, Wartbarkeit, Änderbarkeit, Portierbarkeit und Skalierbarkeit sollen berücksichtigt werden

Chaos zu Struktur / Mud-to-structure

Layers

Pipes und Filter

Blackboard

Domain-driven Design

Verteilte Systeme

- Verteilung von Ressourcen und Dienste in Netzwerken
- Kein "zentrales System" mehr
- Basiert auf guter Infrastruktur lokaler Datennetze

Verteilte Systeme

Serviceorientierte Architektur (SOA)

Peer-to-Peer

Client-Server

Interaktive Systeme

- Strukturierung von Mensch-Computer-Interaktionen
- Möglichst gute Schnittstellen für die Benutzer schaffen
- Der eigentliche Systemkern bleibt von der Benutzerschnittstelle unangetastet.

Interaktive Systeme

Model View Controller (MVC)

Model View Presenter

Presentation-Abstraction-Control (PAC)

Adaptive Systeme

- Unterstützung der Erweiterungs- und Anpassungsfähigkeit von Softwaresystemen.
- Das System sollte von vornherein mögliche Erweiterungen unterstützen
- Die Kernfunktionalität sollte davon unberührt bleiben kann.

Adaptive Systeme

Mikrokernel

Reflexion

Dependency Injection

Anti-Patterns

Anti-Patterns?

Ein Anti-Pattern ist in der Softwareentwicklung ein häufig anzutreffender schlechter Lösungsansatz für ein bestimmtes Problem. Es bildet damit das Gegenstück zu den Mustern (Entwurfsmuster, Analysemuster, Architekturmuster, …), welche allgemein übliche und bewährte Problemlösungsansätze beschreiben.

Überblick über Antipatterns

Projektmanagement Anti-Patterns

Architektur Anti-Pattern

Code Smells

Organisations Anti-Pattern

Projektmanagement Anti-Patterns

Blendwerk

Aufgeblähte Software

Feature creep

Scope creep

Brooks'sches Gesetz

Death Sprint

Death March

Architektur Anti-Patterns

Big Ball of Mud

Gasfabrik

Gottobjekt

Innere-Plattform-Effekt

Spaghetticode

Sumo-Hochzeit

Code Smells

Zwiebel

Copy and Paste

Lavafluss

Magische Werte

Reservierte Wörter

Unbeabsichtigte Komplexität

Organisations Anti-Pattern

Wunderwaffe

Das Rad neu erfinden

Das quadratische Rad neu erfinden

Body ballooning

Empire building

Warme Leiche

Single head of knowledge

Organisations Anti-Pattern II

Management nach Zahlen

Vendor Lock-In

Design by Committee

Boat Anchor

Dead End

Swiss Army Knife

Überblick über Antipatterns

Projektmanagement Anti-Patterns

Architektur Anti-Pattern

Code Smells

Organisations Anti-Pattern

Blendwerk

Das Blendwerk (englisch *Smoke and mirrors*) bezeichnet nicht fertige Funktionen, welche als fertig vorgetäuscht werden.

Aufgeblähte Software

Als Bloatware (englisch *„aufblähen“*) wird Software bezeichnet, die mit Funktionen überladen ist bzw. die Anwendungen sehr unterschiedlicher Arbeitsfelder ohne gemeinsamen Nutzen bündelt. Für den Anwender macht dies das Programm unübersichtlich, für Entwickler unwartbar.

Feature creep

bezeichnet es, wenn der Umfang der zu entwickelnden Funktionalität in einem Projektplan festgehalten wird, diese aber dauernd erweitert wird.

Der Kunde versucht, nach der Erstellung des Projektplanes weitere Funktionalität mit unterzubringen. Dies führt zu Problemen, wenn die in Arbeit befindliche Version nicht das notwendige Design aufweist, Termine nicht eingehalten werden können oder die realen Kosten über die planmäßigen Kosten wachsen.

Scope creep

ist ähnlich wie der Feature creep, jedoch nicht auf Funktionalität bezogen, sondern auf den Anwendungsbereich. Auch hier zeichnet sich der Auftraggeber dadurch aus, dass er geschickt und versteckt den Umfang der Software nachträglich erweitern möchte, ohne dass er dies explizit zugibt.

Death Sprint

Bei einem Death Sprint (Überhitzter Projektplan) wird Software iterativ bereitgestellt. Die Bereitstellung erfolgt hierbei in einer viel zu kurzen Zeitspanne. Nach außen sieht das Projekt zunächst sehr erfolgreich aus, da immer wieder neue Versionen mit neuen Eigenschaften abgeschlossen werden. Allerdings leidet die Qualität des Produktes sowohl nach außen sichtbar, wie auch technisch, was allerdings nur der Entwickler erkennt. Die Qualität nimmt ab mit jeder „erfolgreichen“ neuen Iteration.

Death March

Ein Death March (Todesmarsch; gelegentlich auch Himmelfahrtskommando) ist das Gegenteil von einem Überhitzten Projektplan. Ein Todesmarschprojekt zieht sich ewig hin.

Ein Todesmarschprojekt kann auch bewusst in Kauf genommen werden, um von Defiziten in der Organisation abzulenken und Entwicklungen zu verschleppen, d. h. so lange an etwas zu entwickeln, bis eine nicht genau spezifizierte Eigenschaft in irgendeiner Form subjektiv funktioniert.

Brooks'sches Gesetz

"Adding manpower to a late software project makes it later."

Überblick über Antipatterns

Projektmanagement Anti-Patterns

Architektur Anti-Pattern

Code Smells

Organisations Anti-Pattern

Big Ball of Mud

bezeichnet ein Programm, das keine erkennbare Softwarearchitektur besitzt. Die häufigsten Ursachen dafür sind ungenügende Erfahrung, fehlendes Bewusstsein für Softwarearchitektur, Fluktuation der Mitarbeiter sowie Druck auf die Umsetzungsmannschaft. Obwohl derartige Systeme aus Wartbarkeitsgründen unerwünscht sind, sind sie dennoch häufig anzutreffen.

Gasfabrik

Als Gasfabrik (englisch *Gas factory*) werden unnötig komplexe Systementwürfe für relativ simple Probleme bezeichnet.

The Blob / Gottobjekt

Ein Objekt (“Blob“) enthält den Großteil der Verantwortlichkeiten, während die meisten anderen Objekte nur elementare Daten speichern oder elementare Dienste anbieten.

Innere-Plattform-Effekt

tritt auf wenn ein System derartig weitreichende Konfigurationsmöglichkeiten besitzt, dass es letztlich zu einer schwachen Kopie der Plattform wird, mittels derer es gebaut wurde.

Ein Beispiel sind Datenmodelle, die auf konkrete (anwendungsbezogene) Datenbanktabellen verzichten und stattdessen mittels allgemeiner Tabellen eine eigene Verwaltungsschicht für die Datenstruktur implementieren mit dem eigentlichen Ziel, die Flexibilität zu erhöhen.

Spaghetti Code

Der Code ist weitgehend unstrukturiert; keine Objektorientierung oder Modularisierung: undurchsichtiger Kontrollfluss.

Sumo-Hochzeit

Als Sumo-Hochzeit bezeichnet man es, wenn ein Fat Client unnatürlich stark abhängig von der Datenbank ist.

In der Datenbank ist hierbei sehr viel Logik in Form der datenbankeigenen Programmiersprache positioniert. Beispielsweise in Oracle mit der Programmiersprache PL/SQL. Die ganze Architektur ist dadurch sehr unflexibel.

Überblick über Antipatterns

Projektmanagement Anti-Patterns

Architektur Anti-Pattern

Code Smells

Organisations Anti-Pattern

Zwiebel

Als Zwiebel (engl. Onion) bezeichnet man Programmcode, bei dem neue Funktionalität um (oder über) die alte gelegt wird.

Häufig entstehen Zwiebeln, wenn ein Entwickler ein Programm erweitern soll, das er nicht geschrieben hat. Der Entwickler möchte oder kann die bereits existente Lösung nicht komplett verstehen und setzt seine neue Lösung einfach drüber. Dies führt mit einer Vielzahl von Versionen und unterschiedlichen Entwicklern über die Jahre zu einem Zwiebel-System.

Cut-and-Paste Programming

Code wird an zahlreichen Stellen wiederverwendet, indem er kopiert und verändert wird. Dies sorgt für Wartungsprobleme

Lösung: Black-Box-Wiederverwendung, Refaktorisierung

Lava Flow

Ein Lavafluss (englisch Lava flow oder Dead Code) beschreibt den Umstand, dass in einer Anwendung immer mehr „toter Quelltext“ herumliegt. Dieser wird nicht mehr genutzt. Statt ihn zu löschen, werden im Programm immer mehr Verzweigungen eingebaut, die um den besagten Quelltext herumlaufen oder auf ihm aufbauen. Redundanter Code ist der Überbegriff zu totem Code.

The Golden Hammer / Wunderwaffe

Ein bekanntes Verfahren (Golden ”Hammer“) wird auf alle moglichen Probleme angewandt Wer als einziges Werkzeug nur einen Hammer kennt, lebt in einer Welt voller Nägel.

Lösung: Ausbildung verbessern

Reinvent the Wheel

Da es an Wissen uber vorhandene Produkte und Lösungen (auch innerhalb der Firma) fehlt,wird das Rad stets neu erfunden. Erhöhte Entwicklungskosten und Terminprobleme.

Lösung: Wissensmanagement verbessern

Das quadratische Rad neu erfinden

Mit das quadratische Rad neu erfinden (englisch Reinventing the square wheel) bezeichnet man die Bereitstellung einer schlechten Lösung, wenn eine gute Lösung bereits existiert.

Body ballooning

Beim Body ballooning handelt der Vorgesetzte ausschließlich aus der Bestrebung heraus, seine Machtposition auszubauen, welche sich entweder aus der Unternehmensstruktur oder auch rein subjektiv aus der Anzahl der Mitarbeiter unter sich definiert. Dies kann dazu führen, dass der Vorgesetzte bewusst arbeitsintensivere Lösungen und Arbeitstechniken den effizienten vorzieht.

Empire building

Durch sachlich nicht nachvollziehbare, nicht konstruktive Maßnahmen versucht ein einzelner, seine Macht auszubauen bzw. zu erhalten. Dies kann Body ballooning sein, aber auch das ständige Beschuldigen anderer, gerade derer, die nicht mehr für die Unternehmung arbeiten, die Ausführung von pathologischer Politik, Diskreditierung, Mobbing und sonstige Facetten, die nur darauf abzielen, die eigene Position zu stärken bzw. den eigenen Status zu halten.

Warme Leiche

Eine warme Leiche (englisch warm body) bezeichnet eine Person, die einen zweifelhaften oder keinen Beitrag zu einem Projekt leistet.

Single head of knowledge

Ein Single head of knowledge ist ein Individuum, welches zu einer Software, einem Werkzeug oder einem anderen eingesetzten Medium, als einziges unternehmensweit das Wissen besitzt. Dies zeugt häufig von fehlendem Wissensmanagement, mangelndem Austausch zwischen den Kollegen oder Defiziten in der Organisation, kann aber auch von dem Individuum bewusst angestrebt worden sein.

Management nach Zahlen

(englisch *Management by numbers*) ist eine Anspielung auf Malen nach Zahlen. Beim Management nach Zahlen wird ein übermäßiger Schwerpunkt auf das quantitative Management gelegt. Insbesondere wenn Fokus auf Kosten gelegt wird, während andere Faktoren wie Qualität vernachlässigt werden.

Vendor Lock-In

Ein System ist weitgehend abhangig von einer proprietaren Architektur oder proprietaren Datenformaten

Lösung: Portabilität erhöhen

Lösung: Abstraktionen einführen

Design by Committee

Das typische Anti-Muster von Standardisierungsgremien, die dazu neigen, es jedem Teilnehmer recht zu machen und übermäßig komplexe Entwürfe abzuliefern

Lösung: Gruppendynamik und Treffen verbessern

Boat Anchor

Eine Komponente ohne erkennbaren Nutzen

Dead End

eingekaufte Komponente, die nicht mehr unterstützt wird

Swiss Army Knife

Eine Komponente, die vorgibt, alles tun zu können

Hilfreiches Wissen um Anti-Patterns vorzubeugen

”The Pragmatic Programmer”

(Andy Hunt, Dave Thomas)

”Clean Code”

(Uncle Bob Martin)

”The Developers Code”

(Ka Wai Cheung)

Chaos zu Struktur / Mud-to-structure

Layers

Pipes und Filter

Blackboard

Domain-driven Design

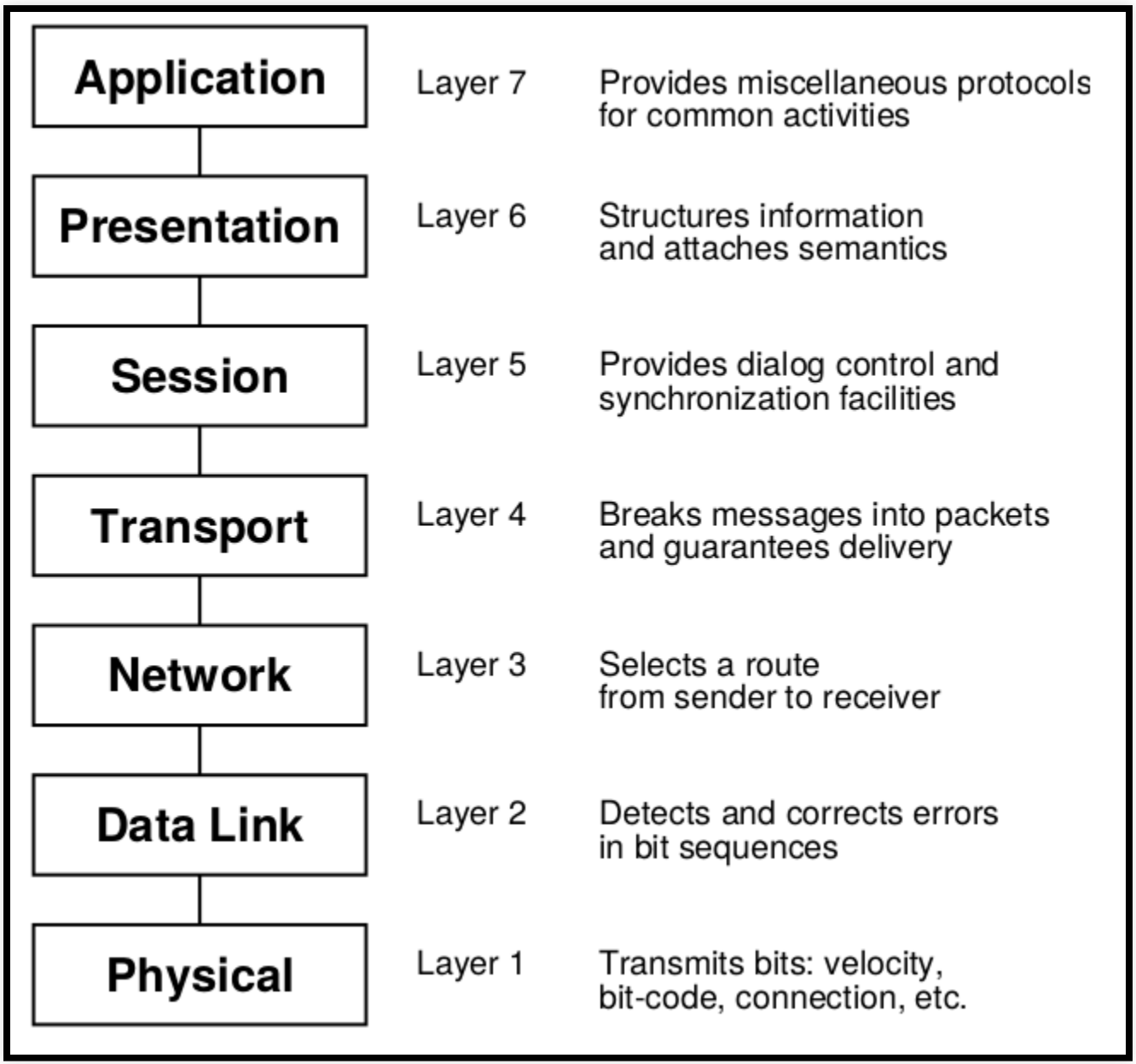
Layers

Das Layers-Muster trennt eine Architektur in verschiedene Schichten, von denen jede eine Unteraufgabe auf einer bestimmten Abstraktionsebene realisiert.

Beispiel: ISO/OSI-Referenzmodell

Netzwerk-Protokolle sind wahrscheinlich die bekanntesten Beispiele für geschichtete Architekturen. Das ISO/OSI-Referenzmodell teilt Netzwerk-Protokolle in 7 Schichten auf, von denen jede Schicht für eine bestimmte Aufgabe zuständig ist:

Beispiel: ISO/OSI-Referenzmodell



Layers

Aufgabe: Folgendes System bauen:

- Aktivitäten auf niederer Ebene wie Hardware-Ansteuerung, Sensoren, Bitverarbeitung
- Aktivitäten auf hoher Ebene wie Planung, Strategien und Anwenderfunktionalität
- Die Aktivitäten auf hoher Ebene werden durch Aktivitäten der niederen Ebenen realisiert

Layers

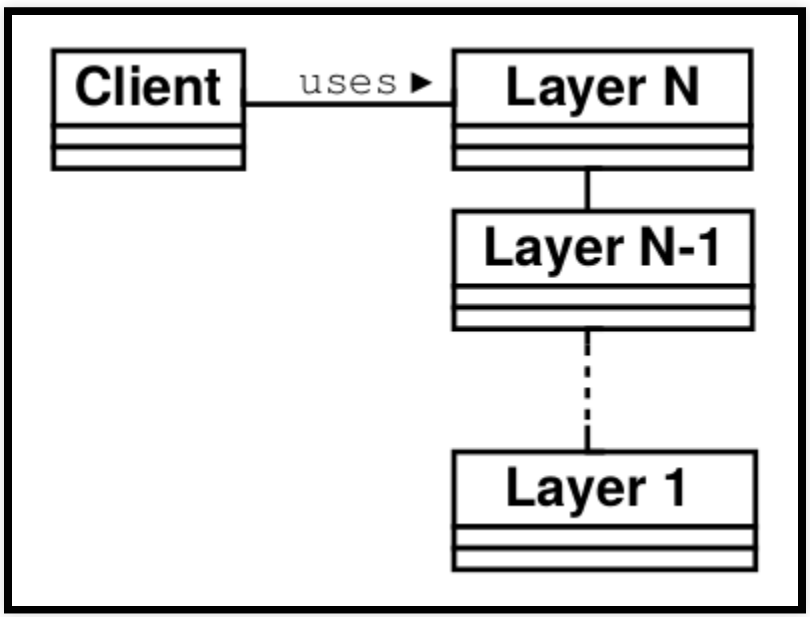
Dabei sollen folgende Ziele berücksichtigt werden:

- Änderungen am Quellcode sollten möglichst wenige Ebenen betreffen
- Schnittstellen sollten stabil (und möglicherweise standardisiert) sein
- Teile (= Ebenen) sollten austauschbar sein
- Jede Ebene soll separat realisierbar sein

Layers

Das Layers-Muster gliedert ein System in zahlreiche Schichten. Jede Schicht schützt die unteren Schichten vor direktem Zugriff durch höhere Schichten.

Layers



Layers

Dynamisches Verhalten

Top-Down Anforderung

Eine Anforderung des Benutzers wird von der obersten Schicht entgegengenommen. Diese resultiert in Anforderungen der unteren Schichten bis hinunter auf die unterste Ebene. Ggf. werden die Ergebnisse der unteren Schichten wieder nach oben weitergeleitet, bis das letzte Ergebnis an den Benutzer zurückgegeben wird.

Layers

Dynamisches Verhalten

Bottom-Up Anforderung

Hier empfängt die unterste Schicht ein Signal, das an die oberen Schichten weitergeleitet wird. Schließlich benachrichtigt die oberste Schicht den Benutzer.

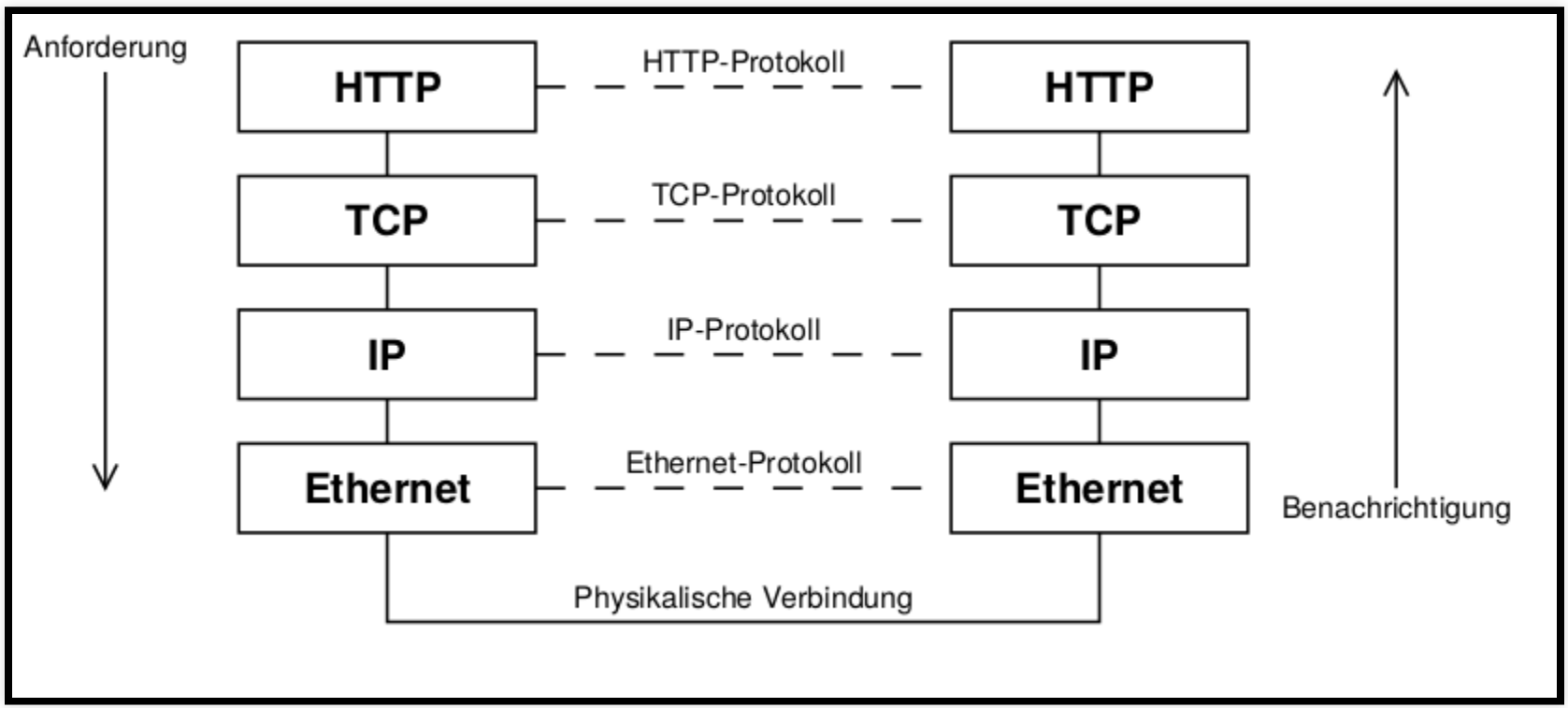
Layers

Dynamisches Verhalten: Protokoll Stack

In diesem Szenario kommunizieren zwei n-Schichten-Stacks miteinander. Eine Anforderung wandert durch den ersten Stack hinunter, wird übertragen und schließlich als Signal vom zweiten Stack empfangen. Jede Schicht verwaltet dabei ihr eigenes Protokoll.

Layers

Beispiel: TCP/IP



Layers

Vorteile

- Wiederverwendung und Austauschbarkeit von Schichten
- Unterstützung von Standards
- Einkapselung von Abhängigkeiten

Layers

Nachteile

- Geringere Effizienz
- Mehrfache Arbeit (z.B. Fehlerkorrektur)
- Schwierigkeit, die richtige Anzahl Schichten zu bestimmen

Layers

Bekannte Einsatzgebiete:

- Application Programmer Interfaces (APIs)
- Datenbanken
- Betriebssysteme
- Kommunikation. . .

Was ist Softwarearchitektur?

Geschichte und Trends

Sichten auf Architekturen

Qualiät und andere nichtfunktionale Anforderungen

Architekturmuster

Dokumentation von Architekturen

Technologien und Frameworks

Fragen?

Unterlagen: ai2018.nils-loewe.de