

# Architektur von Informationssystemen

Hochschule für angewandte Wissenschaften

Sommersemester 2018

Nils Löwe / [nils@loewe.io](mailto:nils@loewe.io) / @NilsLoewe

Was ist Softwarearchitektur?

# Geschichte und Trends

Sichten auf Architekturen

Qualität und andere nichtfunktionale Anforderungen

Architekturmuster

Dokumentation von Architekturen

Technologien und Frameworks

# Wiederholung Geschichte und Trends

Seit wann gibt es den Begriff der Softwarearchitektur?

## Konferenz über Softwaretechnik in Rom

Software Engineering Techniques. Report of a Conference Sponsored by the NATO Science Committee. Scientific Affairs Division, NATO, 1970, S. 12.

Warum?

Die Systeme wurden in den 1960ern so komplex, dass sie von mehreren Teams entwickelt werden mussten.

Beispiel: IBM OS/360

## Planung

- Entwicklungskosten: 40 Mio. USD
- Lines of Code: 1 Mio.
- Fertigstellung: 1965

Beispiel: IBM OS/360

## Realität

- Entwicklungskosten: 500 Mio. USD (Faktor 12,5)
- Lines of Code: 10 Mio. (Faktor 10)
- Fertigstellung: 1967 (2 Jahre zu spät)

## Beispiel: IBM SYSTEM/360

- Mainframes verwalten heute 80 % aller Unternehmensdaten
- Mainframes verarbeiten heute täglich 30 Mrd. Unternehmenstransaktionen (z.B. Banking, Flugbuchungen, ...)
- Modell EC12 (2012): 5,5GHz CMOS Prozessor, 3 TB Ram
- Erstes Modell damals: 0,0018 MIPS, 8 KByte Ram
- Vergleich: Ein iPhone 5S schafft 18200 MIPS



Rechenleistung / Softwarenutzung

Heute vs. 2016?

# Rechenleistung / Softwarenutzung

	1-5	6-10	11-20	21-50	50++		
# Apps	2	2	6	8	9		
# App → daily use	146	11	1	0	0		
Erster PC	C64/Amiga	x86	Pentium 1-2	P3/4	callor multicore	core i3/5/7	
CPU	1	4	15	4	0	0	
Ram	kB	x MB	xx MB	xxx MB	x GB	xx GB	
OS	2	1	10	5	2	0	
	"ohne"	DOS	Win 3x	Win 9x	NT++	linux	?
	1	6	1	17	1	0	0
# Spiele	1-10	11-50	>50				
installiert	12	7	1				
	0	einige	"alle"				
online fähig	18	1	1				

Verwendete Betriebssysteme

Heute vs. 2016?

## Verwendete Betriebssysteme

Windows	linux	mac OS	Android	iOS
14	4	11	2	1

## Softwarearchitektur im Lauf der Zeit

Erste Beschreibung von "Dekomposition, Zerlegung, Entwurf"

- 1970er: Eher im Kontext von Hardware genutzt
- 1972: *"On the criteria to be used in decomposing systems into modules"* von D. L. Parnas
- 1975: *"The Mythical Man Month"* von Frederick Brooks

Softwarearchitektur im Lauf der Zeit

Unabhängiges Teilgebiet der Softwaretechnik

Konzept der Schnittstellen und Konnektoren

- 1992: "*Foundations for the Study of Software Architecture*" von Dewayne Perry und Alexander Wolf
- 1995: "*Software Architecture Analysis Method*" des Software Engineering Institute

## Softwarearchitektur im Lauf der Zeit

### Allgemeine Verbreitung und "Stand der Technik"

- 2000: *"IEEE 1471:2000 Norm Recommended Practice for Architectural Description of Software-Intensive Systems"*
- 2003: Zertifizierung als Softwarearchitekt durch die iSAQB (International Software Architect Qualification Board)
- 2003: UML 2.0 ist geeignet um Softwarearchitekturen zu beschreiben

# Pioniere der Softwarearchitektur



Pioniere der Softwarearchitektur

## David Parnas

\* 10. Februar 1941, New York

Erfinder des *Modulkonzepts* und des \*Geheimnisprinzips

Schaffung der Grundlage der *objektorientierten Programmierung*

Pioniere der Softwarearchitektur

## Frederick Brooks

\* 19. April 1931, North Carolina

Schrieb das erste 'ehrliche' Buch über Software-Projektmanagement

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

(The Mythical Man Month: Essays on Software Engineering)

Pioniere der Softwarearchitektur

## Tony Hoare

\* 11. Januar 1934, Sri Lanka

- Entwickler des Quicksort-Algorithmus
- Erfinder des Hoare-Kalküls zum Beweisen der Korrektheit von Algorithmen
- Entwickler der Prozessalgebra *Communicating Sequential Processes* (CSP), Grundlage der Programmiersprachen Ada, Occam und Go

## Pioniere der Softwarearchitektur

# Edsge Dijkstra

\* 11. Mai 1930, Rotterdam

- Entwickler des Dijkstra-Algorithmus zur Berechnung eines kürzesten Weges in einem Graphen
- Einführung von Semaphoren zur Synchronisation zwischen Threads
- Entwicklung des Shunting-yard-Algorithmus zur Übertragung der Infixnotation in einen abstrakten Syntaxbaum
- Entwickler des Multitasking-Betriebssystems THE, erste dokumentierte Schichtenstruktur
- Mitentwickler von Algol 60 - Schrieb den ersten Compiler dafür

Prägung der Begriffe der strukturierten Programmierung der *Softwarekrise*

## Pioniere der Softwarearchitektur

# Per Brinch Hansen

\* 13. November 1938 in Frederiksberg

- Entwickler des RC-4000-Minicomputer und dessen Betriebssystems (1969): Erste Implementierung des Mikrokern-Konzepts
- Erfinder des Monitor-Konzepts für das Concurrent Programming
- Entwickler von *Concurrent Pascal*, der ersten nebenläufigen Programmiersprache
- Entwickler von *SuperPascal* zur Darstellung paralleler Algorithmen
- Von Per Brinch Hansen stammt die dänische Bezeichnung *Datamat* für Computer

Pioniere der Softwarearchitektur

## Friedrich Bauer

\* 10. Juni 1924 in Regensburg

- Erfinder des Stack-Konzepts ("Kellerspeichers")
- Hielt 1967 an der Technischen Universität München die erste offizielle Informatikvorlesung in Deutschland
- Ausrichter der ersten Computerausstellung im Deutschen Museum 1988
- Autor mehrerer Standardwerke zur Kryptologie

## Pioniere der Softwarearchitektur

# Niklaus Wirth

\* 15. Februar 1934 in Winterthur

- Erfinder des *Wirthschen Gesetzes*, nach dem sich die Software schneller verlangsamt als sich die Hardware beschleunigt.
- Mitentwickler Programmiersprache Euler
- Entwickler der Programmiersprache PL360, die 1968 auf dem System IBM /360 implementiert wurde
- Mitentwickler der Programmiersprache Algol
- Entwickler der Programmiersprache Pascal
- Erweiterung der formalen Sprache Backus-Naur-Form (BNF), zur Erweiterten Backus-Naur-Form (EBNF)
- Entwickler von Modula, Modula-2 und Oberon (1985–1990)  
"Importierte" 1980 eine der ersten Computermäuse nach Europa, was zur Gründung von Logitech führte

Tools und Frameworks im Laufe der Zeit

## Entwicklung des linux-kernels

- 1992: V 0.0.1 / 8k LOC / 230 kB
- 1994: V 1.0.0 / 170k LOC / 1.2 MB
- 1996: V 2.0.0 / 716k LOC / 5.8 MB
- 2011: V 3.0.0 / 14.6 Mio. LOC / 96 MB
- 2015: V 4.0.0 / 19.3 Mio. LOC / 78 MB
- 2018: V 4.1.14 / 23.1 Mio. LOC



Tools und Frameworks im Laufe der Zeit

## Entwicklung von Ruby on Rails

- 2005: V 1.0.0 / 96k LOC / 3365 Klassen / 8523 Methoden
- 2007: V 2.0.0 / 170k LOC / 5255 Klassen / 13260 Methoden
- 2010: V 3.0.0 / 230k LOC / 8334 Klassen / 19785 Methoden
- 2013: V 4.0.0 / 317k LOC / 9430 Klassen / 24143 Methoden

Was ist Softwarearchitektur?

Geschichte und Trends

## Sichten auf Architekturen

Qualität und andere nichtfunktionale Anforderungen

Architekturmuster

Dokumentation von Architekturen

Technologien und Frameworks

# Wiederholung

## Sichten auf Architekturen

# Warum überhaupt Sichten?

*"Es ist eine offensichtliche Wahrheit, dass auch eine perfekte Architektur nutzlos bleibt, wenn sie nicht verstanden wird..."*

Felix Bachmann und Len Bass in "Software Architecture  
Documentation in Practice"

# 1.

Eine einzelne Darstellung kann die Vielschichtigkeit und Komplexität einer Architektur nicht ausdrücken.

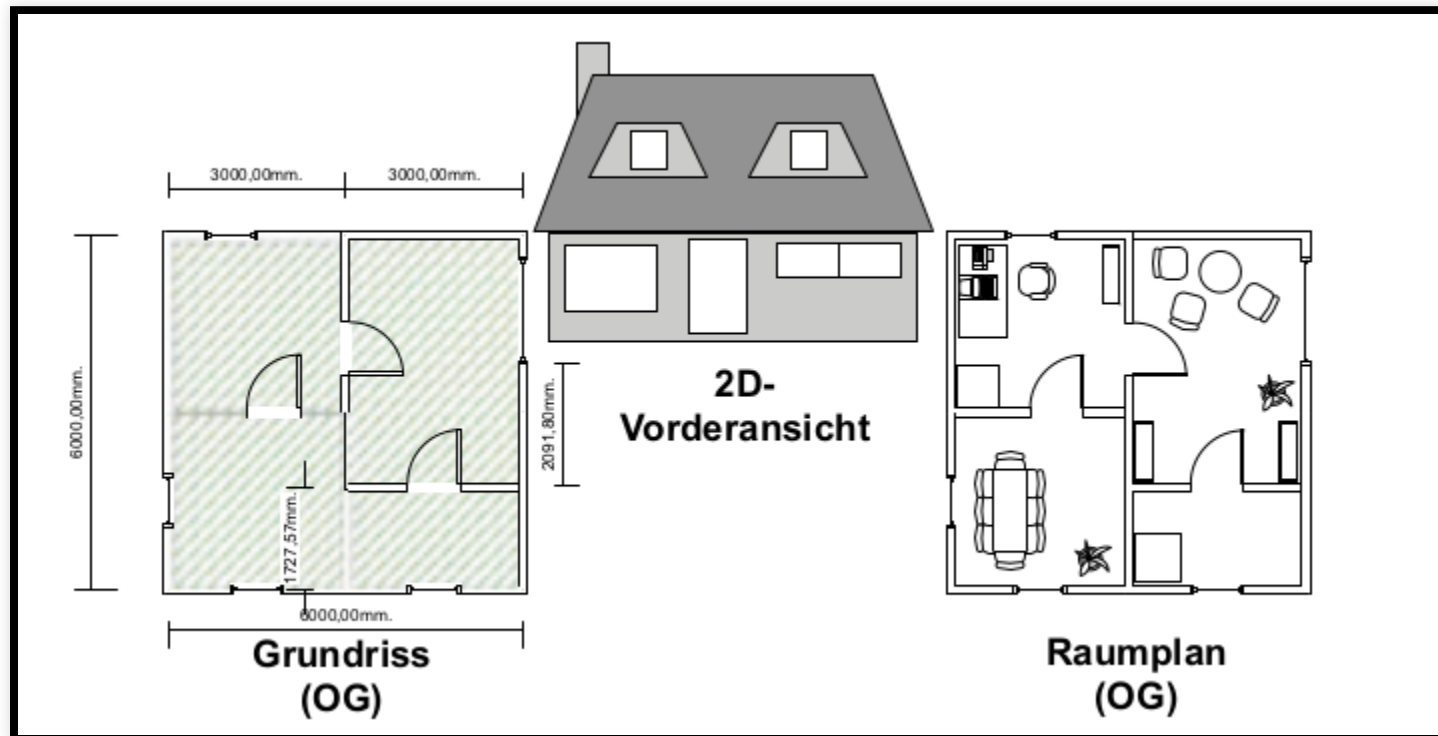
- Genauso wenig, wie man nur mit einem Grundriss ein Haus bauen kann.

## 2.

Sichten ermöglichen die Konzentration auf einzelne Aspekte des Gesamtsystems und reduzieren somit die Komplexität der Darstellung.

3.

Die Projektbeteiligten haben ganz unterschiedliche Informationsbedürfnisse.



Bildquelle: "Effektive Softwarearchitekturen" von Gernot Starke

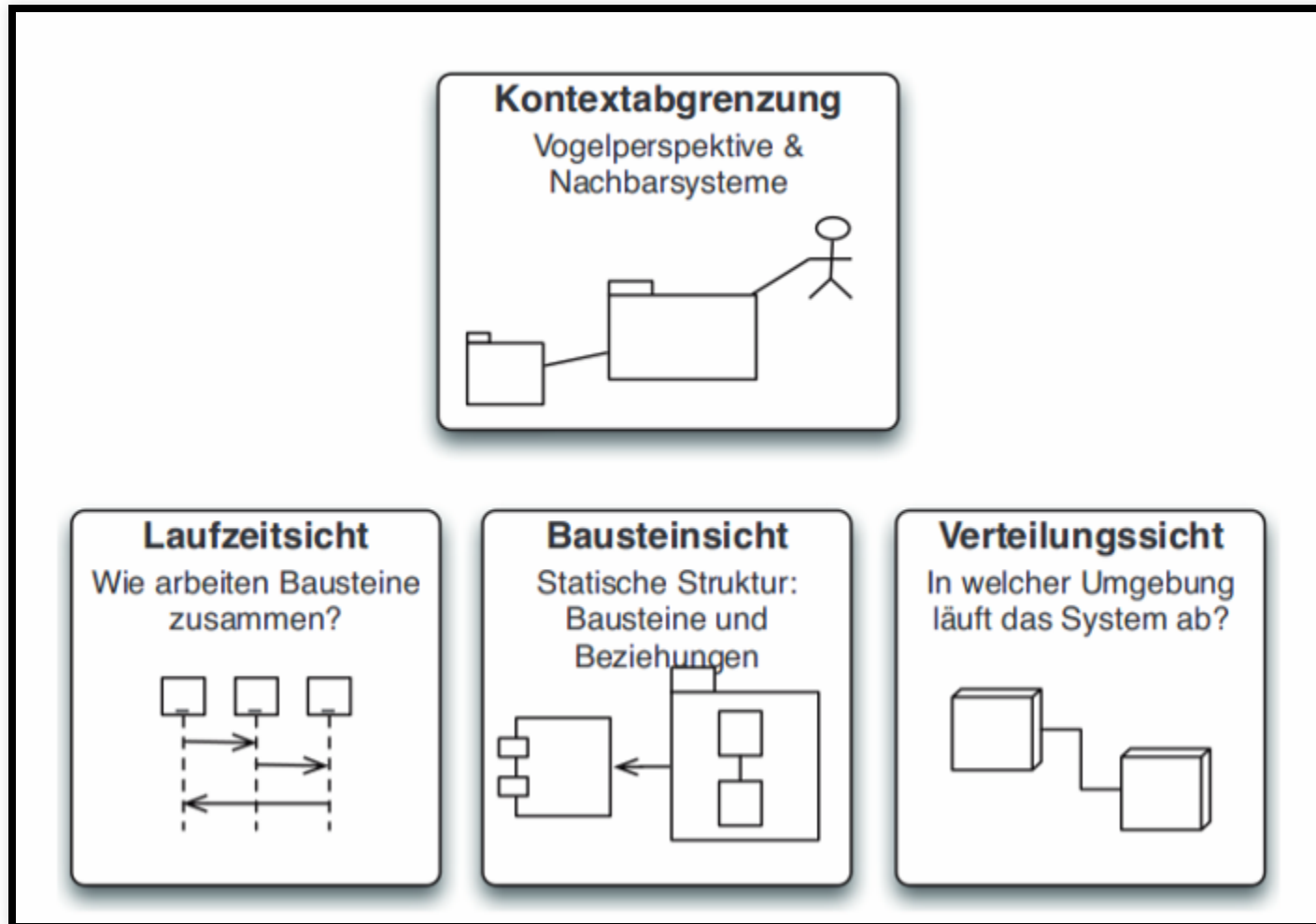


Architekten müssen Projektbeteiligten die Architektur erklären bzw. sie verteidigen/vermarkten

- die entworfenen Strukturen
- die getroffenen Entscheidungen
- ihre Konzepte + Begründungen + Vor- und Nachteile

→ Mit Hilfe von unterschiedlichen Sichten lassen sich viele Aspekte von Architektur verständlich darstellen.

# Überblick über die vier Sichten



Bildquelle: "Effektive Softwarearchitekturen" von Gernot Starke

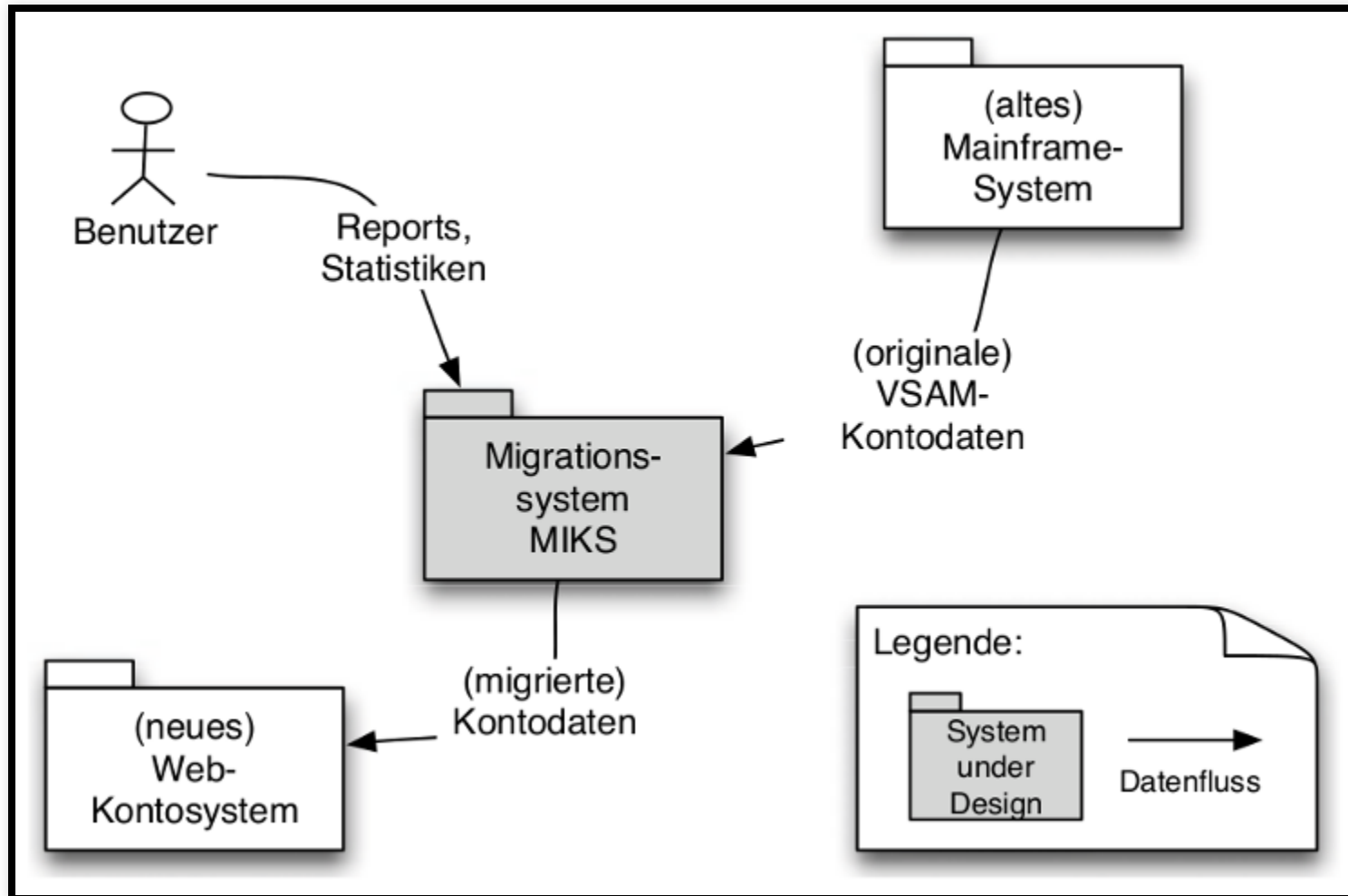
# Kontextsicht

- Wie ist das System in seine Umgebung eingebettet?
- zeigt das System als Blackbox in seinem Kontext aus der Vogelperspektive

## Kontextsicht - Enthaltene Informationen:

- Schnittstellen zu Nachbarsystemen
- Interaktion mit wichtigen Stakeholdern
- wesentliche Teile der umgebenden Infrastruktur

## Kontextsicht - Beispiel



Bildquelle: "Effektive Softwarearchitekturen" von Gernot Starke

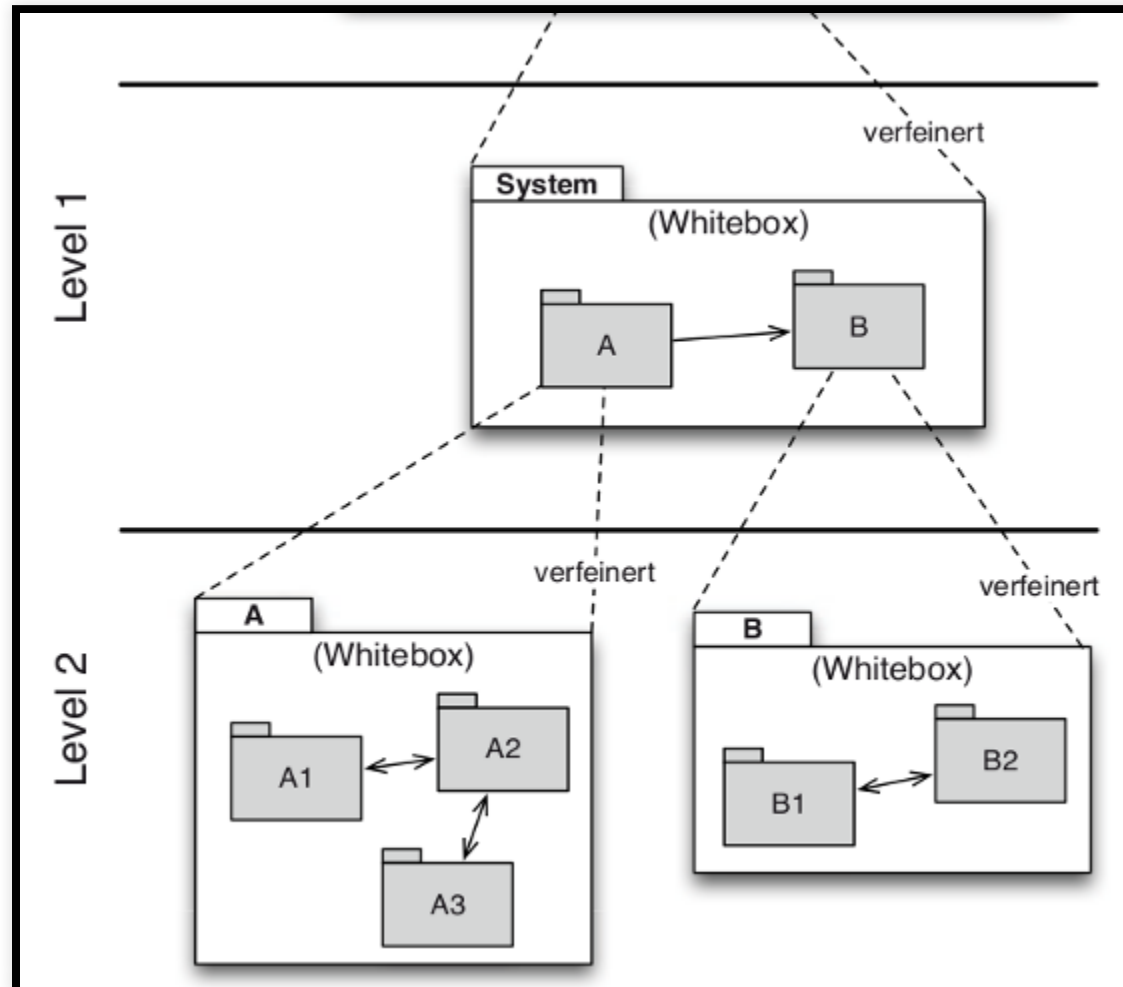
# Bausteinsicht

- Wie ist das System intern aufgebaut?
- unterstützt Auftraggeber und Projektleiter bei der Projektüberwachung
- dient der Zuteilung von Arbeitspaketen
- dient als Referenz für Software-Entwickler

## Bausteinsicht - Enthaltene Informationen:

- statische Strukturen der Bausteine des Systems
- Subsysteme
- Komponenten und deren Schnittstellen

# Bausteinsicht - Beispiel



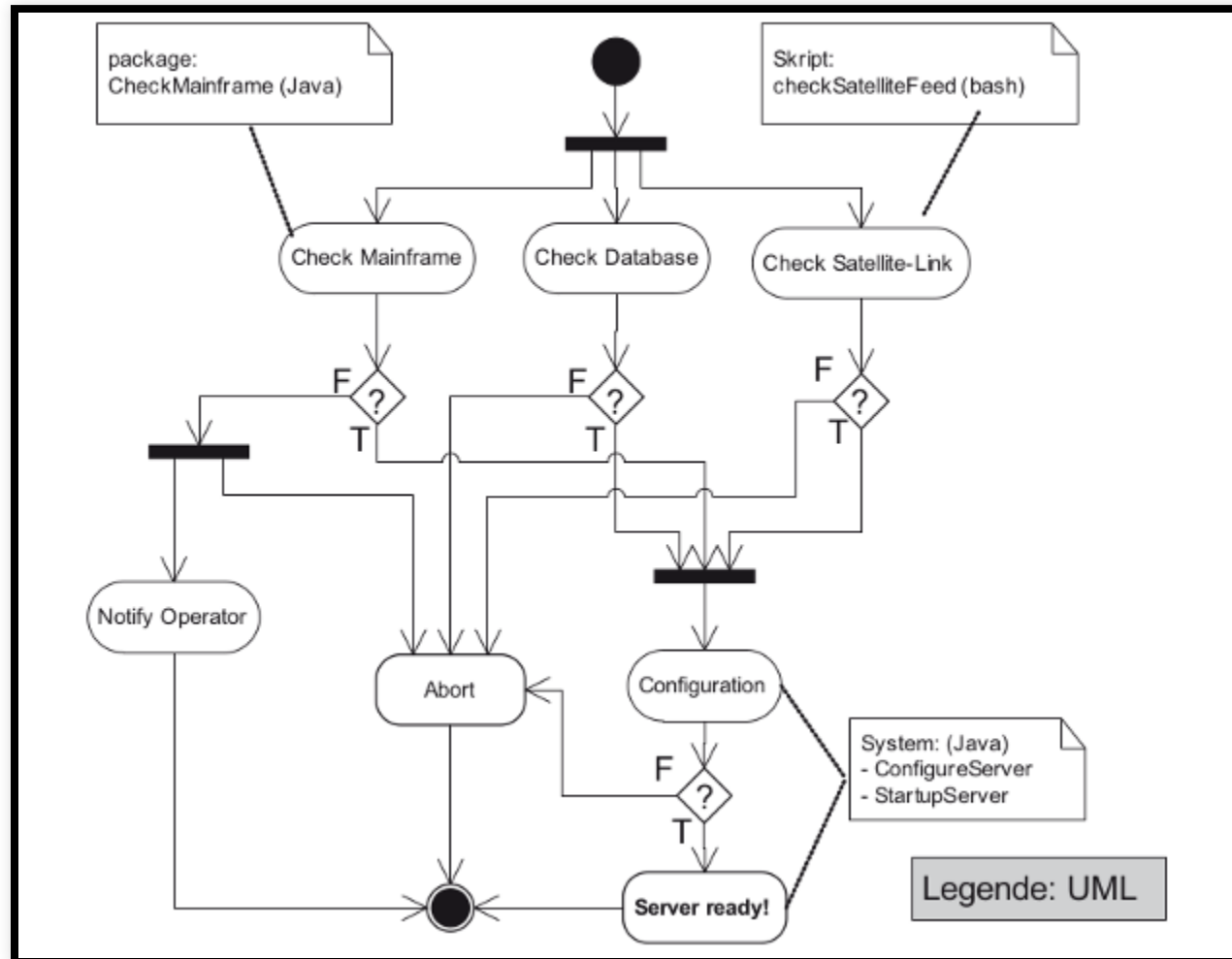
Bildquelle: "Effektive Softwarearchitekturen" von Gernot Starke



# Laufzeitsicht

- Wie läuft das System ab?
- Welche Bausteine des Systems existieren zur Laufzeit?
- Wie wirken die Bausteine zusammen?

# Laufzeitsicht - Beispiel



Bildquelle: "Effektive Softwarearchitekturen" von Gernot Starke

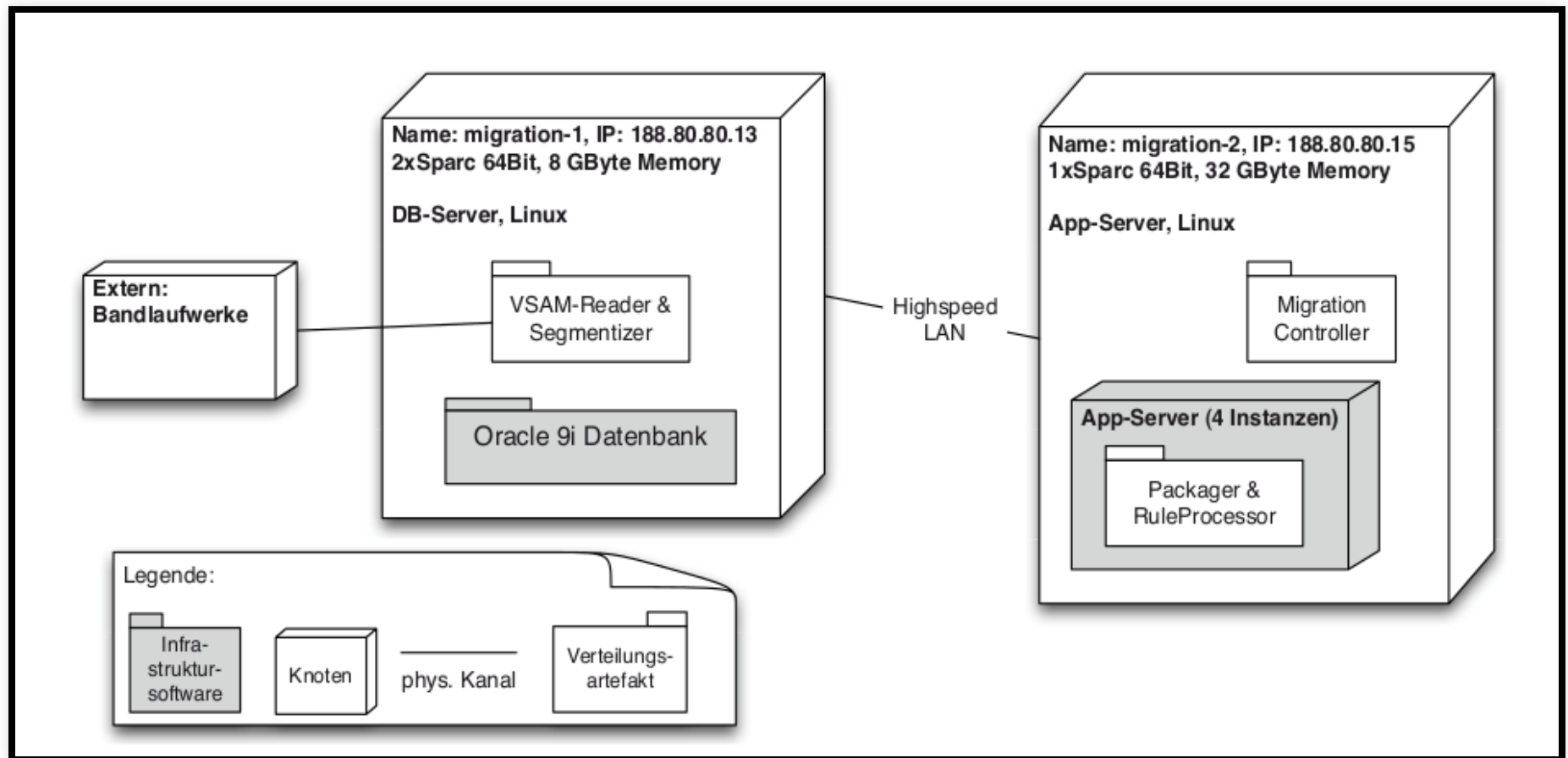
# Verteilungssicht / Infrastruktursicht

- In welcher Umgebung läuft das System ab?
- zeigt das System aus Betreibersicht

## Verteilungssicht - Enthaltene Informationen:

- Hardwarekomponenten: Rechner, Prozessoren
- Netztopologien
- Netzprotokolle
- sonstige Bestandteile der physischen Systemumgebung

## Verteilungssicht - Beispiel



Bildquelle: "Effektive Softwarearchitekturen" von Gernot Starke

Gibt es noch weitere Sichten?

# Empfehlung

- Verzichten Sie möglichst auf weitere Sichten
- Jede Sicht kostet Erstellungs- und Wartungsaufwand
- Die grundlegenden Aspekte der Architektur- und Systementwicklung decken die vier Sichten ab

## Beispiel (nach Peter Hruschka):

Beim Häuserbau könnten Kakteen- und Orchideenzüchter nach der Sonneneinstrahlung in einzelnen Räumen fragen und zum Wohle ihrer pflanzlichen Lieblinge einen gesonderten Plan wünschen. Wie groß ist Ihrer Erfahrung nach die Zahl derer, die beim Bau oder beim Kauf einer Immobilie diese „pflanzliche“ Sicht als Entscheidungskriterium verwenden?



Ok, aber gibt es noch trotzdem noch weitere Sichten?

## Die Standpunktmenge von Clements et al.

- Modul-Standpunkt (*module view type*)
- Komponenten-und-Konnektoren-Standpunkt (*components and connectors view type*)
- Zuordnungs-Standpunkt (*allocation view type*)

# Die „4+1“-Standpunktmenge von Kruchten

- Logischer Standpunkt
- Prozess-Standpunkt
- Physischer Standpunkt
- Entwicklungs-Standpunkt

# Die Standpunktmenge von Hofmeister, Nord und Soni

- Konzeptioneller Standpunkt
- Modul-Standpunkt
- Ausführungs-Standpunkt
- Quelltext-Standpunkt

# Die Standpunktmenge von Reussner

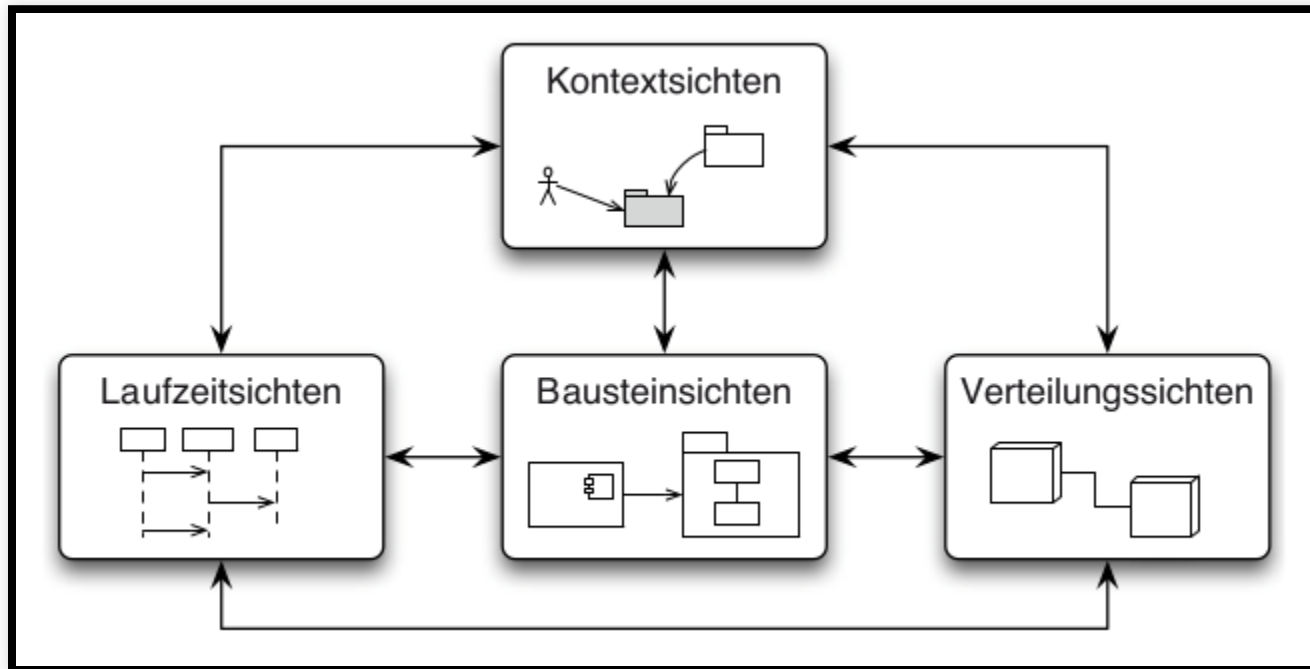
- Statischer Standpunkt
- Dynamischer Standpunkt
- Verteilungs-Standpunkt

Entwurf der Sichten

## Wie sollten die Sichten entstehen?

Der Entwurfsprozess der Sichten wird von deren starken Wechselwirkungen und Abhängigkeiten geprägt. Software-Architekturen sollten daher iterativ entstehen, weil die Auswirkungen mancher Entwurfsentscheidungen erst über die Grenzen von Sichten hinweg spürbar werden.

# Wechselwirkungen



Bildquelle: "Effektive Softwarearchitekturen" von Gernot Starke



# In welcher Reihenfolge sollten die Sichten entstehen?

Letztlich spielt es kaum eine Rolle, mit welcher Architektursicht Sie beginnen.  
Im Laufe des Entwurfs der Software-Architektur werden Sie an allen Sichten  
nahezu parallel arbeiten oder häufig zwischen den Sichten wechseln.

Beginnen Sie mit einer **Bausteinsicht**, wenn Sie

- bereits ähnliche Systeme entwickelt haben und eine genaue Vorstellung von benötigten Implementierungskomponenten besitzen
- ein bereits teilweise bestehendes System verändern müssen und damit Teile der Bausteinsicht vorgegeben sind

Beginnen Sie mit einer Laufzeitsicht, wenn Sie

- bereits eine erste Vorstellung wesentlicher Architekturbausteine besitzen und deren Verantwortlichkeit und Zusammenspiel klären wollen

Beginnen Sie mit einer Verteilungssicht, wenn Sie

- viele Randbedingungen und Vorgaben durch die technische Infrastruktur, das Rechenzentrum oder den Administrator des Systems bekommen

## Wie viel Aufwand für welche Sicht?

Rechnen Sie damit, dass Sie 60 bis 80% der Zeit, die Sie für den Entwurf der Architektursichten insgesamt benötigen, alleine für die Ausgestaltung der Bausteinsicht aufwenden. Der ausschlaggebende Grund hierfür: Die Bausteinsicht wird oftmals wesentlich detaillierter ausgeführt als die übrigen Sichten.

Dennoch sind die übrigen Sichten für die Software-Architektur und das Gelingen des gesamten Projektes wichtig! Lassen Sie sich von diesem relativ hohen Aufwand für die Bausteinsicht in keinem Fall dazu verleiten, die anderen Sichten zu ignorieren.

*Quelle: Starke/Effektive Softwarearchitekturen*

## Wechselwirkungen dokumentieren

Sie sollten in Ihren Architekturbeschreibungen die Entwurfsentscheidungen dokumentieren, die besonderen Einfluss auf andere Sichten haben. Beispielsweise bestimmt die Entscheidung für eine zentrale Datenhaltung in der Bausteinsicht maßgeblich den Aufbau der technischen Infrastruktur

## Wechselwirkungen dokumentieren

- Bessere Nachvollziehbarkeit von Entscheidungen
- Auswirkungen von Änderungen werden vereinfacht
- Das Verständnis der Architekturbeschreibung wird einfacher, da die Zusammenhänge zwischen den Sichten klarer werden

Entwurf der Sichten

Entwurf der Kontextabgrenzung



## Entwurf der Kontextabgrenzung

- Im Idealfall ist die Kontextabgrenzung ein Ergebnis der Anforderungsanalyse
- Zeigen Sie in sämtliche Nachbarsysteme
- Alle ein- und ausgehenden Daten und Ereignisse müssen in der Kontextabgrenzung zu erkennen sein

Entwurf der Sichten

Entwurf der Bausteinsicht

## Entwurf der Bausteinsicht

- Der Entwurf der Bausteinsicht ist der Kern der Architekturbeschreibung
  - Beschreiben Sie exakt, wie das System (strukturell) aufgebaut ist und aus welchen Bausteinen es besteht
  - Beginnen Sie mit einer Vogelperspektive der Implementierungsbausteine
  - Zerlegen Sie Ihr System dazu in große Architekturelemente, wie Sub- oder Teilsysteme
- Erinnerung: Der Aufwand macht 60%-80% der Architekturarbeit aus

Entwurf der Sichten

Entwurf der Laufzeitsicht

## Entwurf der Laufzeitsicht

- Elemente der Laufzeitsichten sind Instanzen der statischen Architekturbausteine, die Sie in den Bausteinsichten dokumentieren
- Ein möglicher Weg zur Laufzeitsicht führt daher über die Bausteinsichten
- Beschreiben Sie die Dynamik der statischen Bausteine, beginnend bei den wichtigsten Use-Cases des Gesamtsystems.
- Einen weiteren Startpunkt kann die Verteilungs-/Infrastruktursicht bilden

Entwurf der Sichten

Entwurf der Verteilungssicht

## Entwurf der Verteilungssicht

- Die Verteilungssicht sollte eine Landkarte der beteiligten Hardware und der externen Systeme beinhalten
- Genügen die verfügbare Hardware und die Kommunikationskanäle, oder gibt es potenzielle Engpässe?
- Falls Ihre Systeme in verteilten Umgebungen ablaufen, sollten Sie vorhandene Kommunikationsmechanismen, Protokolle und Middleware in die Infrastruktursicht aufnehmen

Was ist Softwarearchitektur?

Geschichte und Trends

## Sichten auf Architekturen

Qualität und andere nichtfunktionale Anforderungen

Architekturmuster

Dokumentation von Architekturen

Technologien und Frameworks

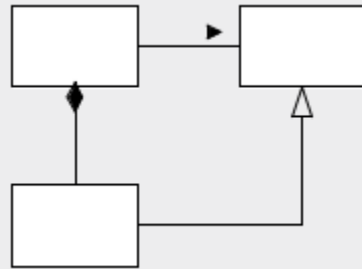


# UML für Softwarearchitekten

UML für Softwarearchitekten

# UML Diagrammtypen

# Strukturdiagramme



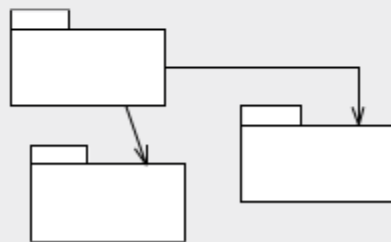
## **Klassendiagramm:**

Zeigt die statische Struktur von Klassen und deren Beziehungen (Assoziationen, Aggregationen sowie Spezialisierungen/Generalisierungen).



## **Objektdiagramm:**

Zeigt eine Struktur von Instanzen sowie deren Verbindungen. Bildet damit einen Schnappschuss des Klassendiagramms.

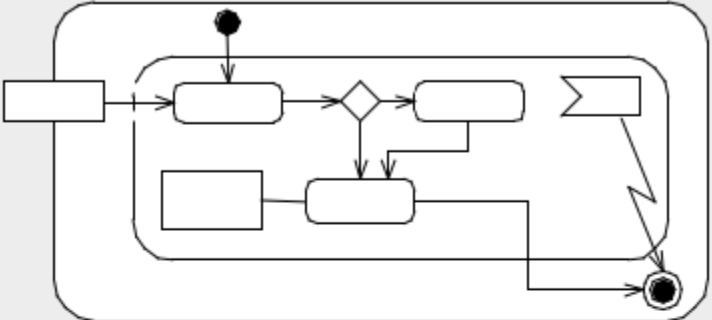
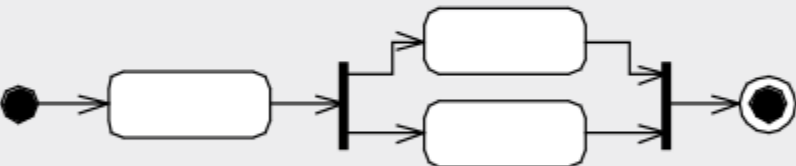
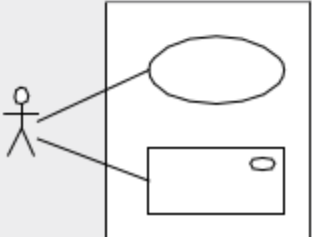


## **Paketdiagramm:**

Gibt einen Überblick über die Zerlegung des Gesamtsystems in Pakete oder Teilsysteme. Enthält logische Zusammenfassungen von Systemteilen.

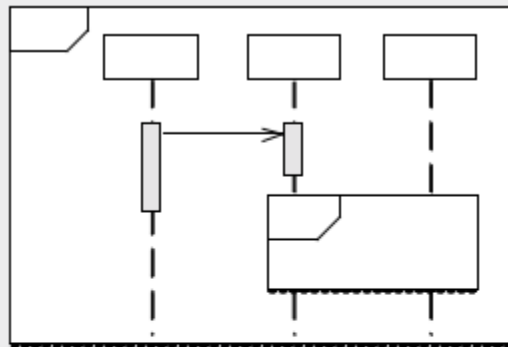
Bildquelle: "Effektive Softwarearchitekturen" von Gernot Starke

# Verhaltensdiagramme

Verhaltensdiagramme	
 A UML Activity Diagram showing a process flow. It starts with a start node (black dot) leading into a large rounded rectangle (activity boundary). Inside, there's a smaller rounded rectangle (compartment) containing several nodes: a start node, a rectangular node, a diamond-shaped decision node, another rectangular node, and a final node (bullseye). Arrows indicate the flow between these nodes, including a loop and a path that exits the compartment and returns to the start of the activity.	<p><b>Aktivitätsdiagramm:</b> Zeigt mögliche Abläufe innerhalb von Systembestandteilen (etwa: Klassen, Komponenten oder Use-Cases). Kann Algorithmen, Daten- und Kontrollflüsse sehr detailliert beschreiben.</p>
 A UML State Machine Diagram. It begins with a start node (black dot) leading to a state (rounded rectangle). This state transitions to a fork node (vertical bar), which then splits into two parallel states (rounded rectangles). These two states converge at a join node (vertical bar), which then leads to a final state (bullseye).	<p><b>Zustandsdiagramm</b> (<i>State Diagram</i>): Zeigt die möglichen Zustände eines Systembestandteils sowie die erlaubten Zustandsübergänge an und verknüpft Aktivitäten mit diesen Zuständen und Übergängen.</p>
 A UML Use Case Diagram. It features a stick figure actor on the left connected by lines to two use cases inside a rectangular system boundary. One use case is represented by an oval, and the other by a rectangle with a small circle inside.	<p><b>Anwendungsfalldiagramm</b> (<i>Use Case Diagram</i>): Zeigt eine Übersicht über alle Prozesse, mit denen das System auf Wünsche von Akteuren (oder Nachbarsystemen) reagiert.</p>

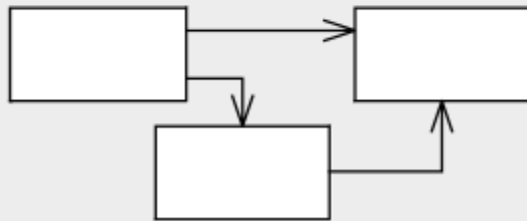
Bildquelle: "Effektive Softwarearchitekturen" von Gernot Starke

# Interaktionsdiagramme



## **Sequenzdiagramm:**

Zeigt den Nachrichtenaustausch zwischen Instanzen von Systembestandteilen für einzelne Szenarien.  
Seit UML 2 schachtelbar.

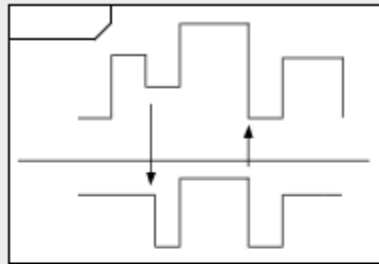


## **Kommunikationsdiagramm:**

In früheren UML Versionen Kollaborationsdiagramm genannt, zeigt die Zusammenarbeit zwischen Instanzen von Systembestandteilen.

Bildquelle: "Effektive Softwarearchitekturen" von Gernot Starke

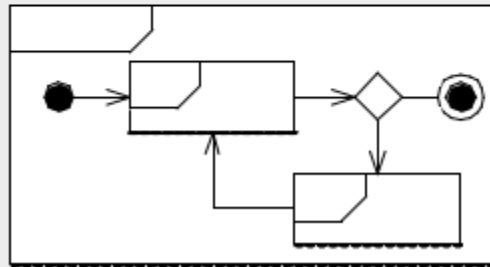
# Interaktionsdiagramme



## **Zeitverhaltensdiagramm**

*(Timing-Diagram):*

Zeigt Zeitabhängigkeiten zwischen Ereignissen beschreibt Zustandsänderungen in Abhängigkeit vom Zeitverlauf.



## **Interaktionsübersichtsdiagramm:**

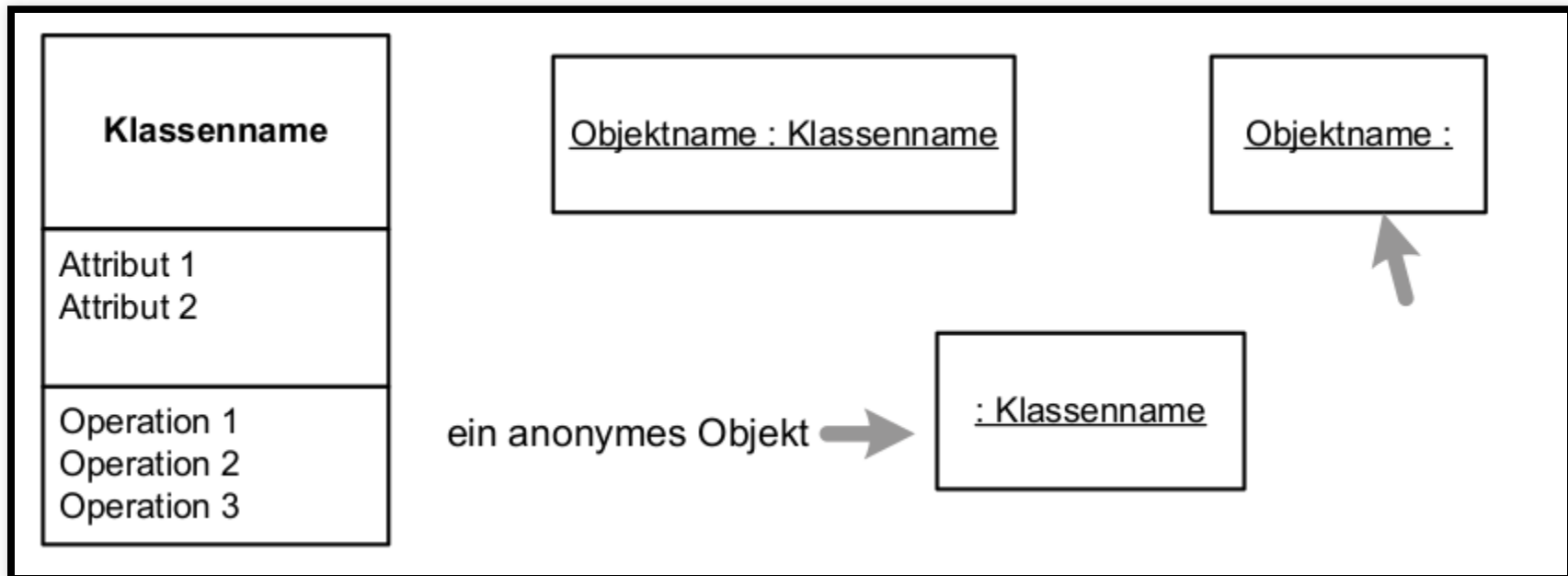
Zeigt das Zusammenspiel verschiedener Interaktionen und besteht in der Regel aus Referenzen auf andere Interaktionsdiagramme. Ein Art Aktivitätsdiagramm auf hoher Abstraktionsebene.

Bildquelle: "Effektive Softwarearchitekturen" von Gernot Starke

UML für Softwarearchitekten

# UML Klassen und Objekte

# Klassen und Objekte



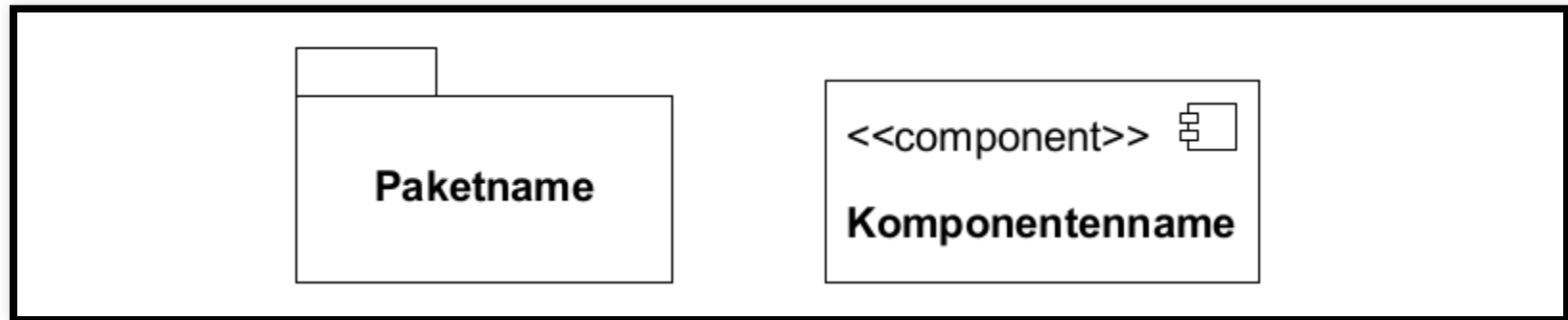
Bildquelle: "Effektive Softwarearchitekturen" von Gernot Starke



UML für Softwarearchitekten

# UML Pakete und Komponenten

## Pakete und Komponenten

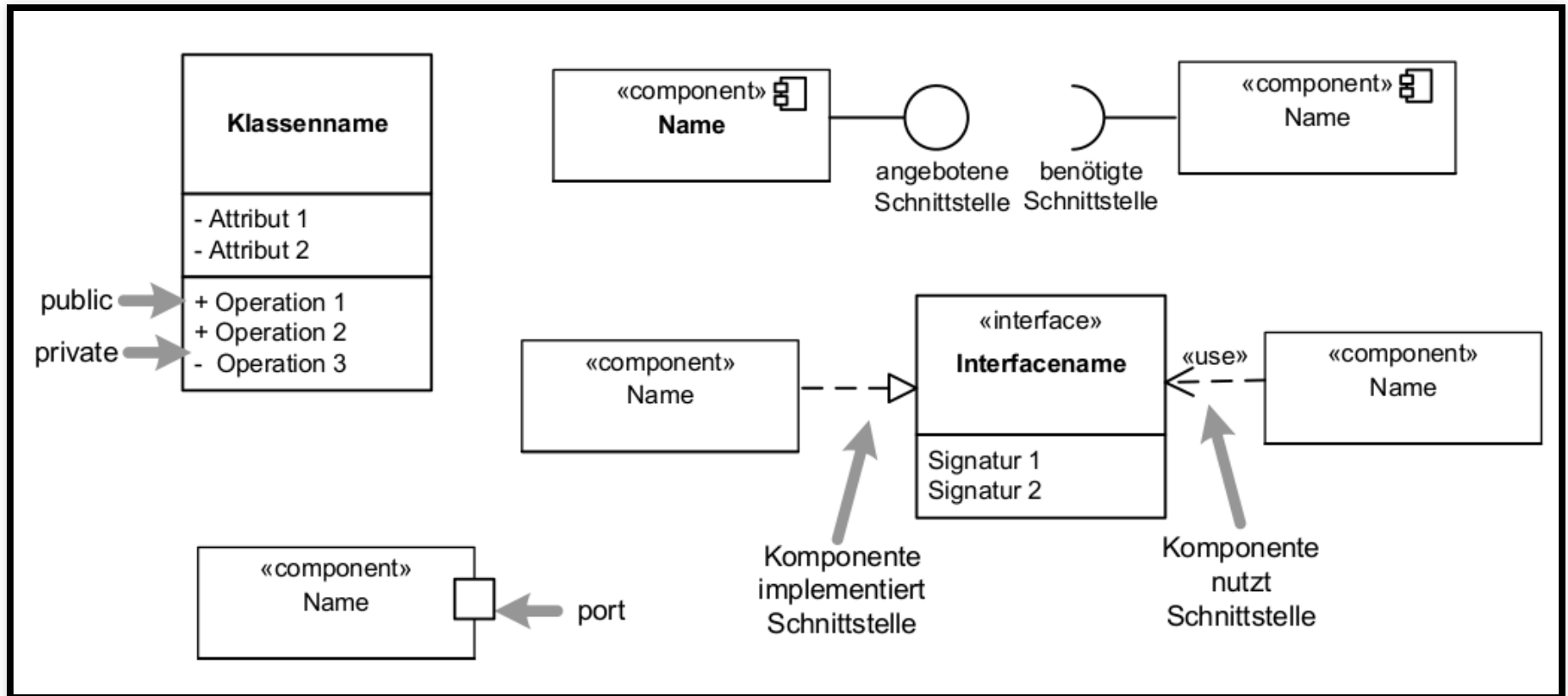


Bildquelle: "Effektive Softwarearchitekturen" von Gernot Starke

UML für Softwarearchitekten

UML Schnittstellen

# Schnittstellen



Bildquelle: "Effektive Softwarearchitekturen" von Gernot Starke

UML für Softwarearchitekten

Welches UML Diagramm für welche Sicht?

## UML für Softwarearchitekten

# Baustein-Sicht

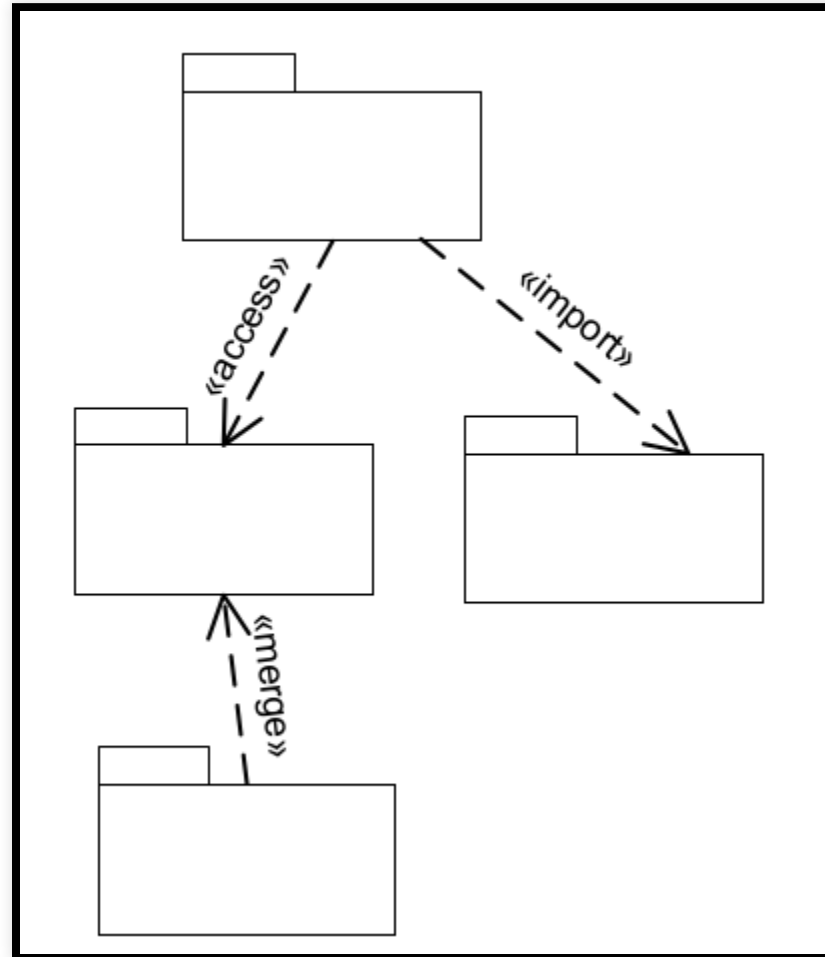
- Gute Namen wählen!
- Rollennamen anzugeben, Navigationsrichtung vorschreiben und Multiplizitäten festlegen
- Verwenden Sie nur eine Art von Schnittstellendarstellung
- Nutzen Sie Stereotypes für verschiedene Arten von fachlichen und technischen Klassen und Komponenten

UML für Softwarearchitekten

## Baustein-Sicht

- Paketdiagramm
- Komponentendiagramm
- Klassendiagramm
- Aktivitätsdiagramm
- Zustandsdiagramm

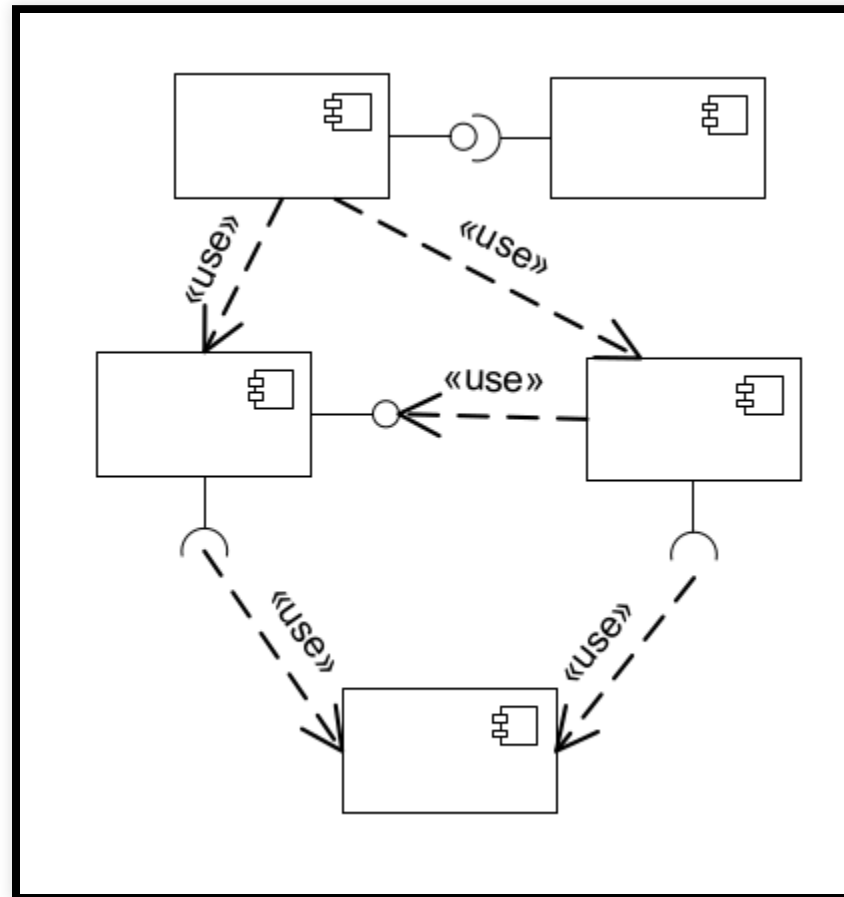
# Bausteinsicht in UML



*Paketdiagramm*

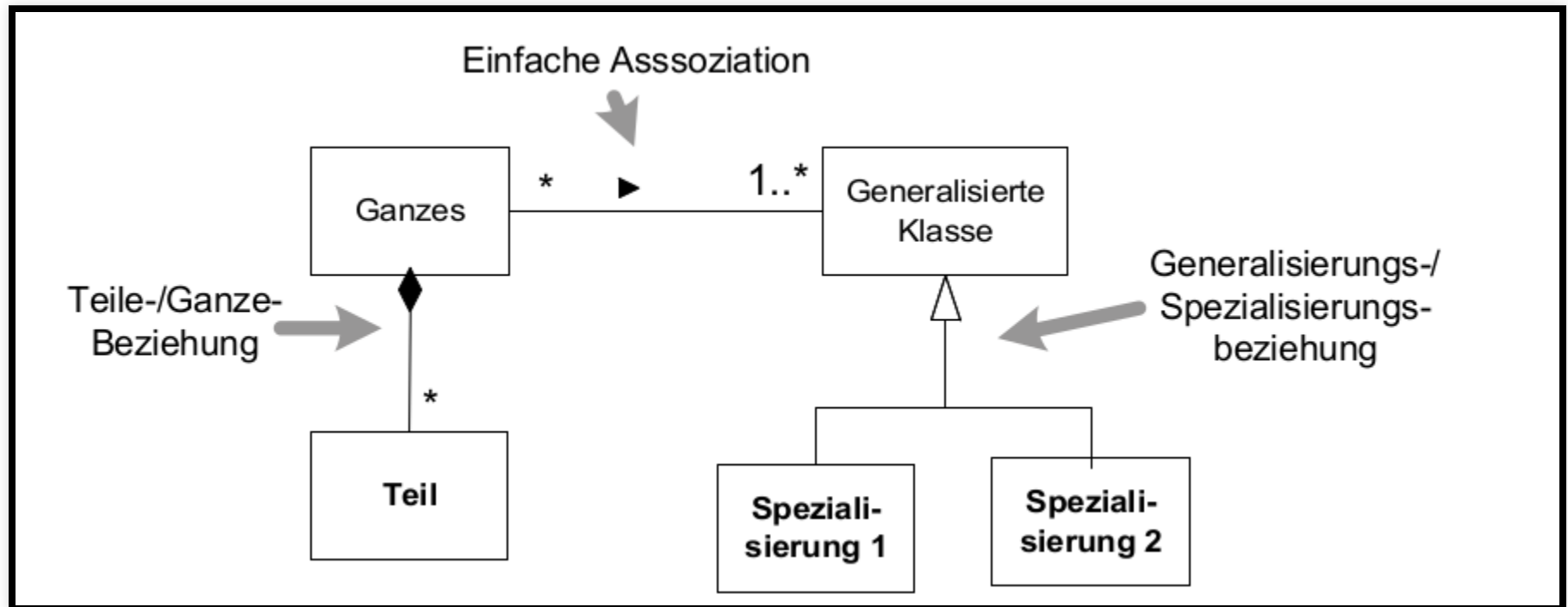


# Bausteinsicht in UML



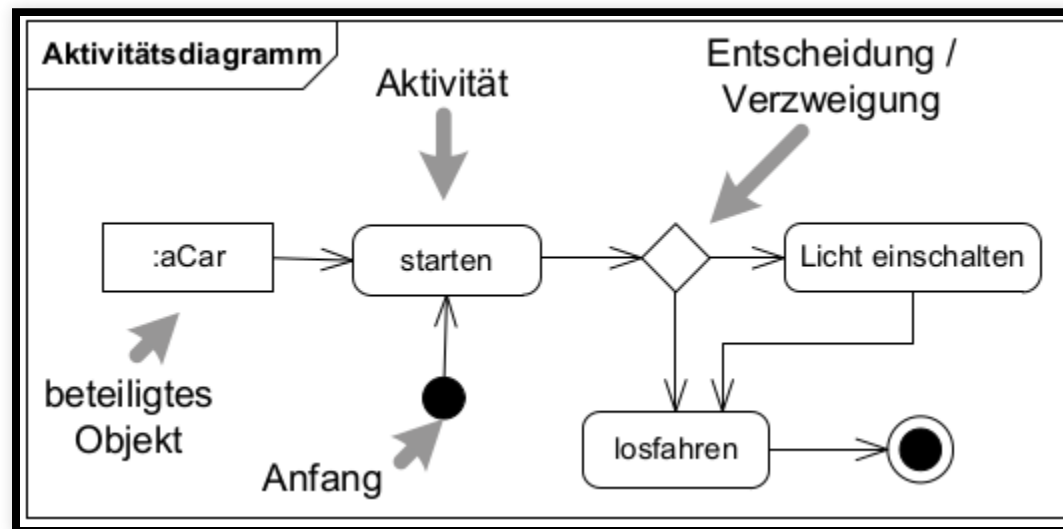
*Komponentendiagramm*

## Bausteinsicht in UML



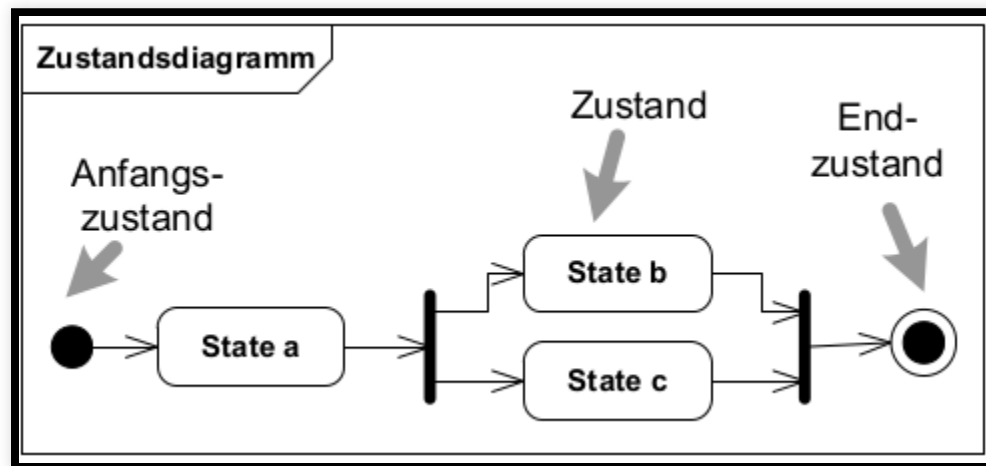
*Klassendiagramm*

## Bausteinsicht in UML



*Aktivitätsdiagramm*

## Bausteinsicht in UML



*Zustandsdiagramm*

UML für Softwarearchitekten

## Verteilungs-Sicht

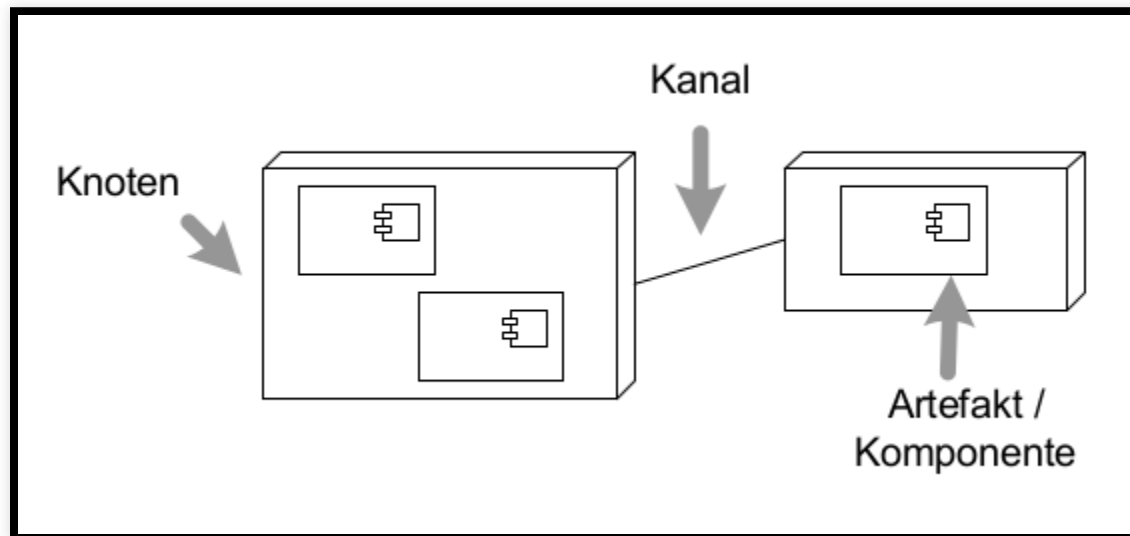
- Hauptelemente: Knoten und Kanäle zwischen den Knoten
- Knoten sind Standorte, z.B. Cluster, Rechner, Chips, ...
- Kanäle sind die physikalischen Übertragungswege, z.B. Kabeln, Bluetooth, Wireless, ...

UML für Softwarearchitekten

## Verteilungs-Sicht

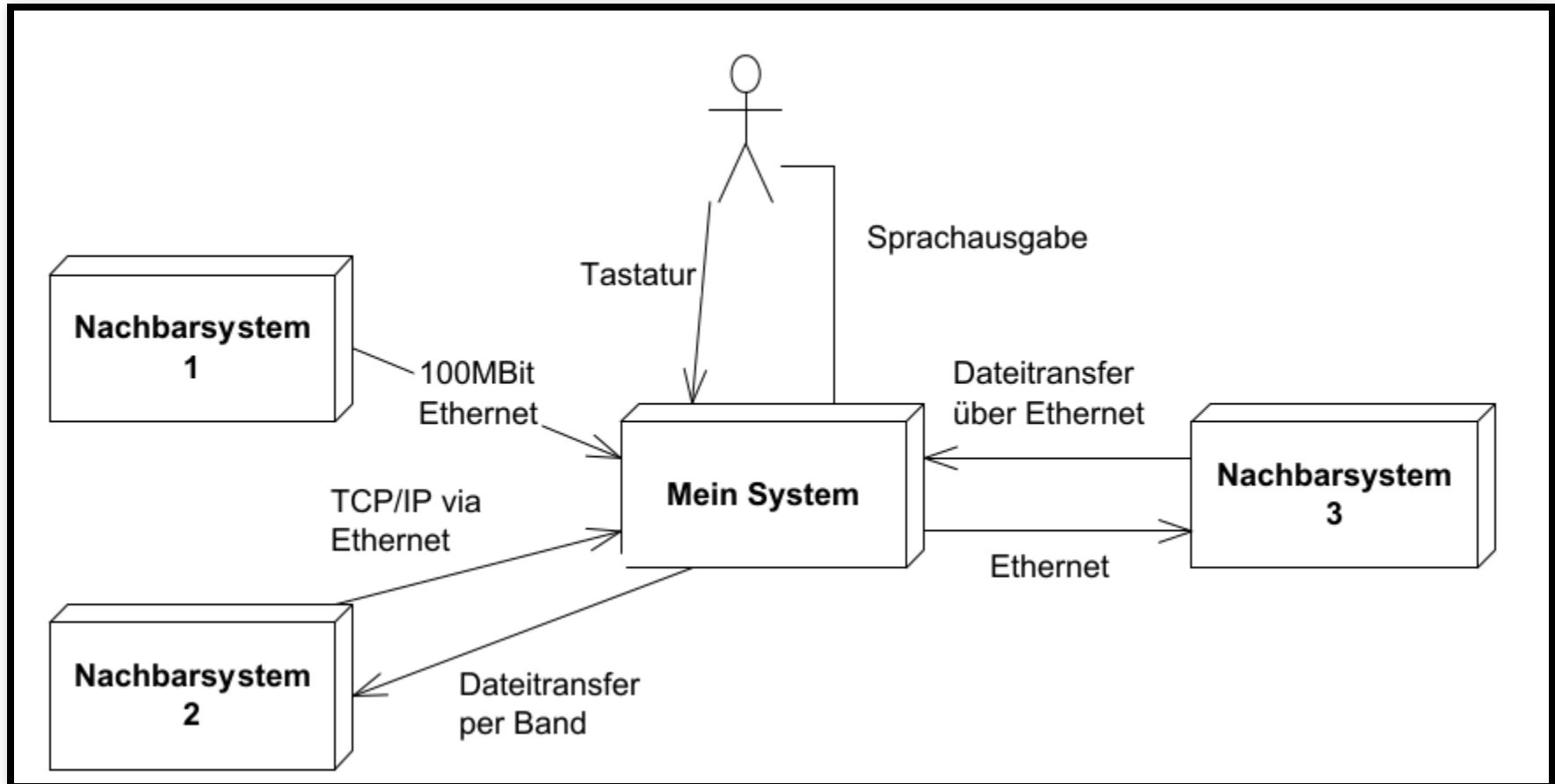
- Verteilungsdiagramm
- Kontextdiagramm

## Verteilungs-Sicht in UML



*Verteilungsdiagramm*

# Verteilungs-Sicht in UML



*Kontextdiagramm*



## UML für Softwarearchitekten

### Laufzeitsicht

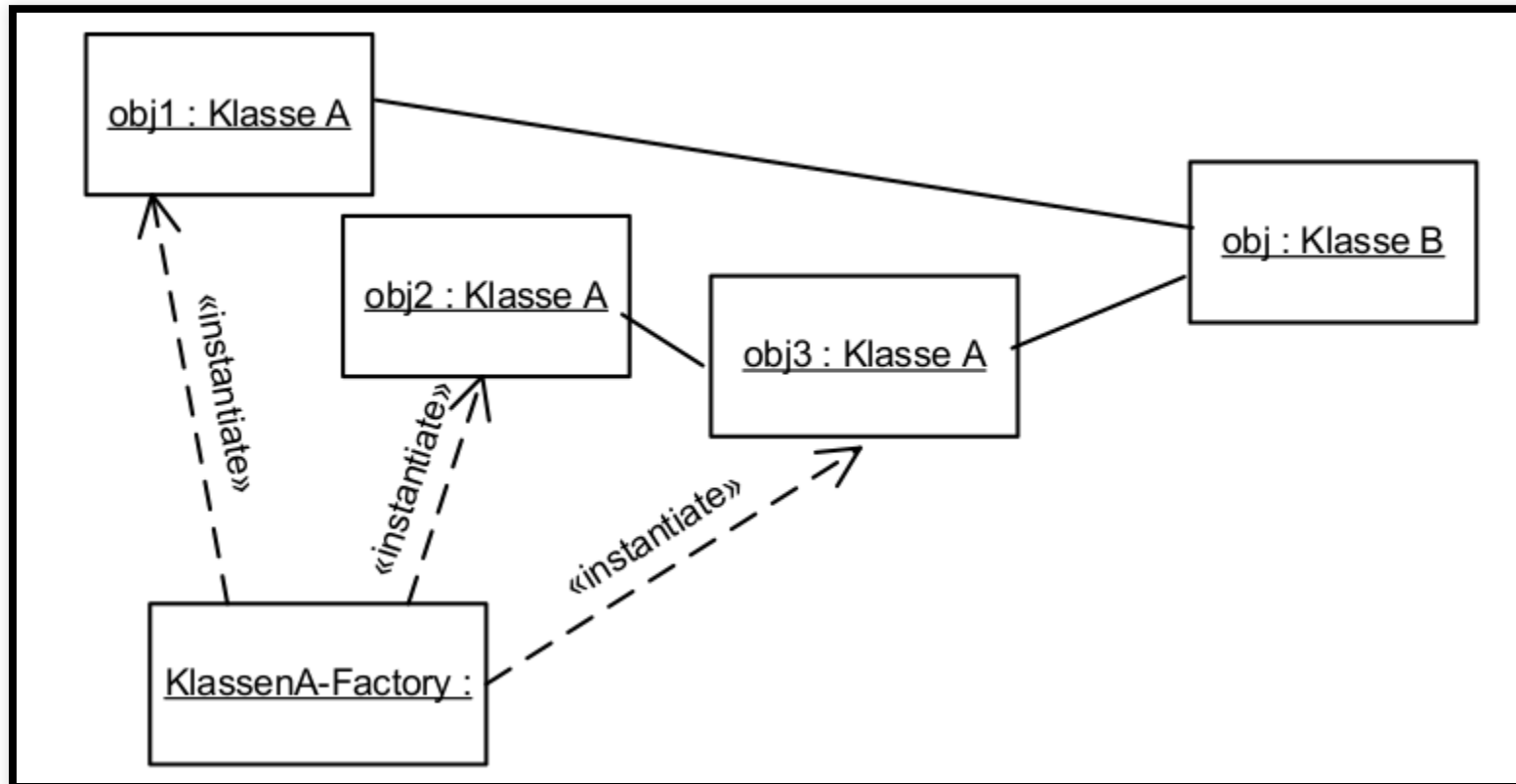
- Elemente der Laufzeitsicht sind immer um Instanzen von Bausteinen, die in der Bausteinsicht enthalten sind, also um Objekte zu den Klassen oder um instanziierte Komponenten.

UML für Softwarearchitekten

## Laufzeitsicht

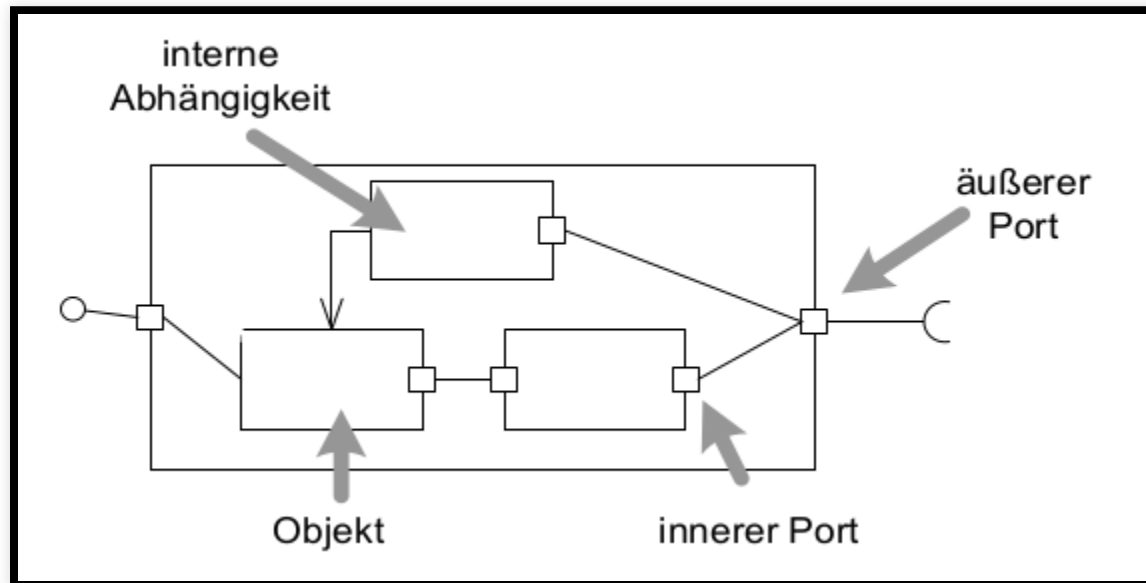
- Objektdiagramm
- Kompositionsstrukturdiagramm
- Sequenzdiagramm
- Laufzeitkontextdiagramm
- Kommunikationsdiagramm
- Interaktionsdiagramm

## Laufzeitsicht in UML



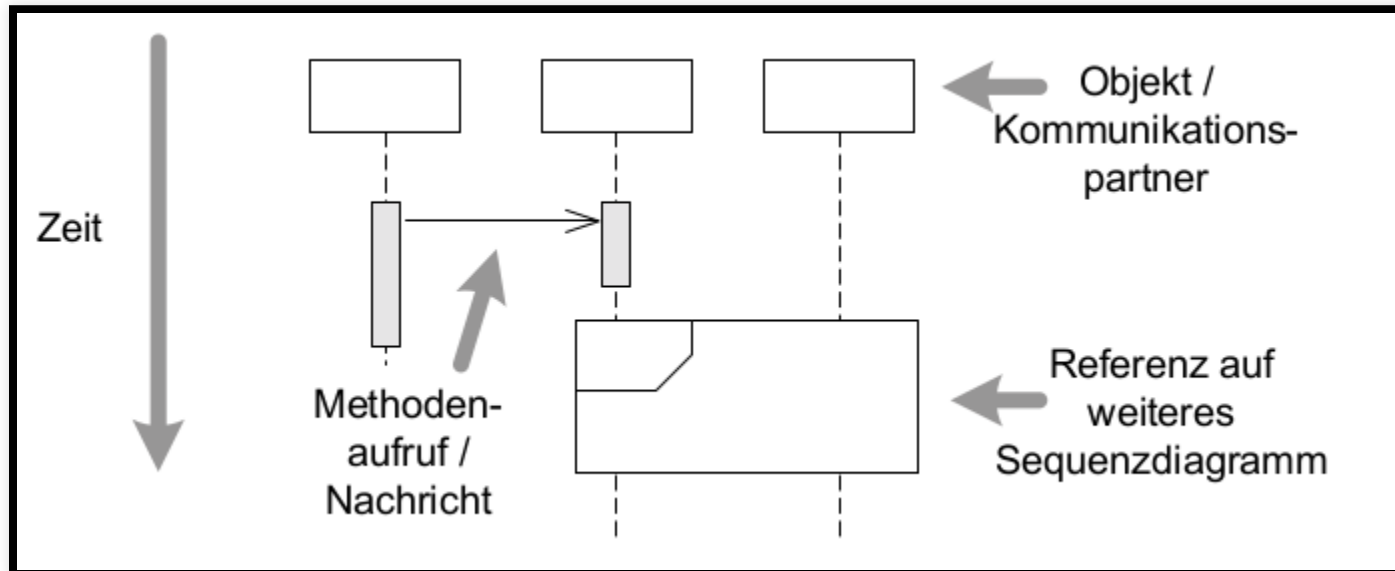
Objektdiagramm

## Laufzeitsicht in UML



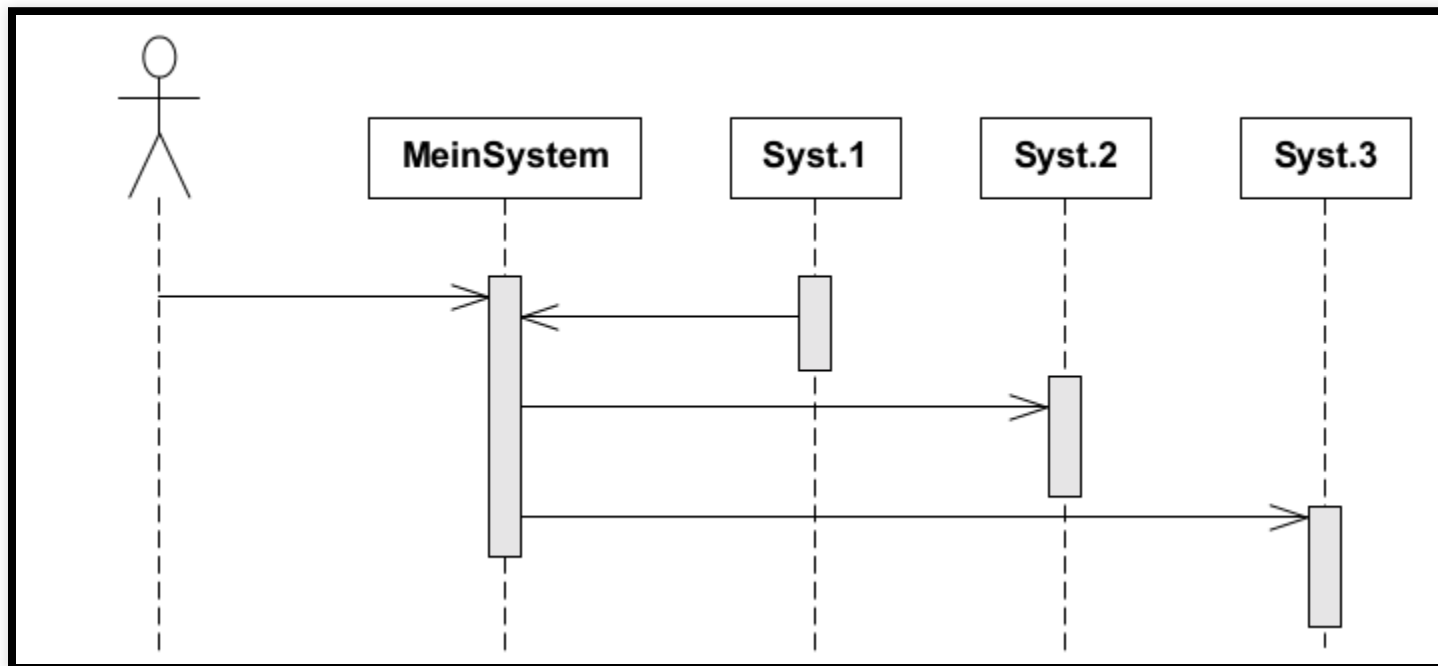
*Kompositionsstrukturdiagramm*

## Laufzeitsicht in UML



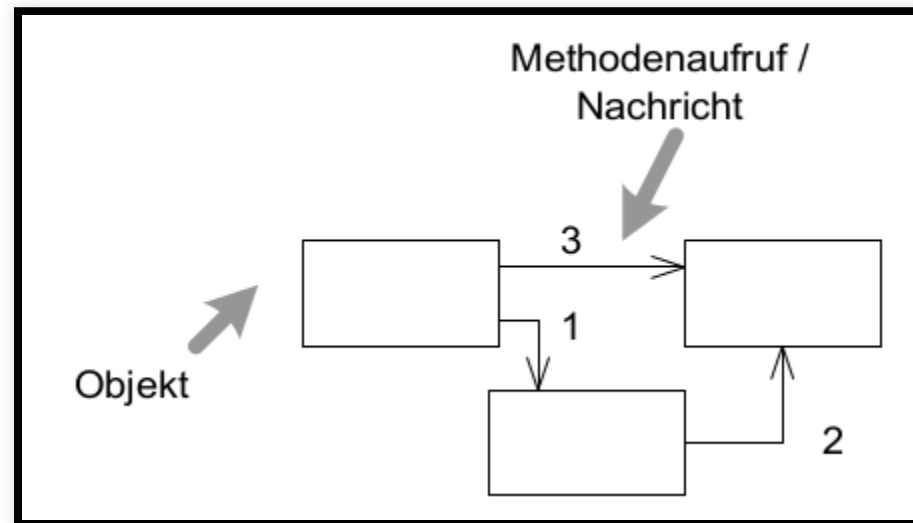
*Sequenzdiagramm*

## Laufzeitsicht in UML



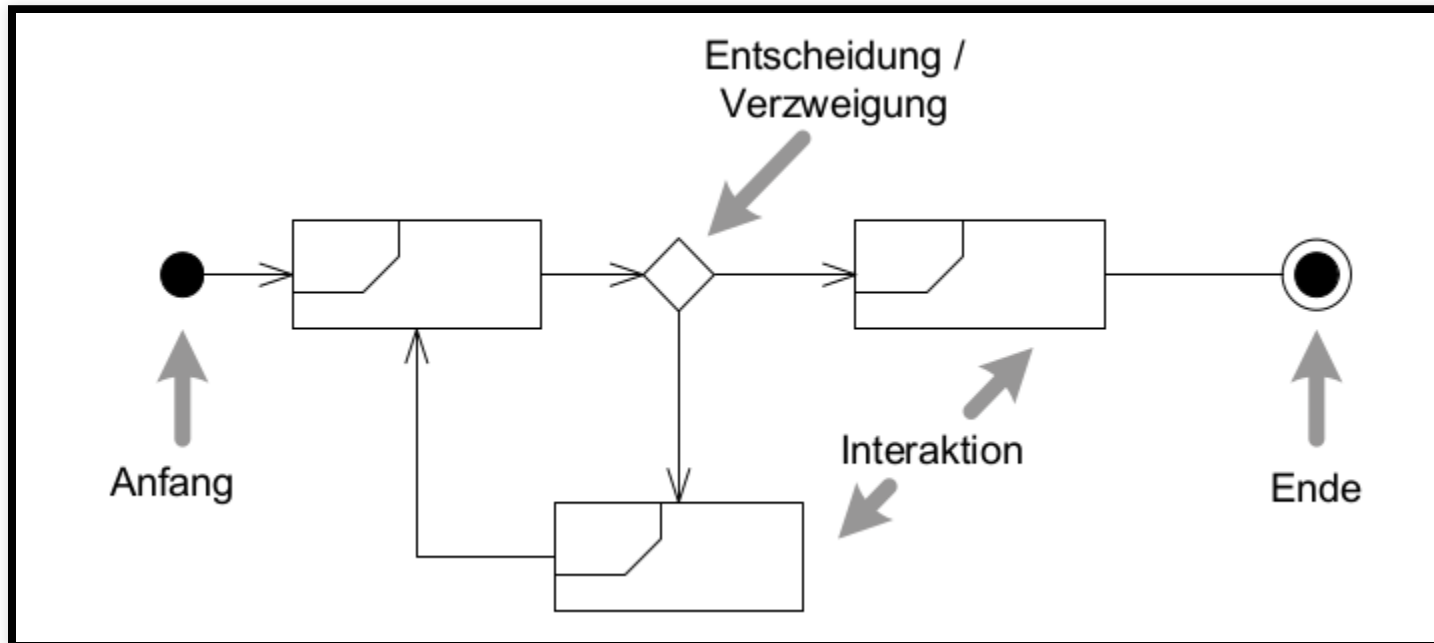
*Laufzeitkontextdiagramm*

## Laufzeitsicht in UML



*Kommunikationsdiagramm*

## Laufzeitsicht in UML



*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?

UML für Softwarearchitekten

Klausurrelevanz?

Was ist Softwarearchitektur?

Geschichte und Trends

Sichten auf Architekturen

Qualität und andere nichtfunktionale Anforderungen

Architekturmuster

Dokumentation von Architekturen

Technologien und Frameworks

Was ist Qualität?

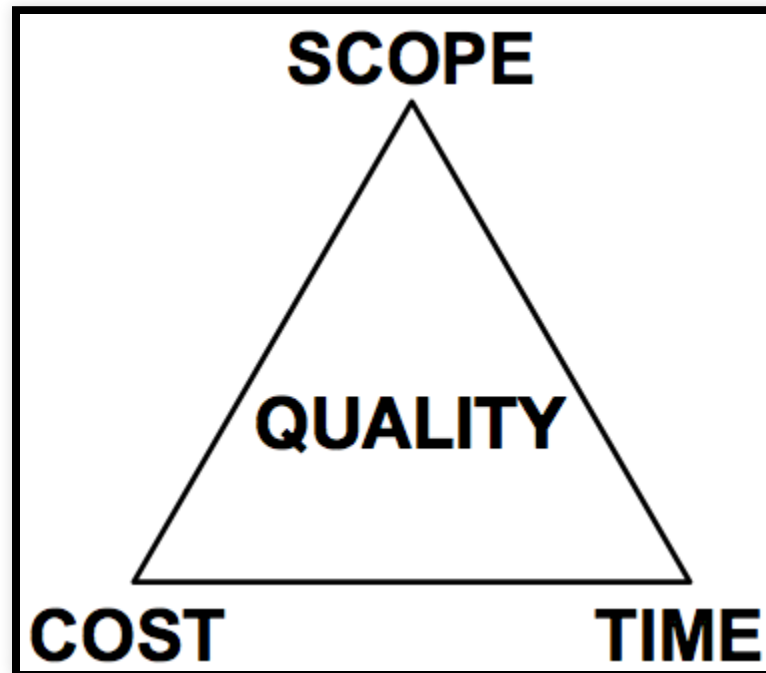
*Was ist Qualität?*

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

# Beispiel funktionale Anforderung „Sortierung von Daten“

- Kann funktional erfüllt sein, aber nichtfunktional?
- Sortierung großer Datenmengen (Terabyte), die nicht mehr zeitgleich im Hauptspeicher gehalten werden können
- Sortierung robust gegenüber unterschiedlichen Sortierkriterien (Umlaute, akzentuierte Zeichen, Phoneme, Ähnlichkeitsmaße und anderes)
- Sortierung für viele parallele Benutzer
- Sortierung unterbrechbar für lang laufende Sortiervorgänge
- Erweiterbarkeit um weitere Algorithmen, beispielsweise für ressourcenintensive Vergleichsoperationen
- Entwickelbarkeit im räumlich verteilten Team

Beispiel funktionale Anforderung „Sortierung von Daten“

<http://www.sorting-algorithms.com/>

Qualitätsmerkmale nach DIN/ISO 9126

Funktionalität

Zuverlässigkeit

Benutzbarkeit

Effizienz

Änderbarkeit

Übertragbarkeit

Funktionalität

Existenz eines Satzes von Funktionen mit spezifizierten  
Eigenschaften

Zuverlässigkeit

Fähigkeit, Leistungsniveau über einen Zeitraum  
aufrecht zu erhalten

Benutzbarkeit

Aufwand zur Benutzung und individuelle Beurteilung  
der Benutzung



Effizienz

Verhältnis Leistungsniveau / eingesetzte  
Betriebsmittel

Änderbarkeit

Aufwand zur Durchführung von Änderungen

Übertragbarkeit

Eignung zur Übertragung in andere Umgebung

Was ist Softwarearchitektur?

Geschichte und Trends

Sichten auf Architekturen

Qualität und andere nichtfunktionale Anforderungen

**Architekturmuster**

Dokumentation von Architekturen

Technologien und Frameworks

Was sind Architekturmuster?

A pattern for software architecture 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, their relationships, and the ways in which they collaborate.

*(1996 / Pattern Oriented Software Architecture)*

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.

Haben Sie jemals einen dummen Fehler zweimal begangen?

– *Willkommen in der realen Welt.*

Haben Sie diesen Fehler hundertmal hintereinander gemacht?

-*Willkommen in der Software-Entwicklung.*

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.

# Architektur: Von der Idee zur Struktur

Ein klassischer und systematischer Ansatz der Beherrschung von Komplexität lautet „teile und herrsche“ (divide et impera). Das Problem wird in immer kleinere Teile zerlegt, bis diese Teilprobleme eine überschaubare Größe annehmen.

# 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

## Dokumentation von Entscheidungen

- Warum wurde eine Entscheidung so getroffen?
- Welche Alternativen wurden bewertet?
- Andere Projektbeteiligte werden diese Entscheidungen später kritisieren!

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?

Unterlagen: [ai2018.nils-loewe.de](http://ai2018.nils-loewe.de)