DIE ESSENZ

Stand: April 2020

ARC42 Architektur-Dokumentation

1. Einführung und Ziele
2. Randbedingungen
3. Kontextabgrenzung
4. Lösungsstrategie
5. Bausteinsicht
6. Laufzeitsicht
7. Verteilungssicht
8. Konzepte
9. Entwurfsentscheidungen
10. Qualitätsszenarien
11. Risiken & techn. Schulden
12. Glossar

Whitebox-Template
1. Name
2. Überblick (Diagramm!)
3. Begründung
4. Enthaltene Blackboxes
5. Interne Schnittstellen
6. Offene Punkte

Blackbox-Template
1. Name
2. Zweck / Verantwortlichkeit
3. Schnittstellen
4. Ablageort / Datei
5. Erfüllte Anforderungen
6. Variabilität
7. Offene Punkte

Struktur von Entwurfsentscheidungen
1. Was muss entschieden werden (Fragestellung)?
   1.1 In welchem Kontext?
2. Entscheidungskriterien
3. Mögliche Alternativen
4. Wie wurde entschieden?
   4.1 Warum?
   4.2 Getroffene Annahmen
   4.3 Verworfene Alternativen
5. Konsequenzen?
6. Bekannte Risiken?
7. Wer hat wann entschieden?

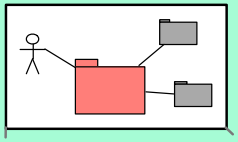
Szenarien zur Definition von Qualitätsanforderungen
1. Qualitätsbaum
2. Anwendungsszenarien/ Änderungsszenarien

Struktur von Konzepten
1. Ziele und Anforderungen
2. Randbedingungen
3. Scope / Kontext
4. Lösung / Vorgehen
   4.a Strukturen & Abläufe
   4.b Beispiele inkl. Code
5. Betrachtete Alternativen
6. Risiken

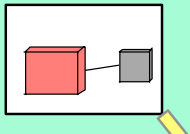
...
...
...
Logging
Sicherheit
Transaktionen
Sessions
Typische Muster & Abläufe

Kontextabgrenzung

fachlich



technisch



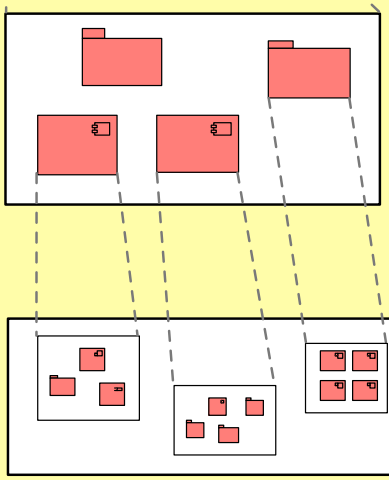
Qualitäts-ziele

Ziel	Erklärung
...	...

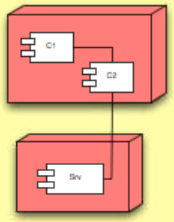
Stakeholder-tabelle

Wer?	Interesse?
...	...

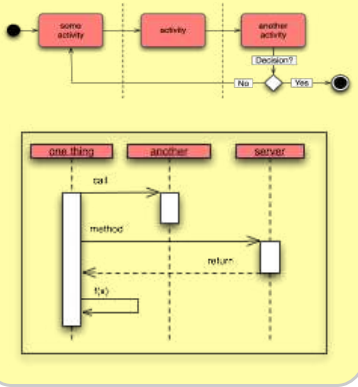
Bausteinsicht



Verteilungssicht



Laufzeitsicht



arc42 V 7.0 (Feb. 2017)
BigPicture
created by Peter Hruschka
& Gernot Starke

Definition Softwarearchitektur: IEEE-1471

(eine von ca. vielen Definitionen)

„Die grundsätzliche Organisation eines Systems, wie sie sich in dessen Komponenten, deren Beziehung zueinander und zur Umgebung widerspiegelt, sowie die Prinzipien, die für seinen Entwurf und seine Evolution gelten.“

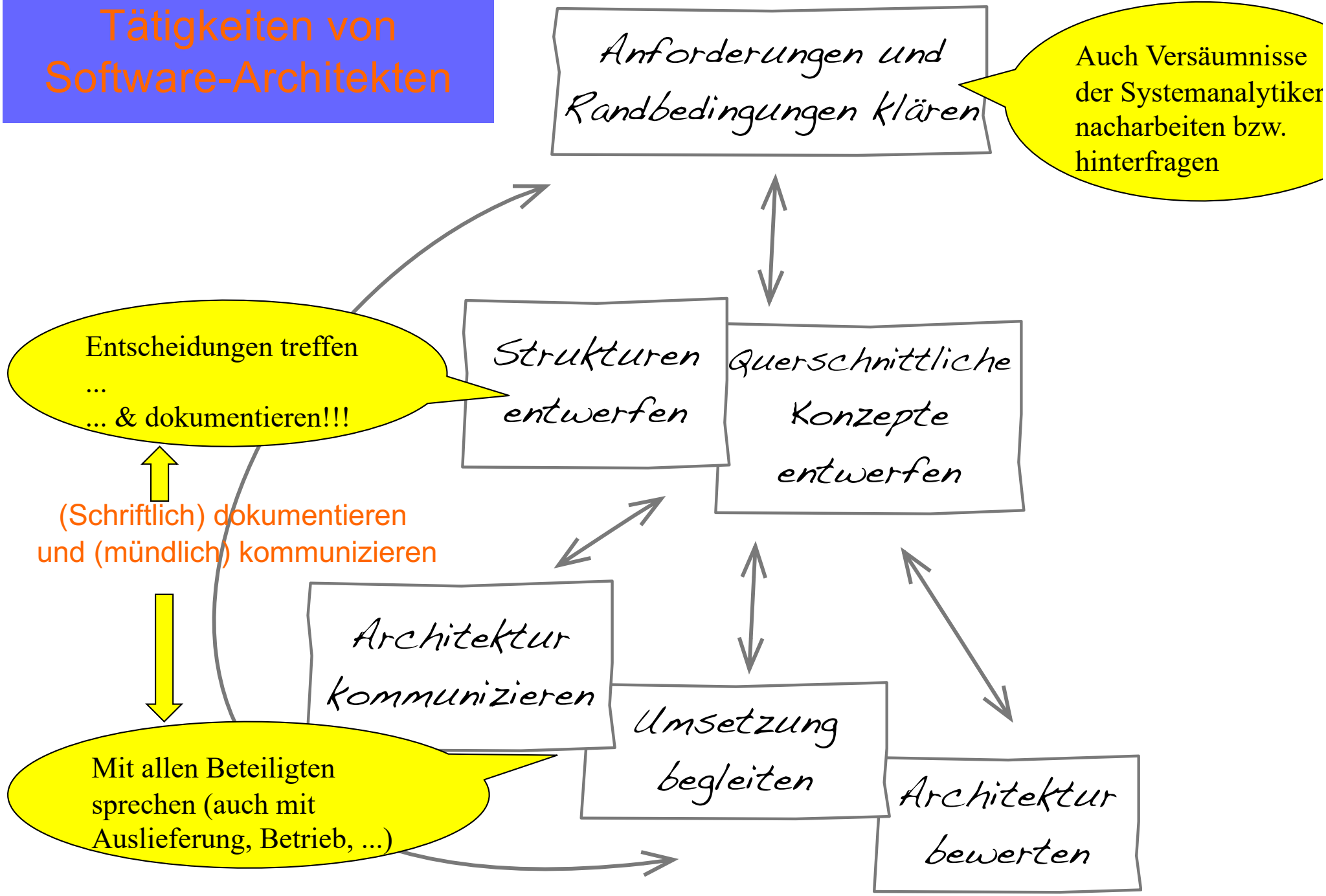
Abhängig-keiten, Schnittstellen

Bausteine:
Klassen,
Komponenten
Pakete,
Subsysteme,
Layer, ...

Konzepte, Regeln

Entwurfs-entscheidungen

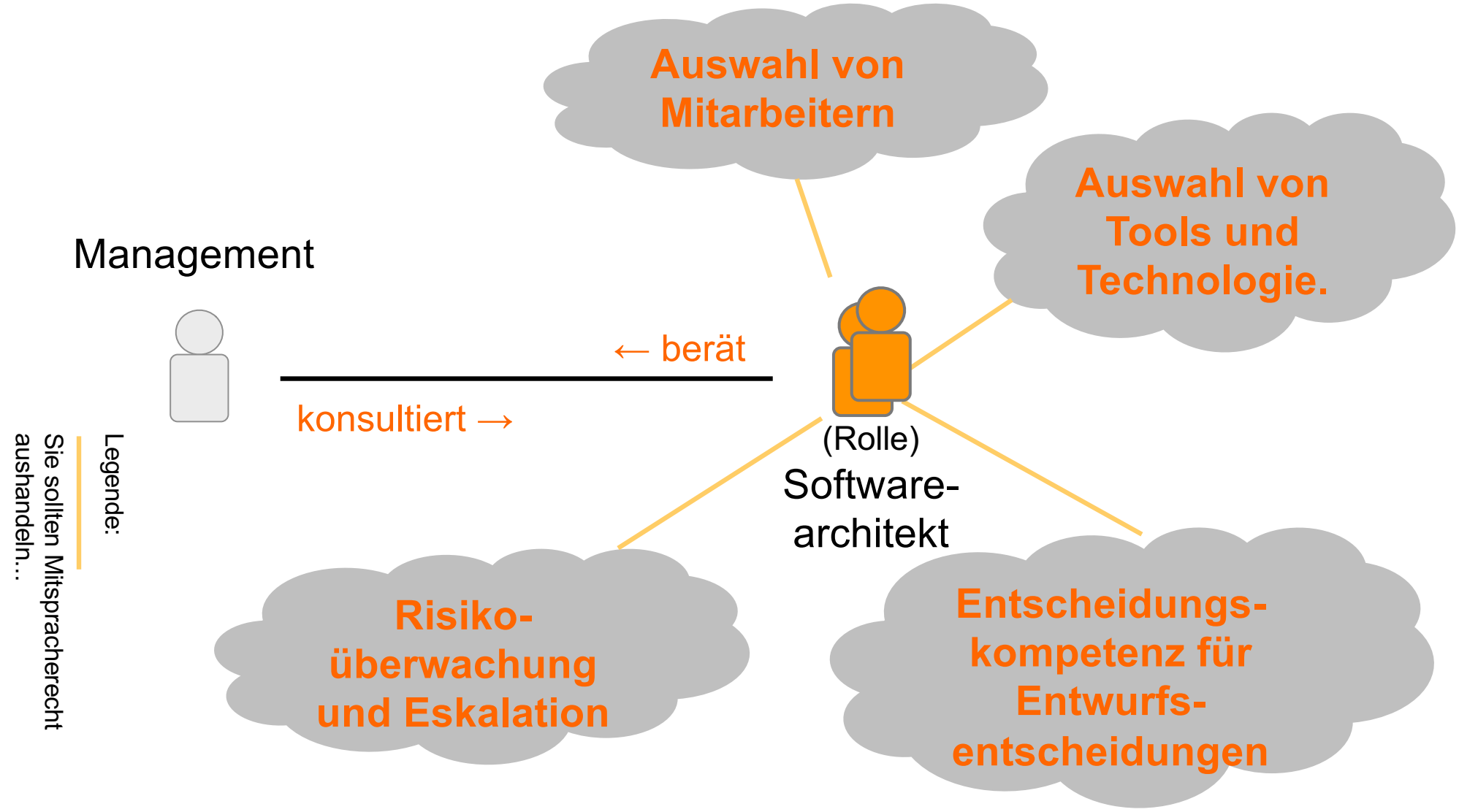
Tätigkeiten von Software-Architekten



Vorteile iterativ/inkrementellen Entwurfs

- Frühzeitige Rückmeldung:
  - **frühes Erkennen von Risiken**
  - **verlängert Zeit zur Abhilfe**
- Ermöglicht (frühzeitige) Reaktion auf Änderungen
- Ermöglicht frühe Lieferbarkeit von Kernfunktionalität

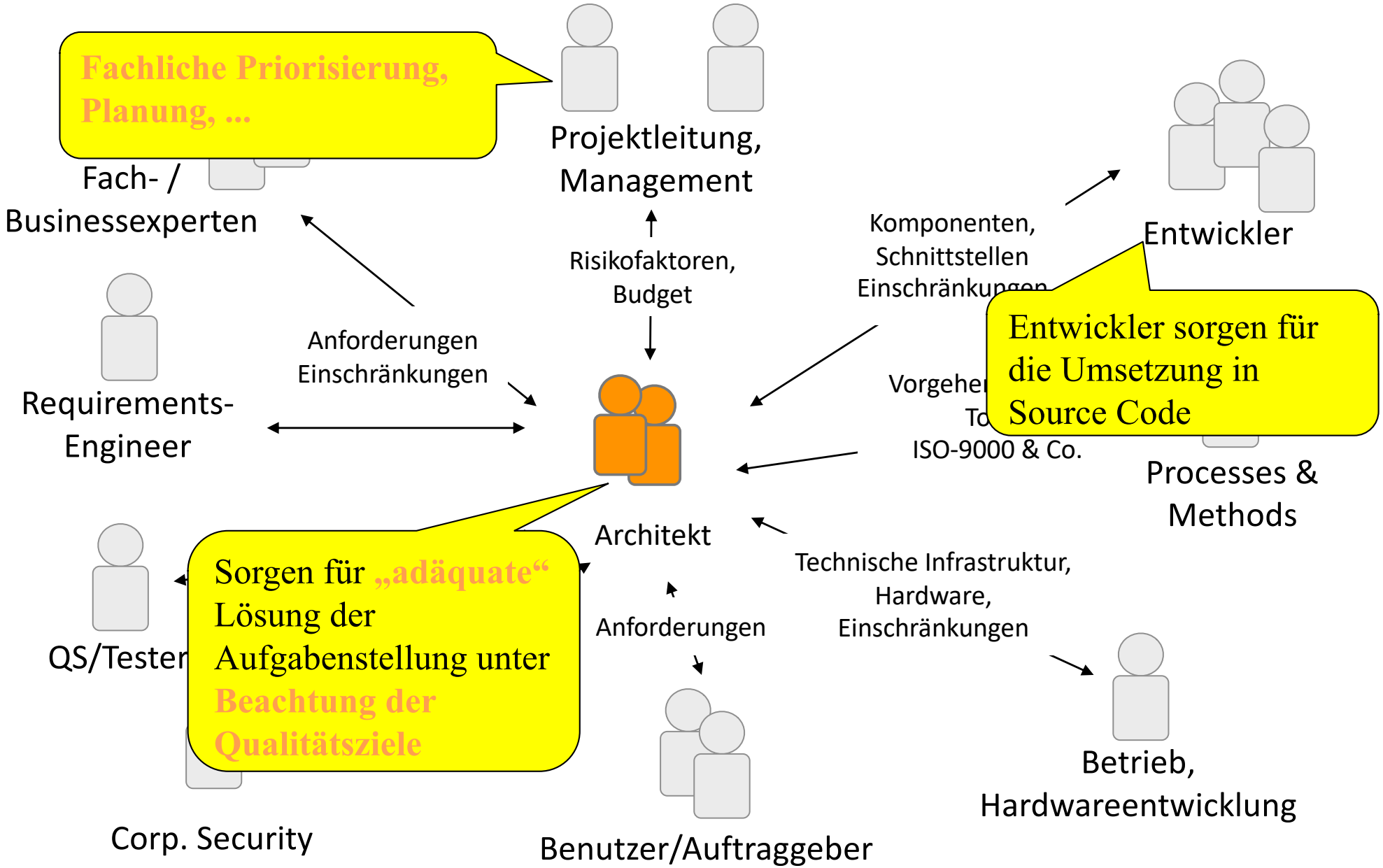
Management und Architekten



Management und Architektur ff. (**Conways Law**...)

- Team- oder Organisationsstruktur ist (oft) kongruent zur Architektur

Architekt als Multilinguist und Dolmetscher



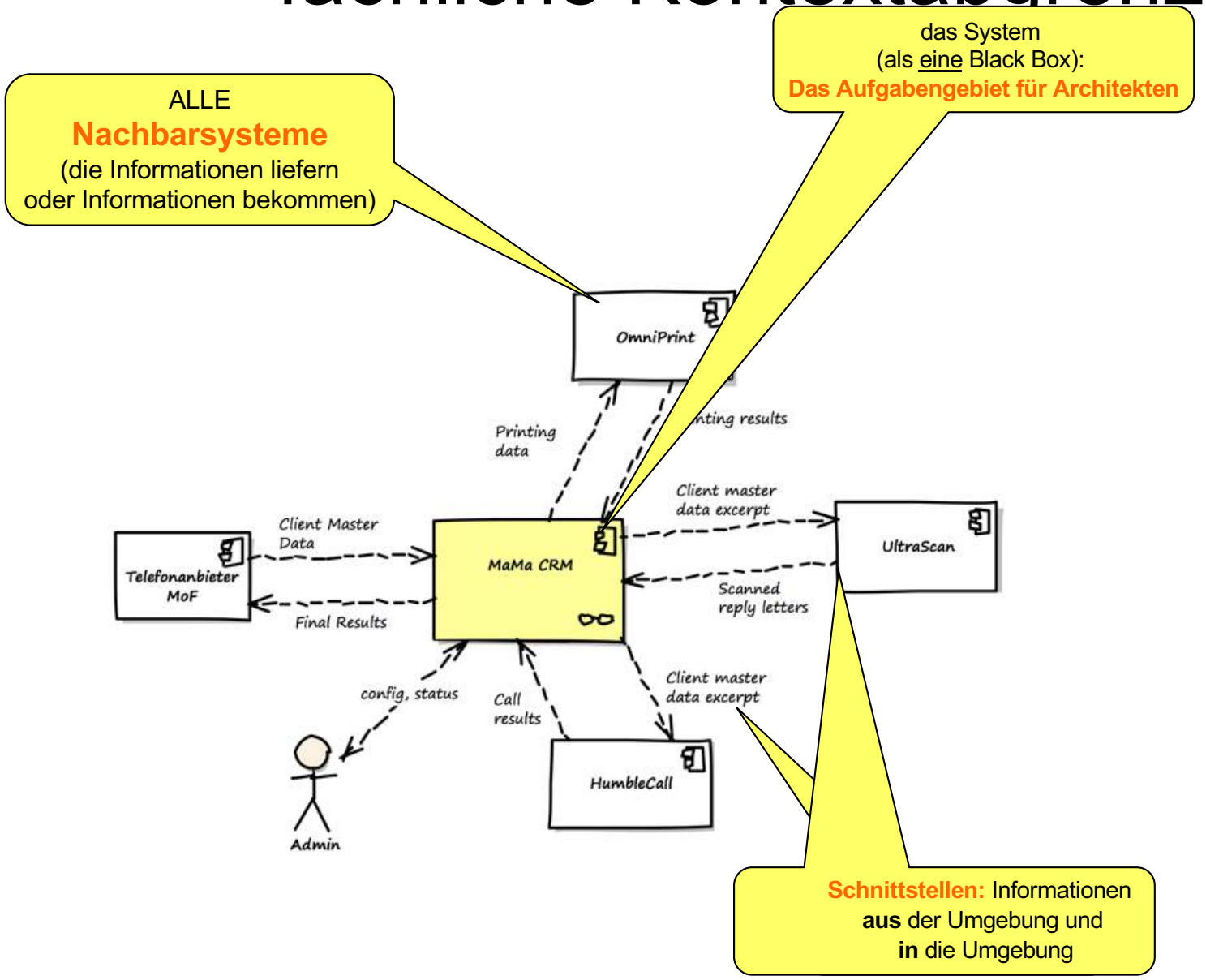
Warum Architekturtemplates?

- EINHEITLICHER Dokumentationsaufbau
  - Leichte Auffindbarkeit
  - Wiederverwendbarkeit von Teilen

- Vorgegebene Beschreibungsmuster ... um nichts zu vergessen

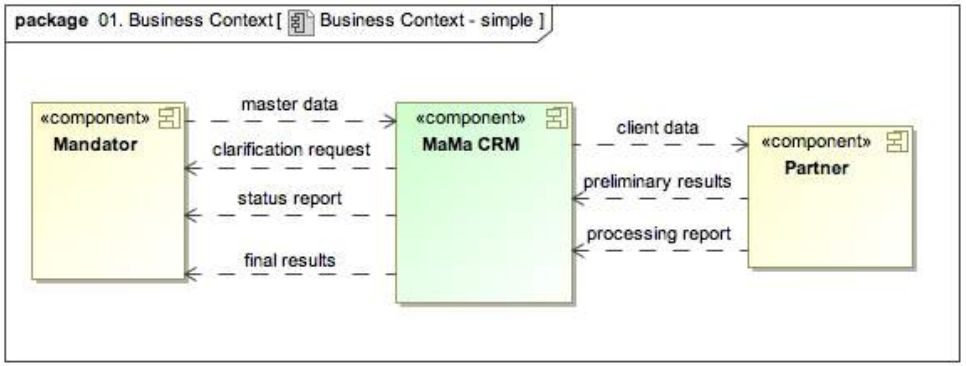
- Auch Templates für einzelne Bausteine:
  - Black-Box-Template (mit externen Schnittstellen)
  - White-Box-Template (interne Bausteinstrukturen)
  - .....

ARC42 (Sektion 3.1)
fachliche Kontextabgrenzung



Praxistipps Kontextabgrenzung

- Beginne Kontext am Tag-0!
  - Starte grob
  - Detailliere später

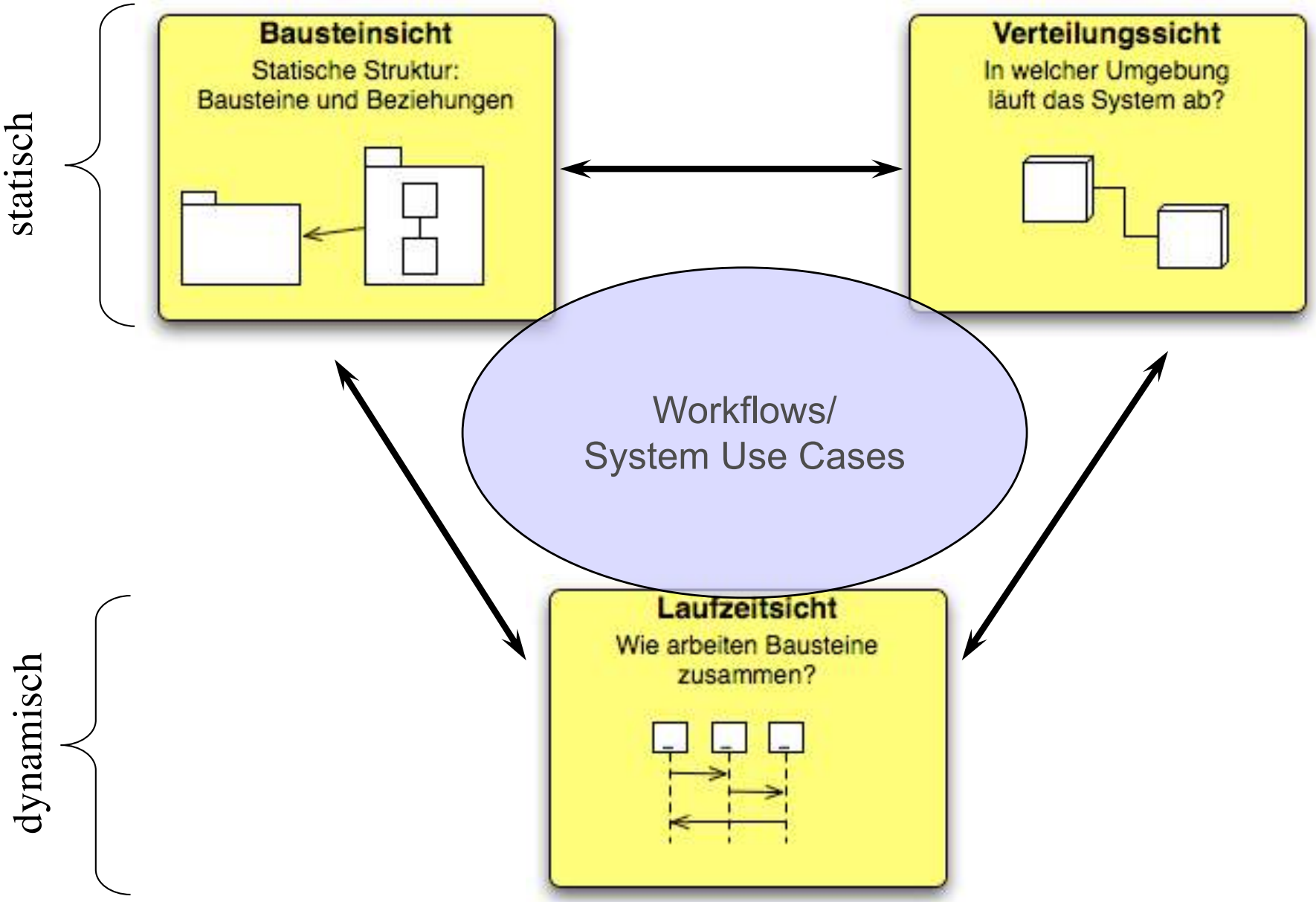


- Unterscheide fachlichen und technischen Kontext
- UML-relaxed: Datenflüsse statt Aufrufe/Abhängigkeiten
- Hole Feedback zum Kontext von **allen** Stakeholder

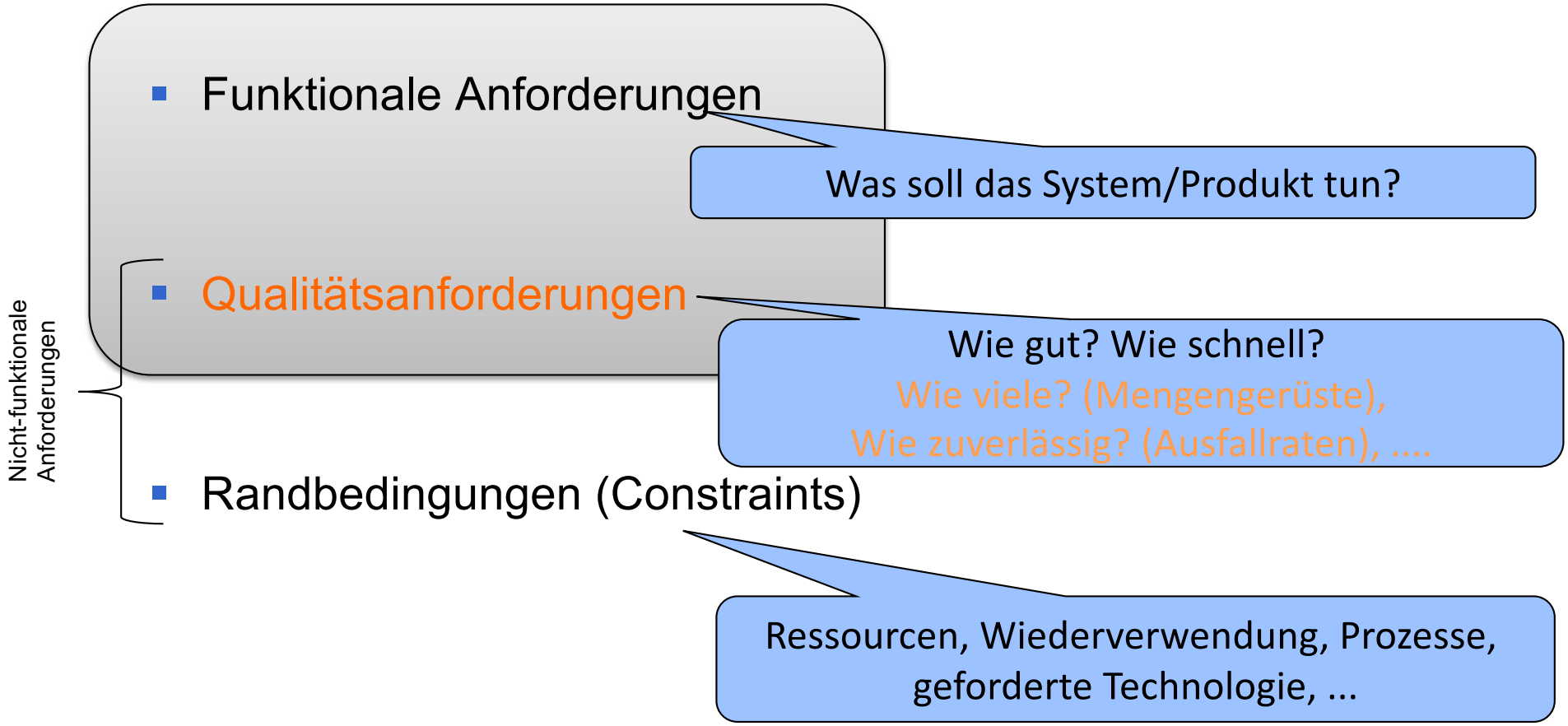
- Viele Nachbarn?
  Bilde (fachliche) Cluster
- **Volatile Nachbarn?**
  Explizit darauf hinweisen
- Bewerte die **Risiken aller externen Schnittstellen**
- Starte KEIN Projekt ohne Kontextabgrenzung.

Architekturen dokumentieren ...

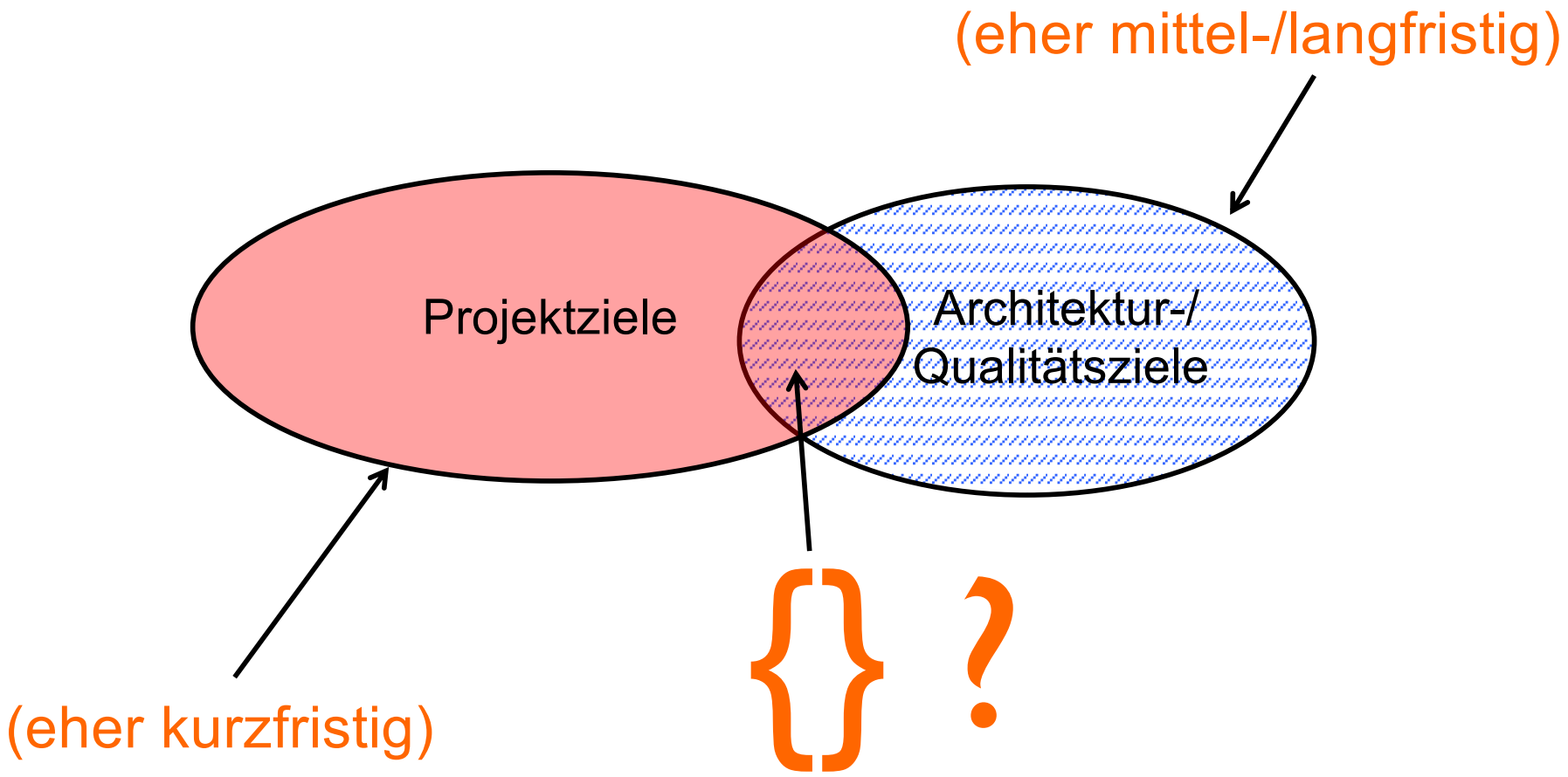
... mit verschiedenen Sichten



Architekturziele sind Ausschnitte aus den Qualitätsanforderungen

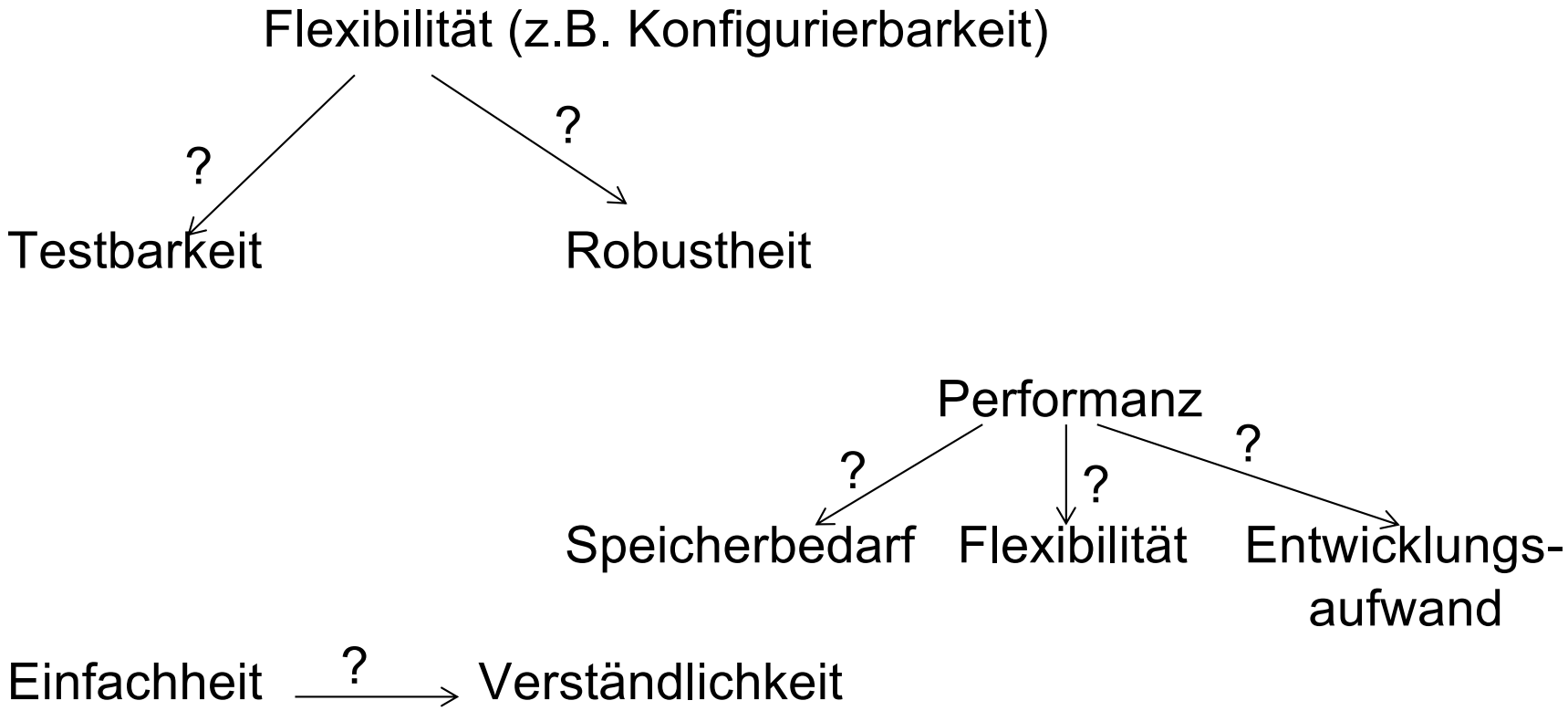


Projekt- und Architekturziele



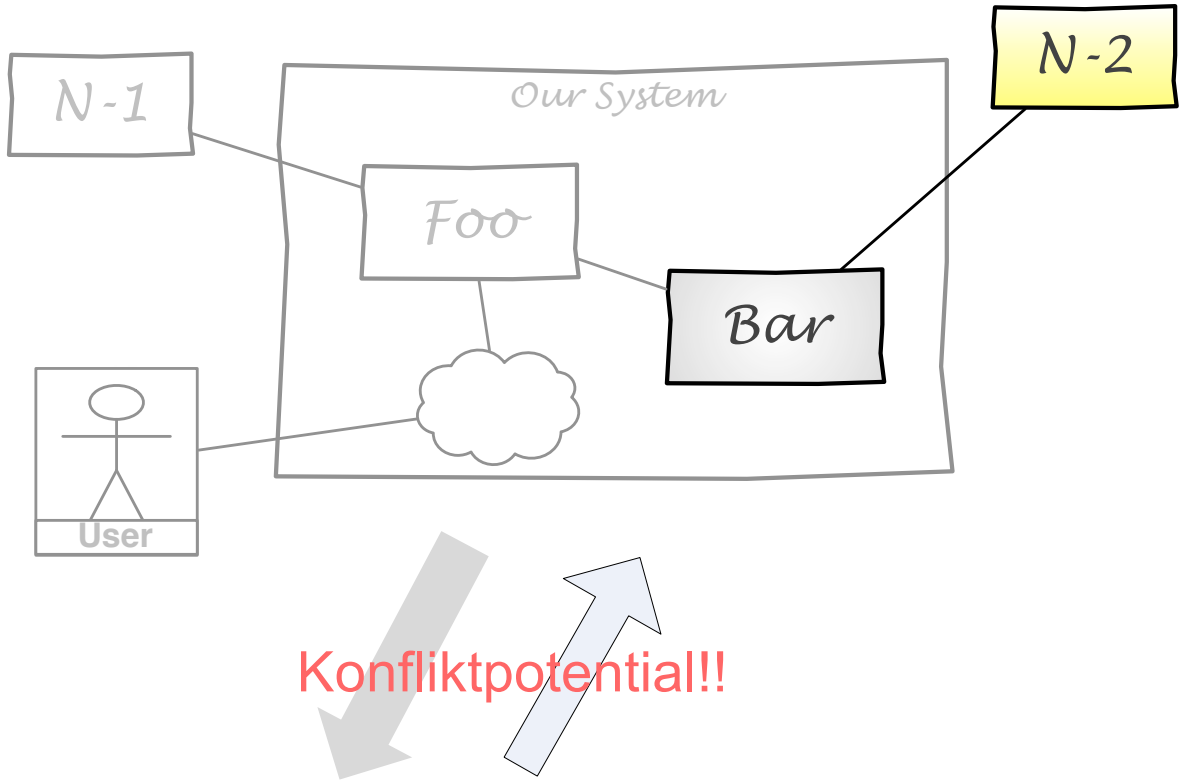
Beide müssen unter den maßgeblichen Beteiligten verhandelt und abgestimmt werden.

Abhängigkeiten von Qualitätsmerkmalen



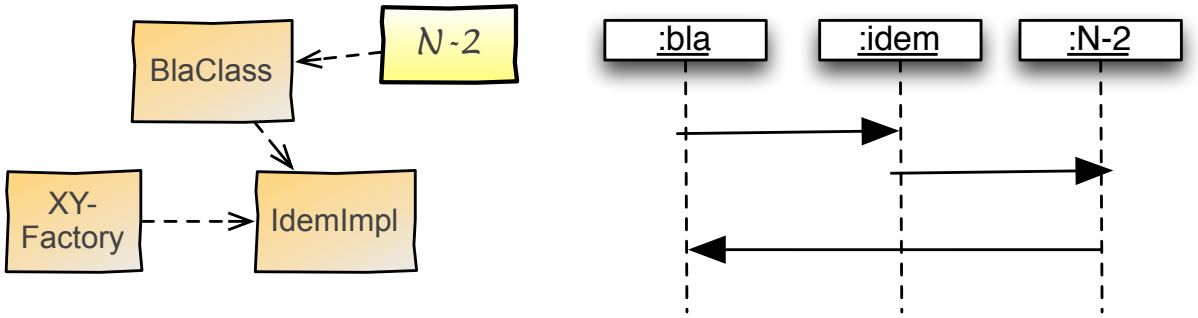
Top-Down...

- Grob nach fein
- Abstrakt nach konkret/detailliert



Bottom-Up...

- Fein nach grob
- Konkret / detailliert nach abstrakt



Schichten haben Vor- und Nachteile

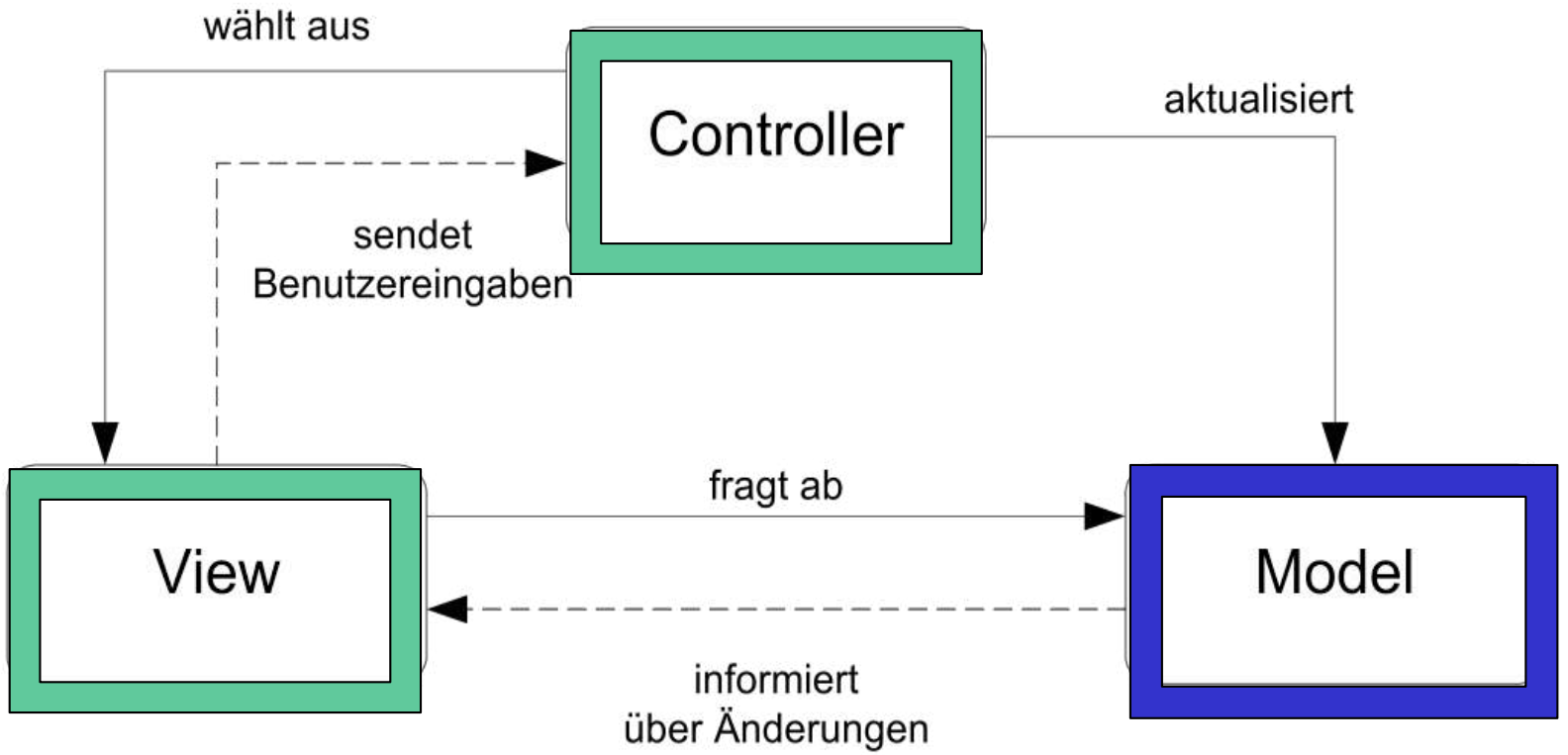
- Vorteile:
  - Geringe / keine Abhängigkeiten bei Erstellung und Betrieb
  - Flexibilität fördern:
    Implementierung einzelner Schichten austauschbar
  - leicht verständliches Strukturkonzept

- Nachteile:
  - Kann Performance negativ beeinflussen
  - Manche Änderungen sind sehr aufwändig
    - Beispiel: Persistentes Datenfeld hinzufügen: Änderung mindestens in GUI- und auch DB-Schicht.

Model-View-Controller, MVC

Ein Architekturmuster zur **Förderung der Flexibilität**

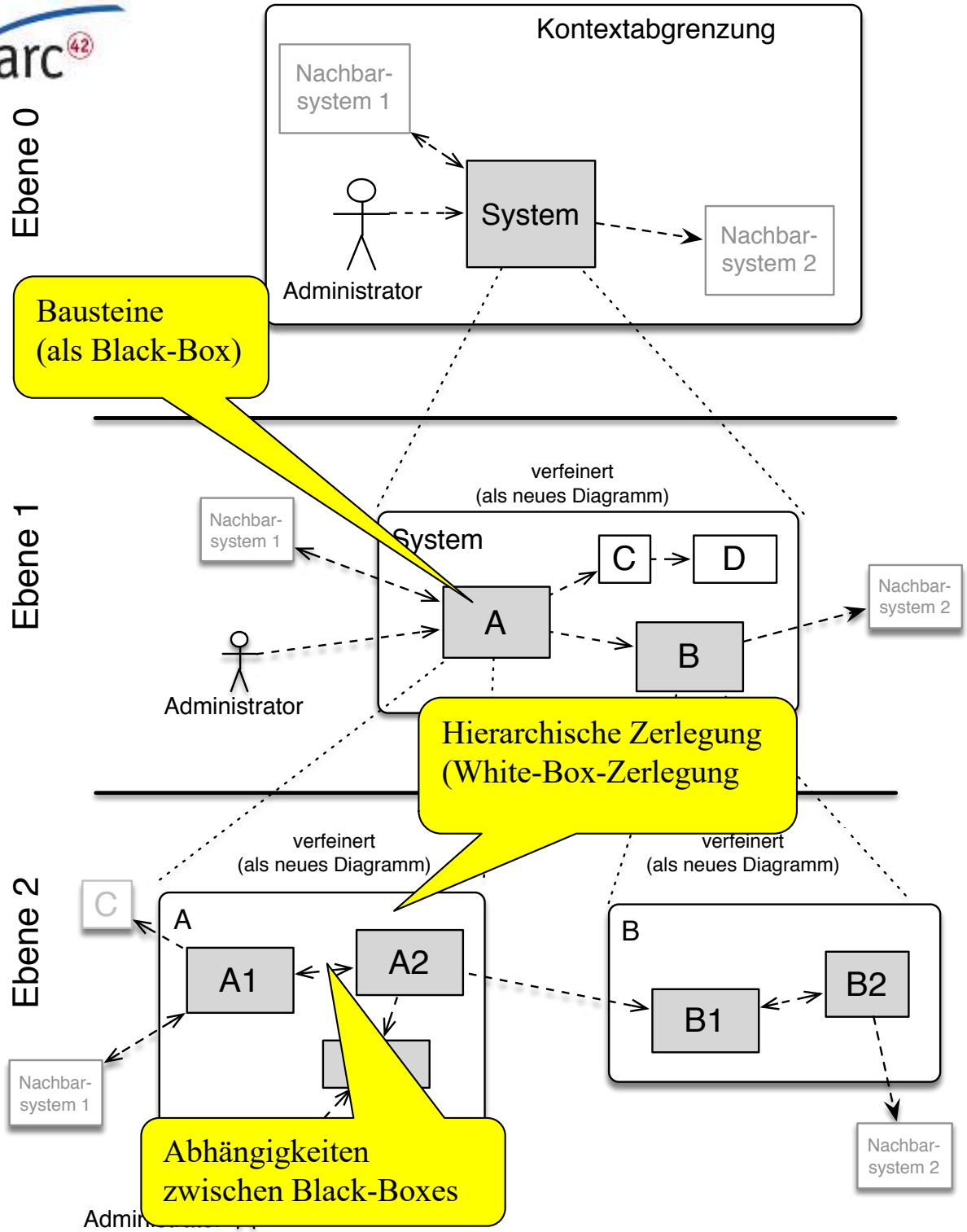
- **Model**: Daten, Funktionen / Zustand
- **View**: Darstellung der Daten des Modells
- **Controller**: Steuerung des Verhaltens der Anwendung



Aufgaben der Verteilungssicht:

- Modellierung/Dokumentation der Infrastruktur
- Mapping der Bausteine (bzw. der Artefakte, die aus Bausteinen entstehen) auf diese Infrastruktur (Deployment der Software auf Infrastruktur)
- Spezifikation von Knoten- und Kanteneigenschaften


Konfiguration
Buildingblock
Quellcode
Programm
Package
Modul
Funktion
Datei
Komponente
Block
Methode
Routine
Subsystem
File
Paket
Klasse
Skript
Executable
Library
Bausteine



Das Ziel:

Bausteine des Systems als Hierarchie darstellen:

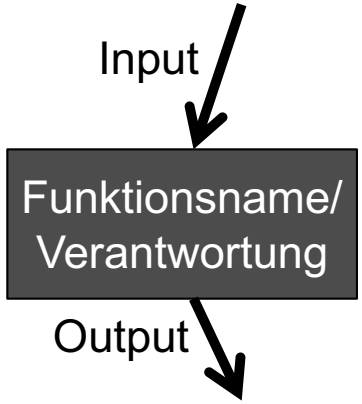
- vom Gesamtsystem als Black Box
- bis zu kleineren Bausteinen
  - in angemessenem Detailgrad
  - mit deren Abhängigkeiten

- Bausteine nur verfeinern, wenn mehr Details bekannt sein müssen

Black- und Whitebox

Blackbox:

- *Geschlossenes System mit (i.d.R definiertem) Ein-/Ausgabeverhalten ohne Beachtung des inneren Aufbaus*

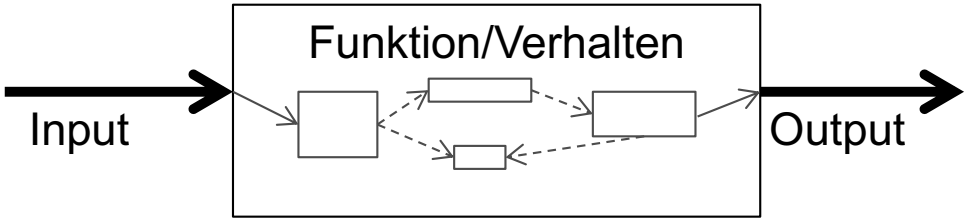


- Erfüllt:
  - Geheimnis-Prinzip (Information Hiding*))
    („WIE“ bleibt verborgen)
  - Kapselung des Innenlebens

*) David Parnas, 1972

Whitebox:

- *Geschlossenes System mit (i.d.R definiertem) Ein-/Ausgabeverhalten und bekanntem oder vorgegebenen innerem Aufbau*

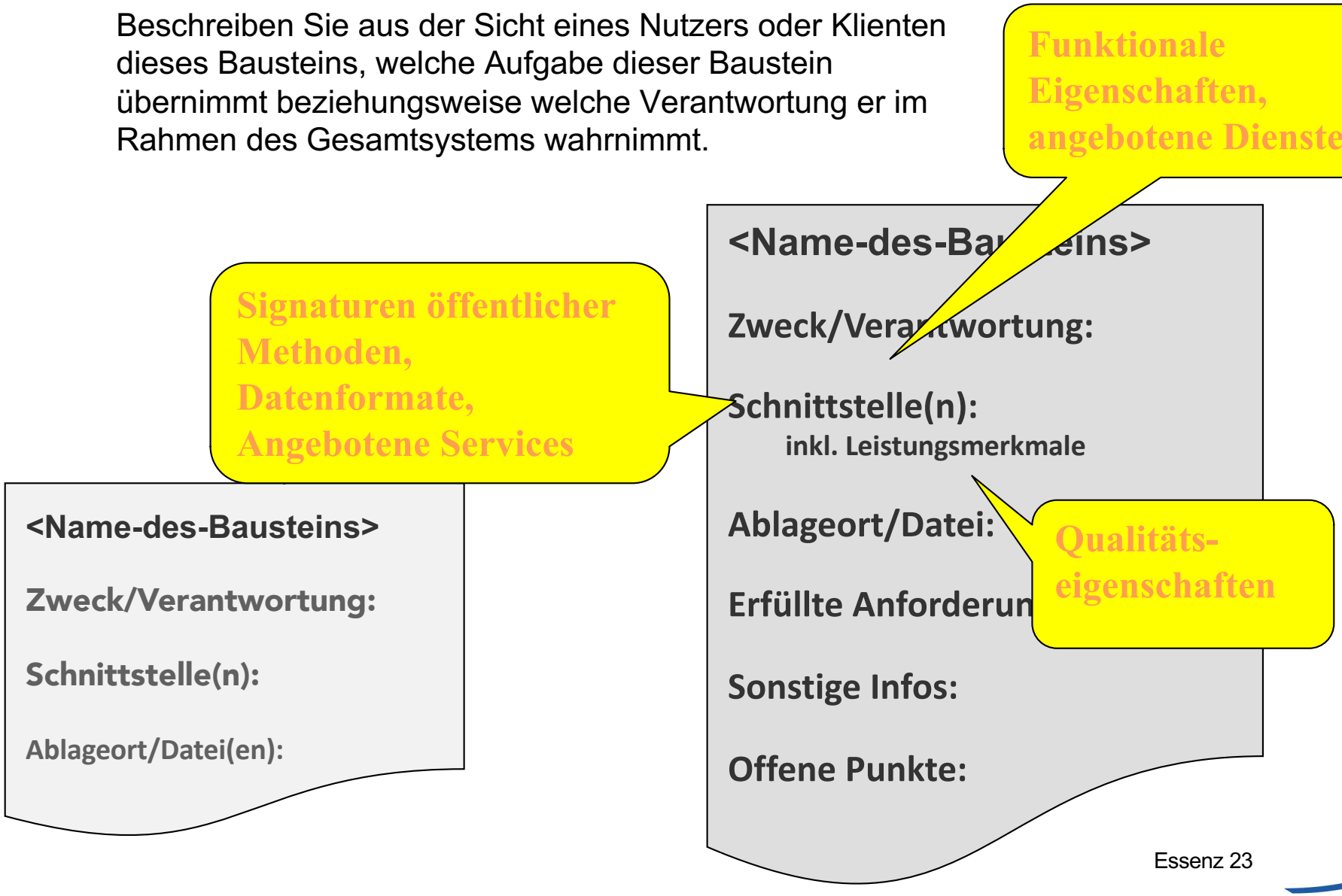


Blackbox-Template

Alles außer „Innereien“,
Keine interne Struktur

Zweck/Verantwortung

Beschreiben Sie aus der Sicht eines Nutzers oder Klienten dieses Bausteins, welche Aufgabe dieser Baustein übernimmt beziehungsweise welche Verantwortung er im Rahmen des Gesamtsystems wahrnimmt.



Was macht gute Schnittstellen aus?

- Einfach zu benutzen
  - Einfach zu erlernen (intuitiv)
- Vollständig aus Benutzersicht
  - Deckt alle funktionalen Anforderungen ab
  - Erfüllt Qualitätsanforderungen
- Angemessene Dokumentation
- Client-Code leicht zu verstehen
- Neue Versionen brechen keinen bestehenden (Client-) Code

(Ein) Ratschlag

Robustheitsprinzip
(aka Postel‘s Law):

„be **conservative** in what you do, be **liberal** in what you accept from others“

„sei streng bei dem was du tust und offen bei dem was du von anderen akzeptierst“

– RFC 761

a general social code for software: “be strict in your own behavior, but tolerant of harmless quirks in others."

Vorteile:

- Möglicherweise höhere Robustheit
- (Kleine) Fehler anderer Bausteine führen nicht zu Abbruch

Nachteile:

- Probleme werden u.U. von verschiedenen Bausteinen unterschiedlich behandelt
- Verletzt Separation of Concern und Kohäsion
- Verminderte Datenintegrität
- (Erheblicher) Mehraufwand

https://devopedia.org/postel-s-law

Whitebox-Template

<Name-des-Bausteins>
Übersicht:

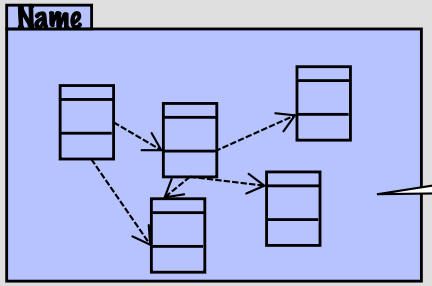


Diagramm mit feineren Bausteinen und deren Abhängigkeiten

Entscheidungen, Gründe & Alternativen

.....

.....

Motivation für Zerlegung

Blackbox-Beschreibung der lokalen Bausteine

Lokaler Baustein 1
Lokaler Baustein 2
.....
Lokaler Baustein n

„Innereien“ = Spezifikation enthaltenen Black-Boxes

(interessante) Beziehungen zwischen lokalen Bausteinen

.....

.....

Offene Punkte: .....

.....

... und deren Zusammenhang („benutzt“, „erbt von“, „instanziiert“)

Design Patterns

Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides, "Design Patterns", Addison-Wesley 1995.

- **5 Creational Patterns**: Abstract Factory, Builder, Factory Method, Prototype, Singleton
- **7 Structural Patterns**: Adapter, Bridge, Composite, Decorator, Façade, Flyweight, Proxy
- **11 Behavioral Patterns**: ChainOfResponsibility, Command, Interpreter, Iterator, Mediator, Memento, Observer, State, Strategy, Template Method, Visitor

Manche Patterns eignen sich nicht nur auf Klassenebene, sondern auch für größere Bausteine.

Patterns beschreiben immer ein Problem und schlagen dafür eine Lösung vor.

Bausteinsicht vs. Laufzeitsicht

+ Vollständige Modelle
  (Abstraktionen des gesamten Source Codes mit adäquatem Tiefgang)
+ Alle möglichen Services und alle Schnittstellen
- Zusammenhänge sind schwer zu sehen

Passende UML-Notation

- **Klassenmodelle, Paketdiagramme, Komponentendiagramme**

+ wesentliche Laufzeitaspekte werden hervorgehoben und exemplarisch diskutiert und analysiert
- beispielhafte Abläufe (Szenarien), nie alle Möglichkeiten

Weitere Notationen:

- Flussdiagramme, BPMN, EPKs
- Nummerierte Listen
- ...

Passende UML-Notation

Aktivitätsdiagramme (mit Swimlanes), **Sequenzdiagramme**, Kommunikationsdiagramme, Interaktionsübersichtsdiagramme, Timing Diagrams

Entwurfsprinzipien, die Wartbarkeit fördern

- **{schwache | lose | geringe | niedrige} Kopplung**
- **{starke | hohe} Kohäsion (innerer Zusammenhalt)**
- **Offenheit für Erweiterungen (Open/Closed Prinzip)**
- **Geheimnisprinzip (Kapselung)**
- Einfachheit (geringe Komplexität)
- Konsistenz („Standardisierung“)

Kopplung reduzieren

- durch
  - **Einführen von Interfaces** (-> Abhängigkeit „nur noch“ vom Interface, nicht mehr direkt von anderen Baustein);
    -> **Adapter**, **Fassaden**, ...
  - **Einführen von Factories** (Ein Baustein „lässt Objekte erzeugen“ und kennt daher nur noch die öffentlich Schnittstelle. Die Fabrik weiß einzig und allein über den internen Aufbau der erzeugten Objekte Bescheid)
  - ....

Zu starke Kopplung:

-> schwierige Wiederverwendbarkeit

-> hoher Aufwand bei Änderungen

Was sind „Konzepte“?

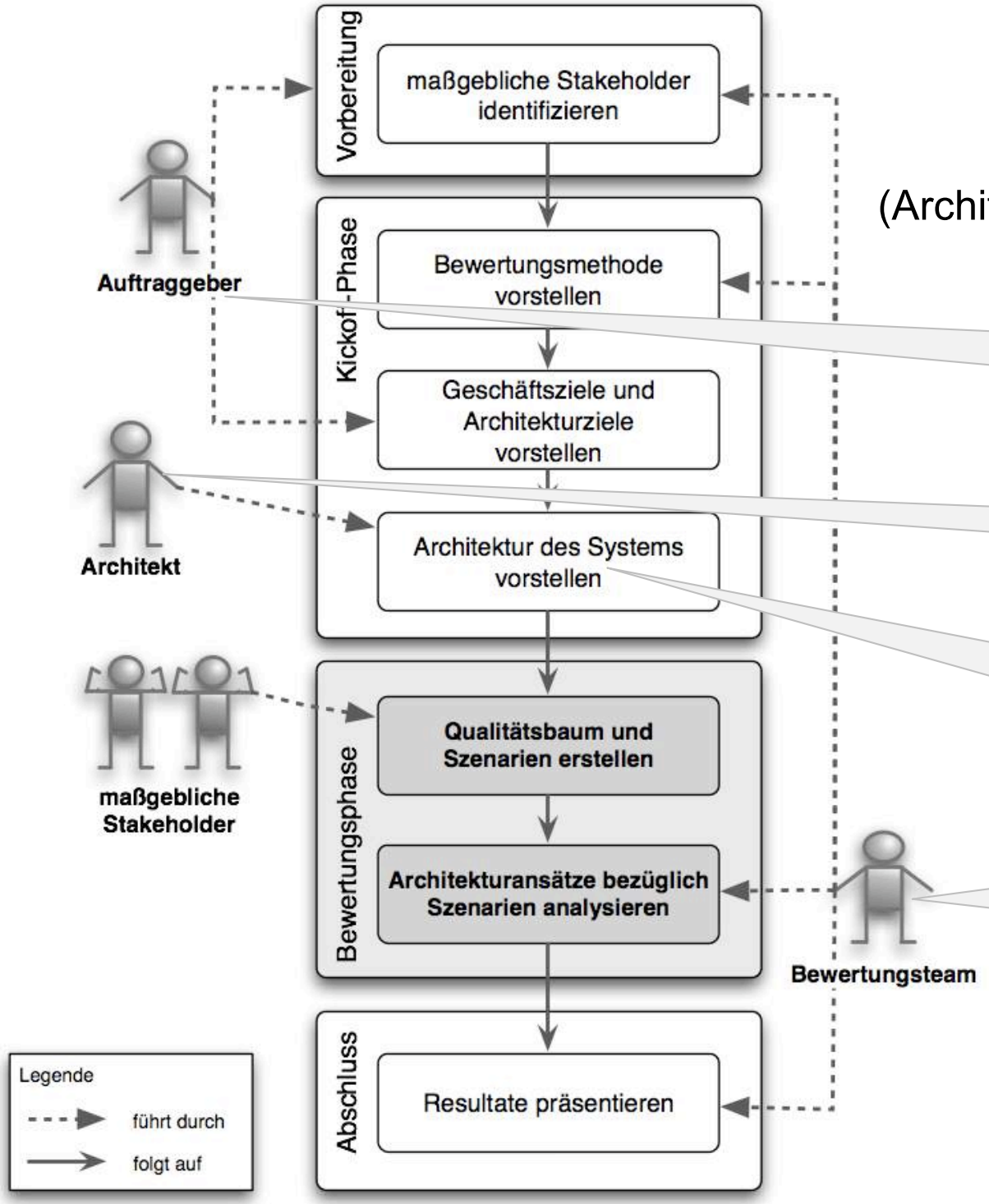
- **Übergreifende** Themen
- *Plan* zur Lösung eines (übergreifenden) Problems
  - Beispiele: Persistenz, UI, Logging, Security...
    - Auch: Build-Management, Modellierung & Generierung
  - Betrifft mehrere „Bausteine“
- Analog zu „aspektorientierter Programmierung“ (AOP) auch als „Aspekte“ bezeichnet

Anforderungen an Architekturdokumentation
(Qualitätsmerkmale einer Architekturdoku)

- **Verständlich**
- **Korrekt/richtig** und genau
  - lieber weniger, dafür definitiv richtig!
  - Aktuell
- **Angemessen, einfach** (statt umfänglich und vollständig)
- Wartbar/**effizient pflegbar**

Was braucht man zu ATAM-Bewertungen

(Architecture Tradeoff Analysis Method)



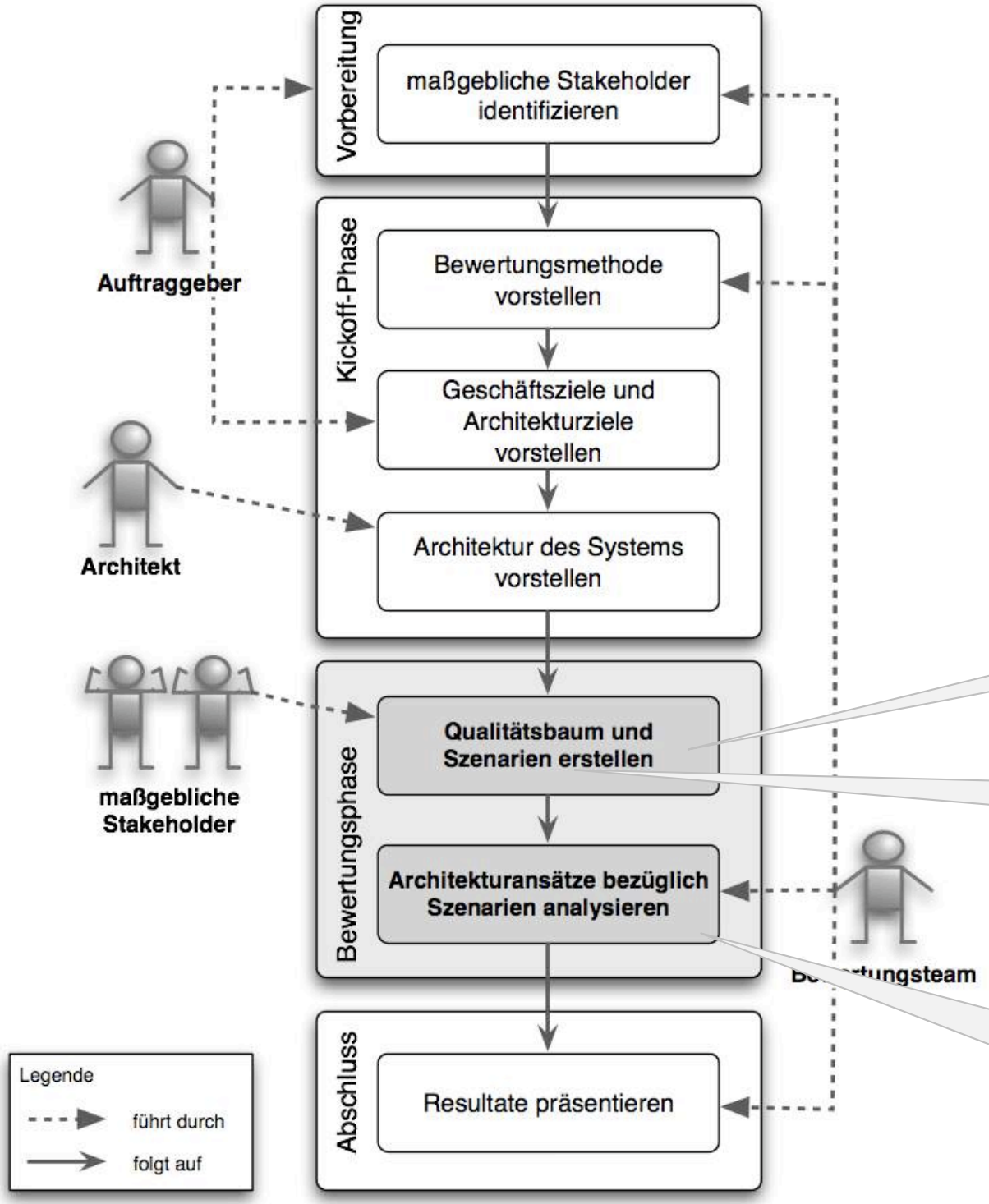
Fachlich Verantwortlichen

Architekten

Architektur-dokumentation

jemanden, der bewertet

ATAM-Ergebnisse



Qualitätsbaum

Bewertungsszenarien

Liste mit
Chancen und Risiken

Toolkategorien

- Modellierungswerkzeuge
  - Erlauben Dokumentation der Sichten
- Generierungswerkzeuge
  - Erzeugen Source Code
- Werkzeuge zur statischen und dynamischen Analyse
  - Finden **(einige)** Abhängigkeiten

- Anforderungsmangementwerkzeuge
- Dokumentationswerkzeuge
- Build-Systeme/-Werkzeuge
- Konfigurationsmanagement

(einige) Kriterien zur Werkzeugwahl

- **Einfachheit der Benutzung**
  - Erstellung von Doku
  - Finden / Lesen von Doku
  - Volltextsuche
- Verfügbarkeit im Team
- Integration mit Code-Repository
  - Merge-Fähigkeit der Inhalte
- Integration Text, Tabellen und Diagramme
- Robustheit / Ausfallsicherheit
- Rechte & Rollenkonzept
  - Zugriffsschutz

Nichts ist unmöglich!

- Automatisierbarkeit
  - Generierung benötigter Artefakte (Reportgenerator)
  - Integration in Daily-Build
  - Offene Datenformate
- UML-(Metamodel) Compliance
- Flexibilität der Modellierung
- Lizenz- und Betriebskosten
  - Lizenzmodell
- Multi-User-Fähigkeit
- Offline-Fähigkeit

Architekturdokumentation und Tools

- Egal, welches Tool Sie einsetzen:
  - Teile der Dokumentation können natürlich verantwortlich von Mitgliedern des Entwicklungsteams erstellt werden.
  - ABER: **Der Architekt verantwortet die Zusammenführung von Teilergebnissen zu einer Gesamtarchitektur**

„One Throat to Choke“

(Adrenalin Junkies & Formular Zombies, DeMarco et al.)