Architekturmethodik:

Systematisches Vorgehen beim Entwurf

Stand: April 2020

www.arc42.de

Ziele und Inhalt

Sie lernen

- Vorgehen bei der Architekturentwicklung
  - Quality-Driven
  - Domain-Driven
  - Top-Down, Bottom-Up
  - Patterns und Styles
- Entwurfsentscheidungen treffen

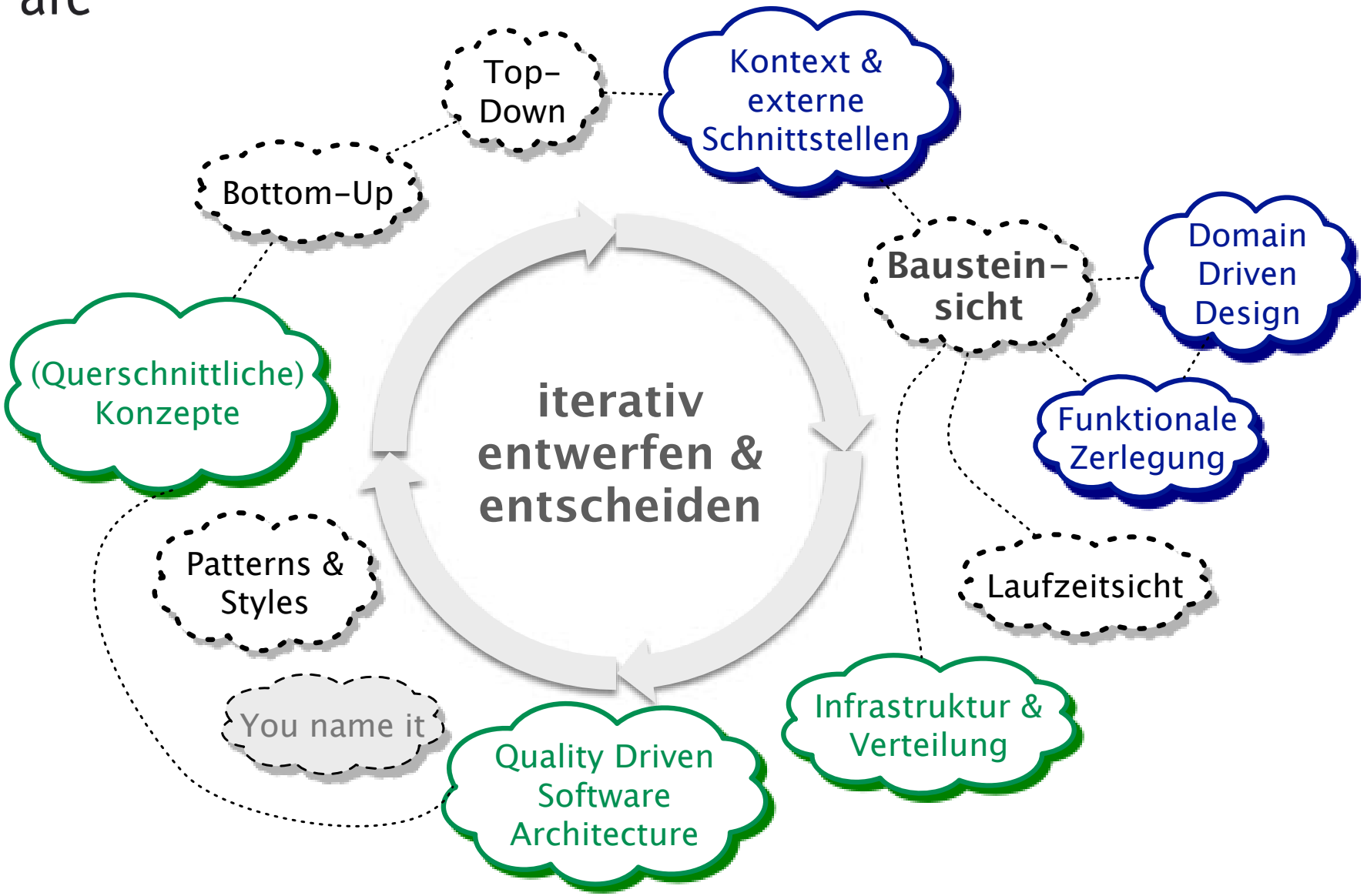
Sie üben:

- diverse Vorgehensweisen

Lernziele gemäß iSAQB CPSA-F:

- **LZ 2-1: Vorgehen und Heuristiken zur Architekturentwicklung auswählen und anwenden können (R1-R3)**
- **LZ 2-2: Softwarearchitekturen entwerfen (R1)**
- **LZ 2-5: Wichtige Architekturmuster beschreiben, erklären und angemessen anwenden (R1-R3)**
- **LZ 2-8: Qualitätsanforderungen mit passenden Ansätzen und Techniken erreichen (R1)**
- LZ 3-8: Architekturentscheidungen erläutern und dokumentieren (R2)
- **LZ 4-1: Qualitätsmodelle und Qualitätsmerkmale diskutieren (R1)**

Methodische Werkzeuge



Legende

technisch | fachlich | neutral

methodische Werkzeuge

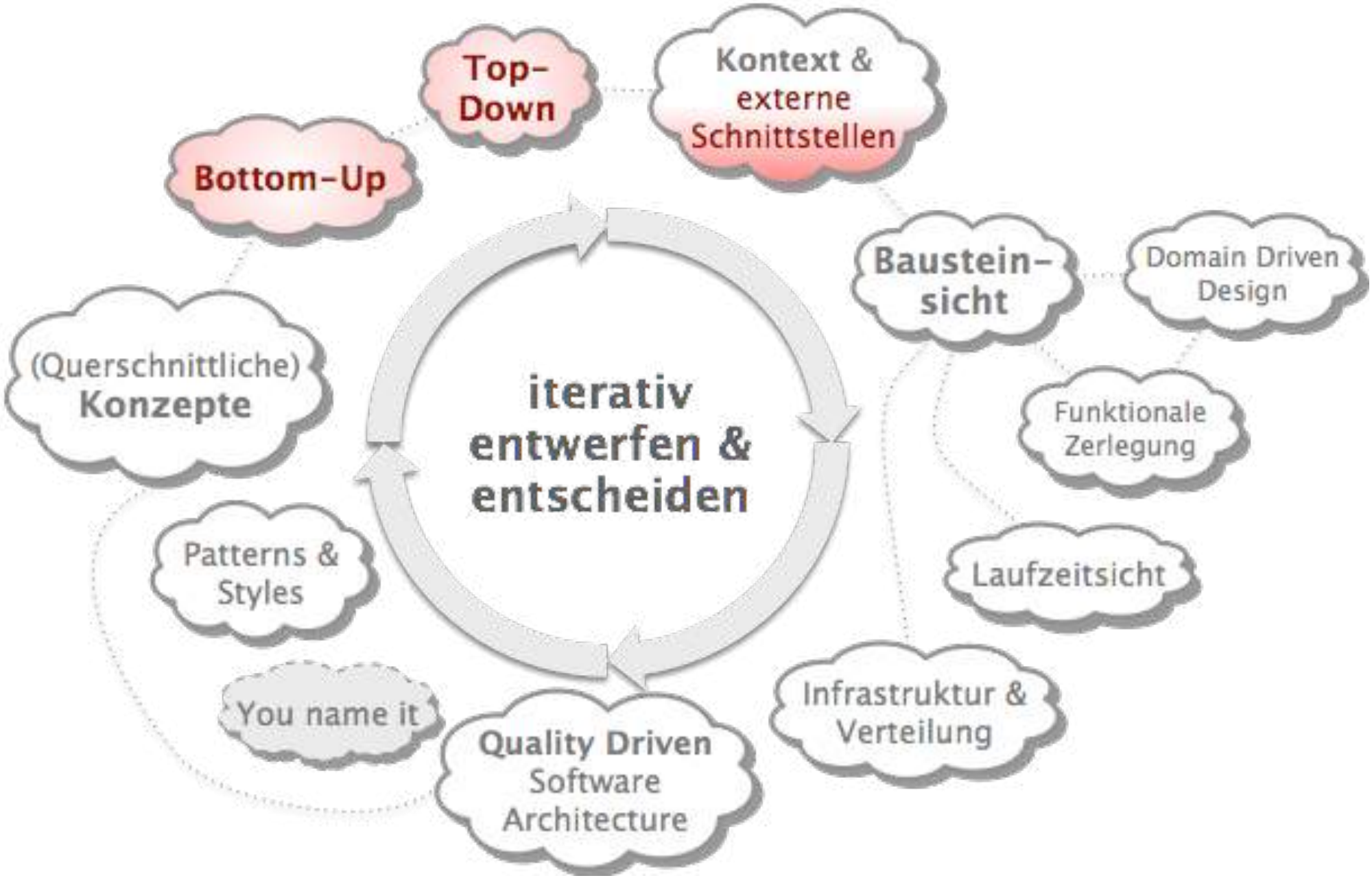
Lösungsstrategien
und erste Systemidee

Kap. 4

Vor „Details“ erst fundamentale Ansätze klären:

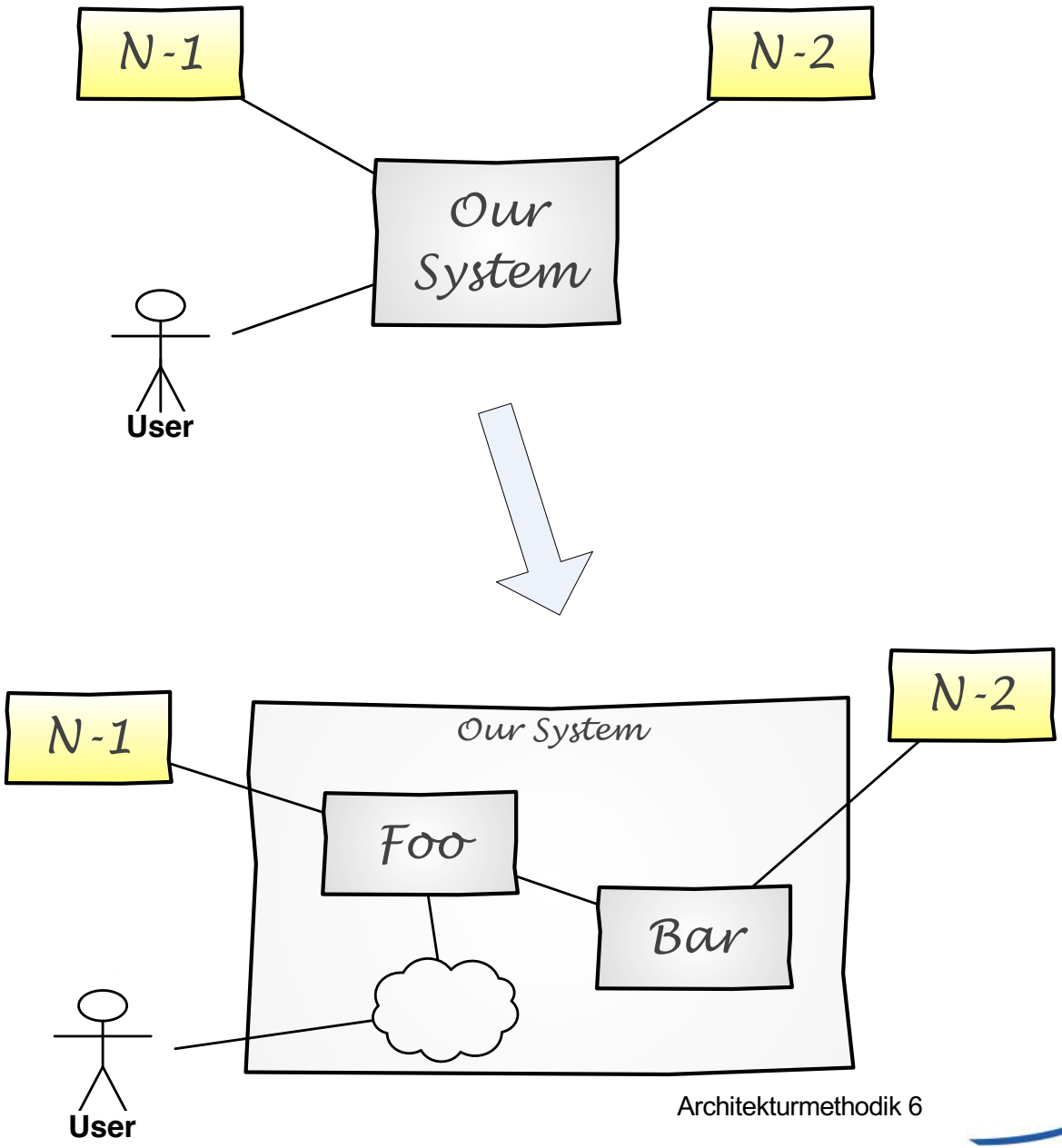
- Technologien (Sprachen, Frameworks, Werkzeuge)
- Make / Buy / Re-Use
- Persistenz
  (Wie werden Daten gespeichert?)
- Benutzerschnittstelle
  (Wie wird UI implementiert?)
- Abläufe/Workflows
  (Wie wird das System gesteuert?)
- Betrieb
  (Wie wird das System betrieben & administriert?)
- (spezielle) Qualitätsanforderungen
  (Wie werden die zentralen Qualitätsanforderungen erreicht?)

Top-Down, Bottom-Up.....



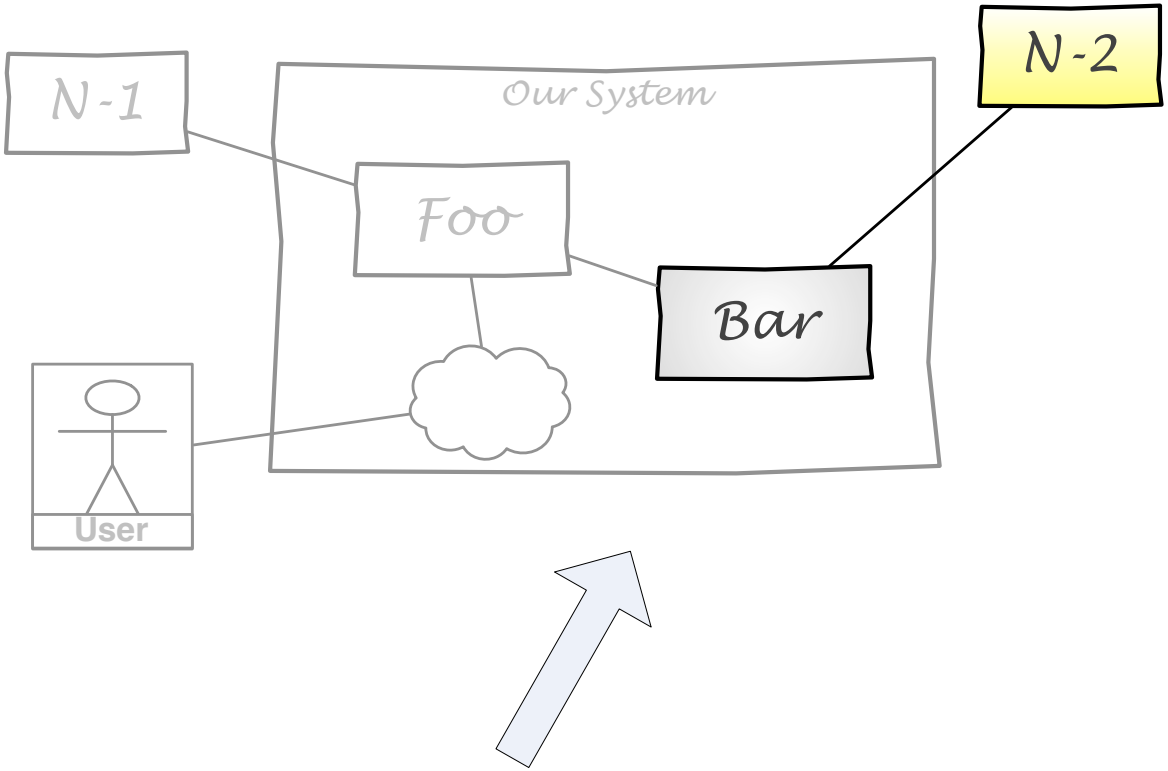
Top-Down...

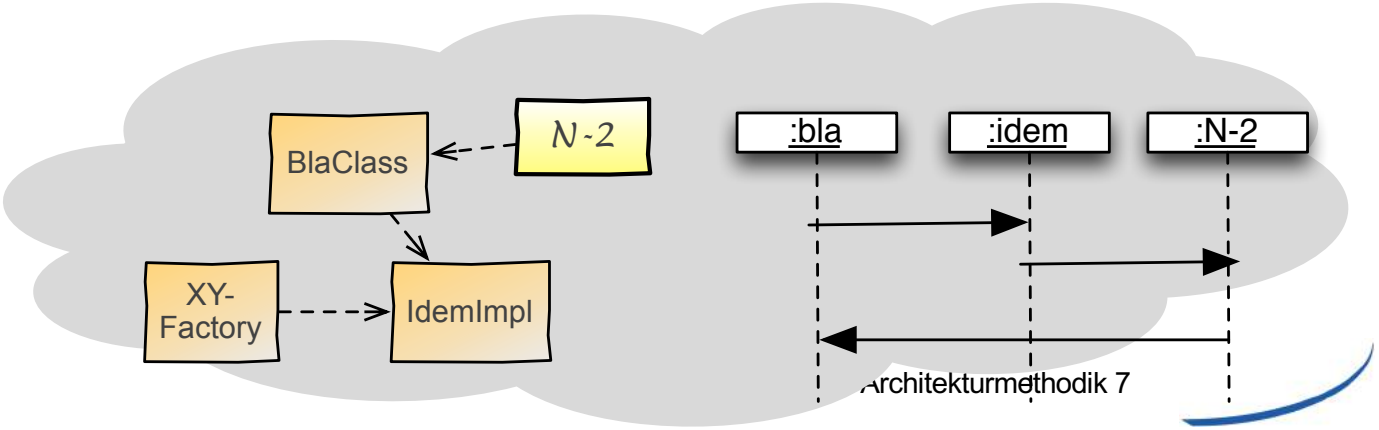
- Grob nach fein
- Abstrakt nach konkret/detailliert



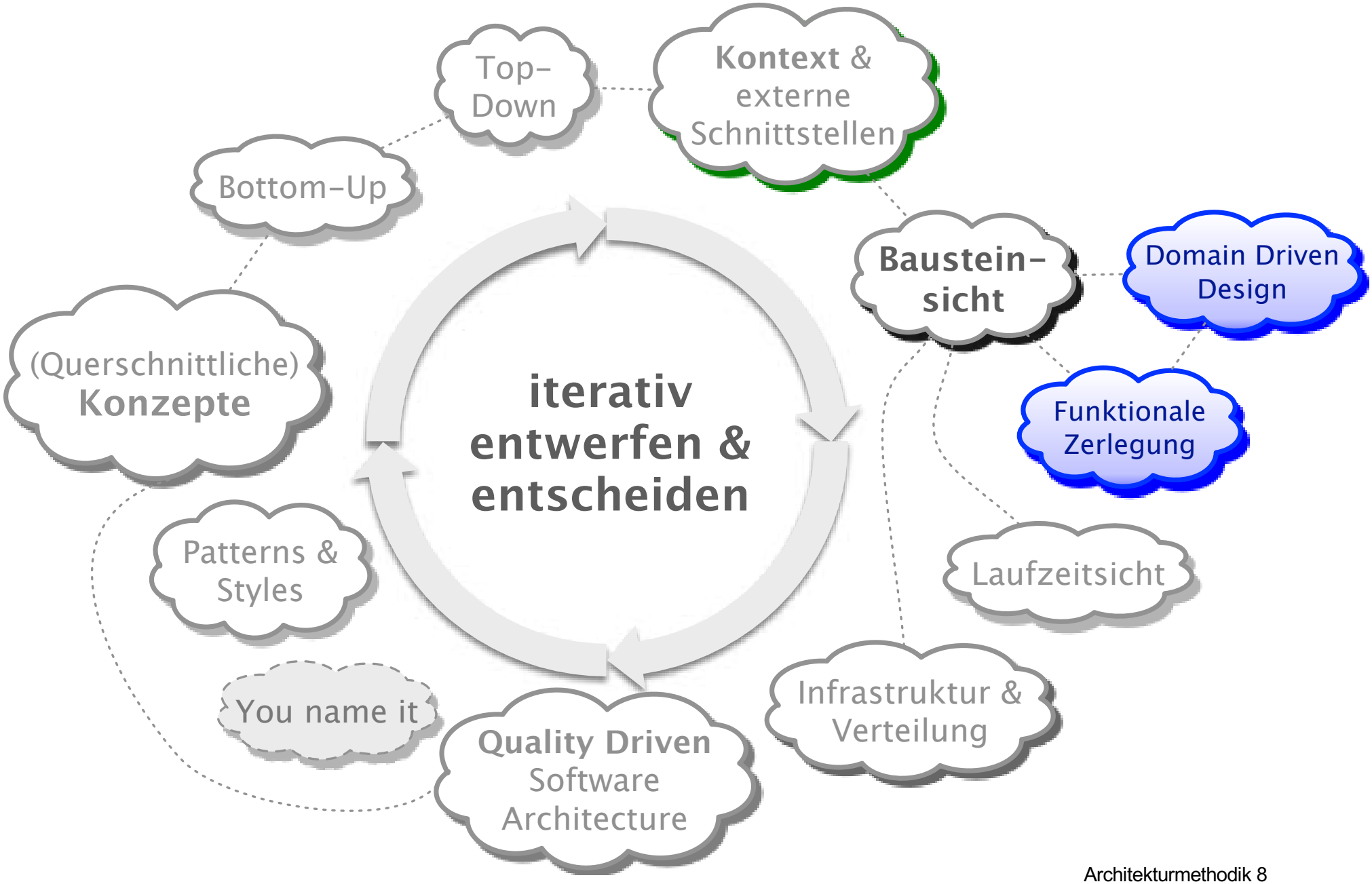
Bottom-Up...

- Fein nach Grob
- Konkret / detailliert nach abstrakt





Domain-Driven Design und Funktionale Zerlegung.



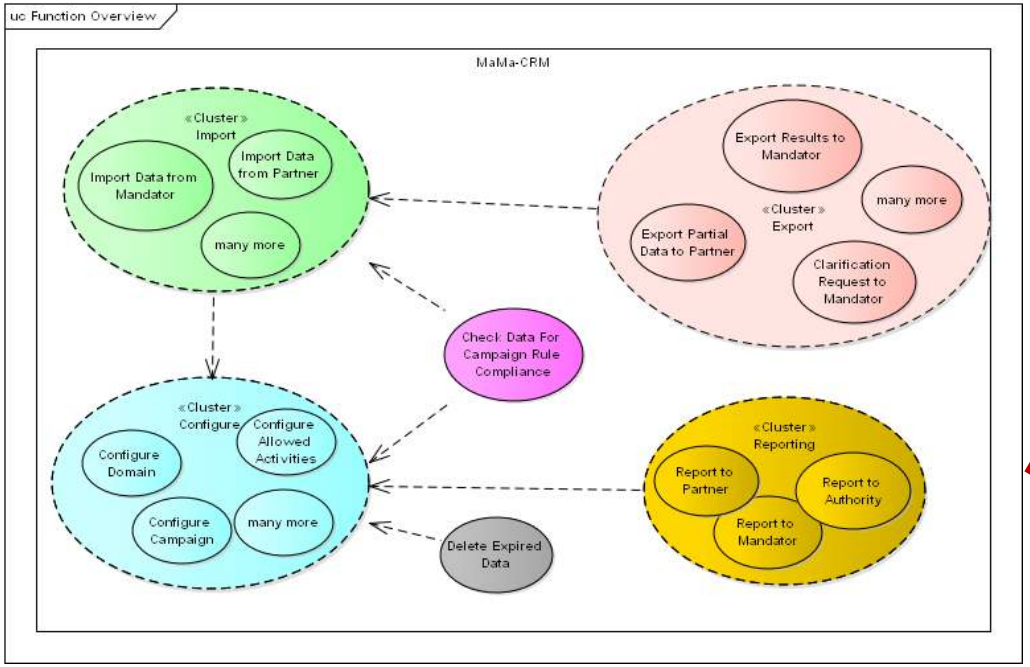
Funktionale Zerlegung

Idee: Aus „**Fachgebieten**“ oder „**fachlichen Clustern**“ Architekturbausteine ableiten

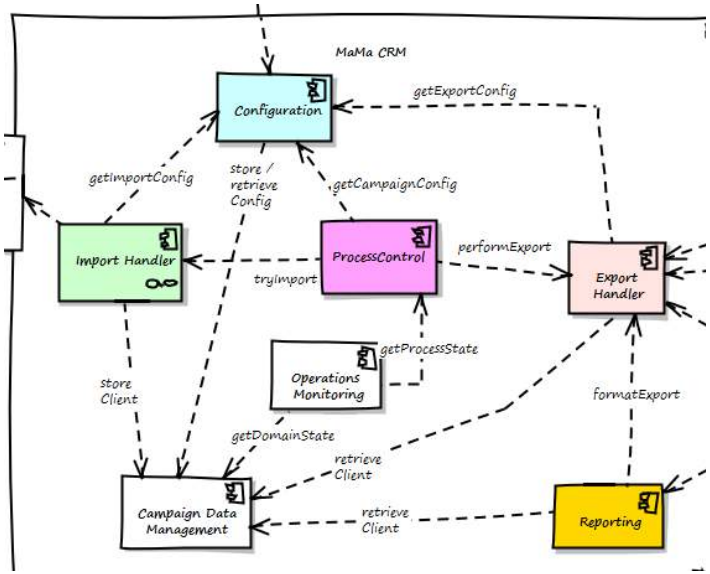
Quelle: Use-Cases, Business- oder Geschäftsprozesse; Epics, Features

1. Use-Cases, Prozesse, Epics gruppieren (➔ **Cluster**)
2. Cluster fachlich sinnvoll benennen
3. Cluster jeweils durch eigene Bausteine implementieren
    a. Technische Redundanz herauslösen
    b. Als eigene Microservices / Self-Contained-Systems betreiben

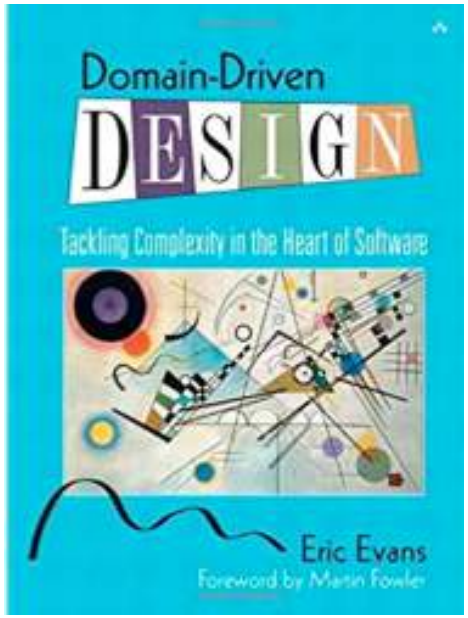
Beispiel „funktionale Zerlegung“



Intuitiv...



Domain Driven Design (E. Evans)

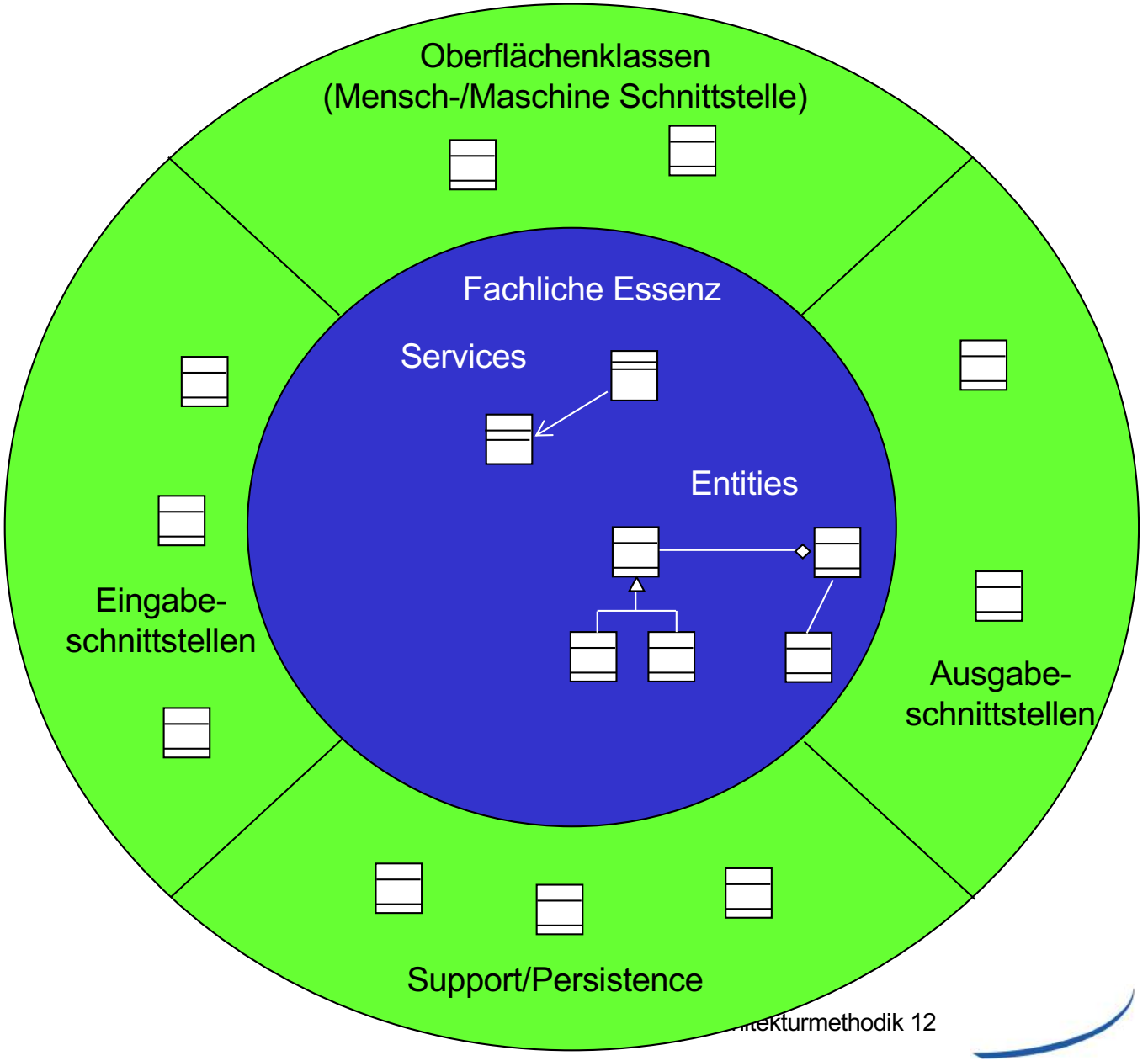


> **Fachlichkeiten** der Anwendungsdomäne **bestimmen Struktur** der Software.

Systematisch und ausführlich...

Domain Driven Design

- Trennen Sie fachliche Bausteine von Technik und Infrastruktur
- Beginnen Sie mit der Fachlichkeit

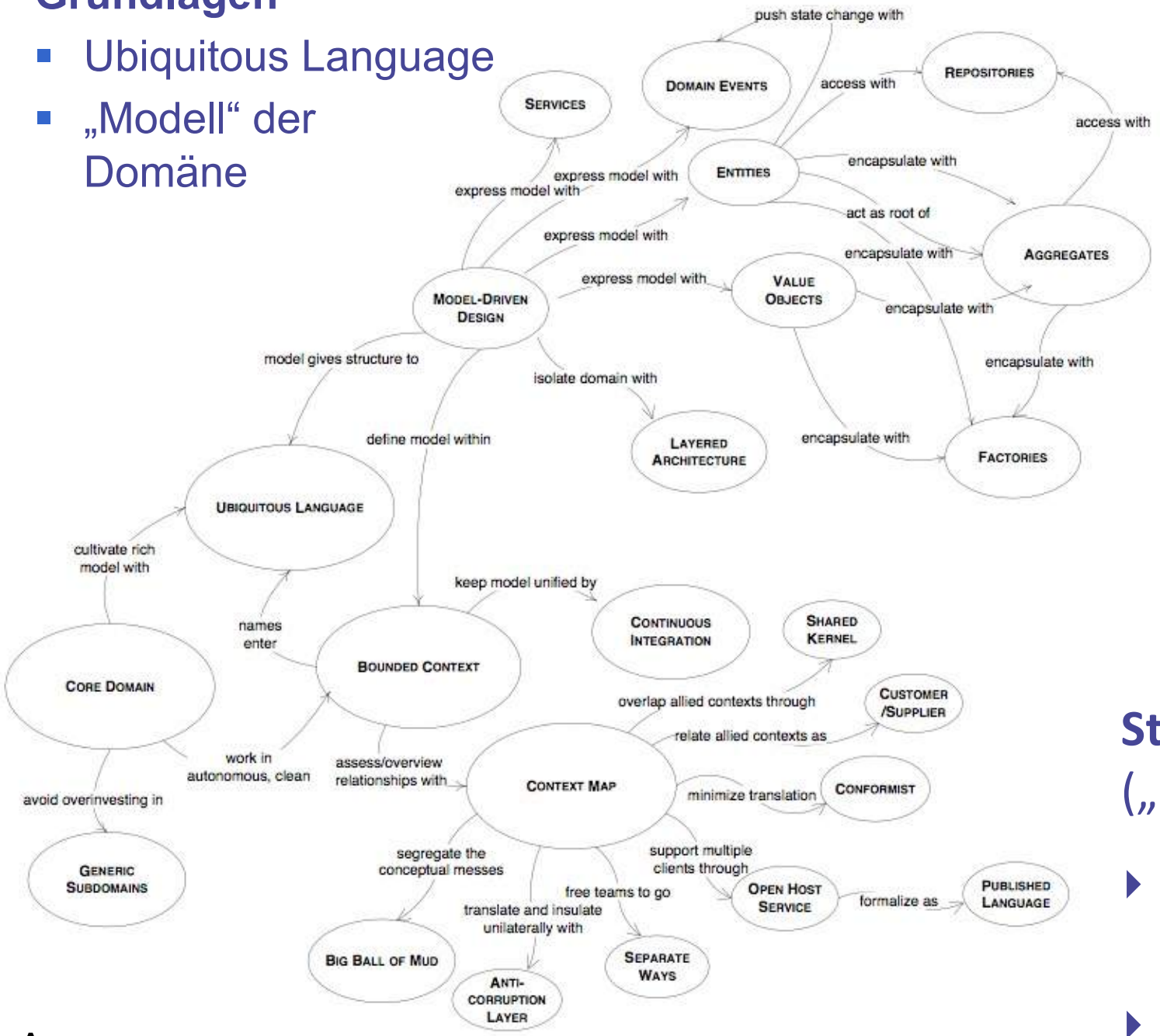


- Onion Architecture*
- Hexagonal Architecture*
- Clean Architecture*

* siehe Folgeseite(n)

Grundlagen

- Ubiquitous Language
- „Modell“ der Domäne



Auszug aus:
http://domainlanguage.com/ddd/reference/

by Eric Evans

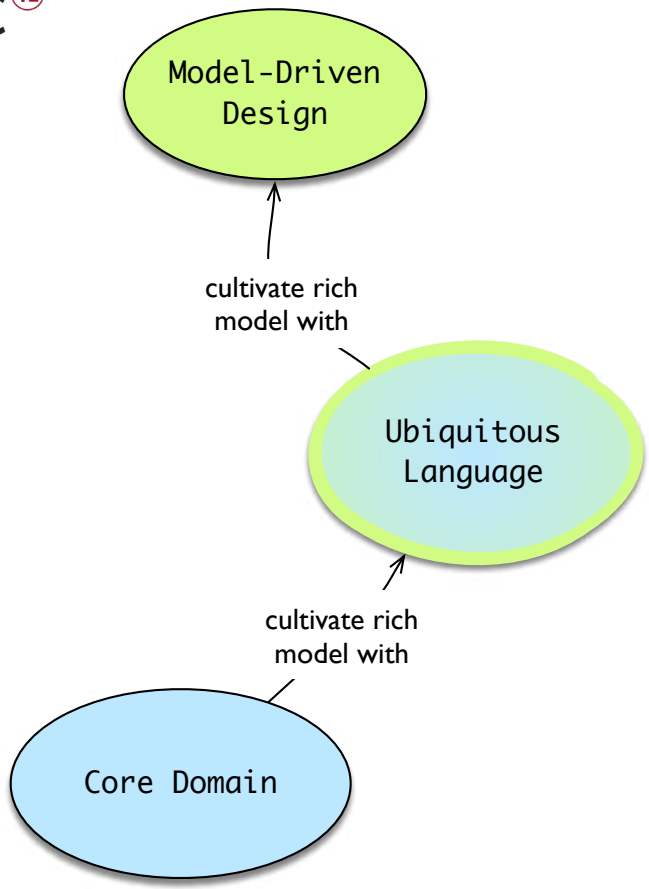
Taktisches Design („im Kleinen“)

- Entities, Value-Objects, Aggregate, Repositories
- Services, Domain-Events

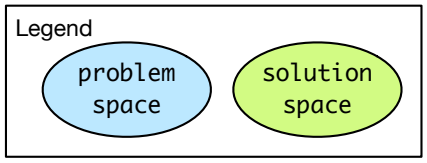
Strategisches Design („im Großen“)

- Bounded Contexts + deren Integration
- Core-/Generic Domains

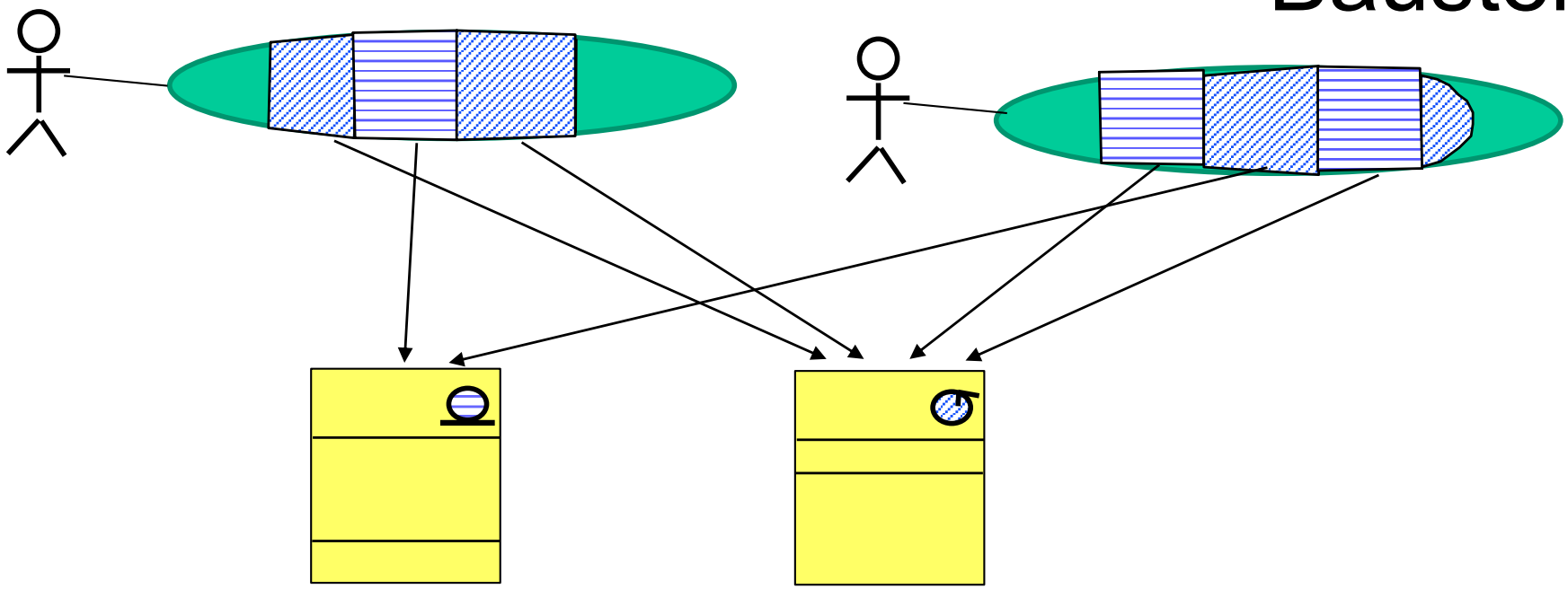
Domain-Driven Design



- ***Domäne:*** Wissens- oder Fachgebiet, bezogen auf das zu lösende Problem.
- Ein ***Domain-Model*** repräsentiert nur lösungsrelevante Teile der Domäne.
  - Domain-Model gehört zum Lösungsraum, die repräsentierte Domäne zum Problemraum.
- Kläre spezifische Worte und Konzepte („***Ubiquitous Language***“) der Domäne.
  - Gemeinsames Vokabular für Fachleute, Entwicklungsteam und Sourcecode!



Use-Cases als Quelle für fachliche Bausteine



hauptsächlich für: Daten

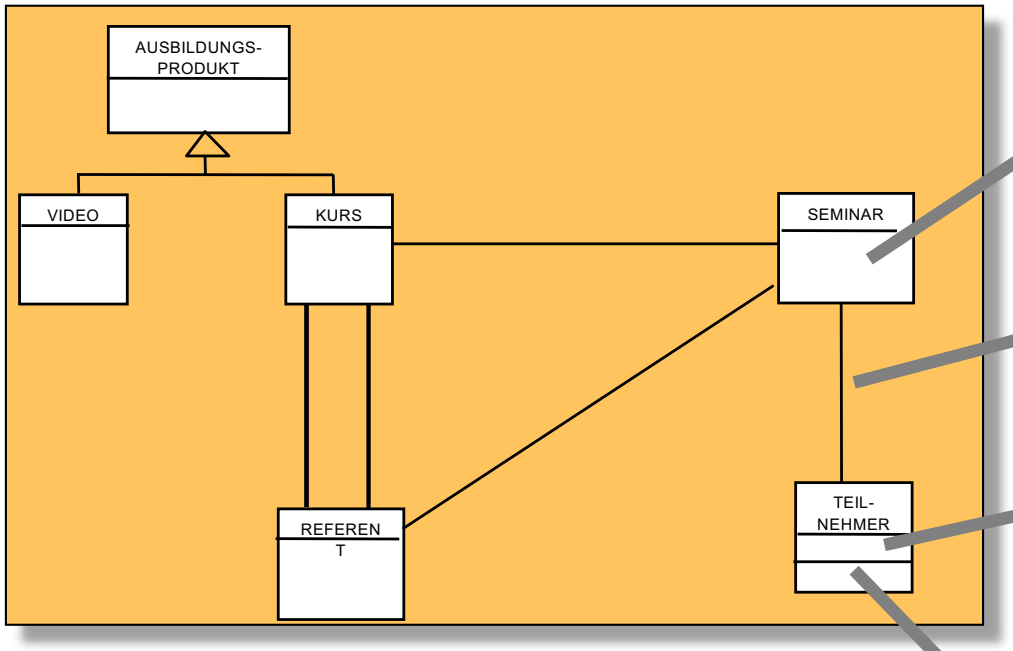
Finden über Heuristiken

Steuerung

Ableiten aus den (hoffentlich dokumentierten) Use-Cases und Aktivitätsdiagrammen samt ihren Spezifikationen

Unser Zwischenziel: Fachliches Entity-Modell

Klassendiagramm & textuelle Spezifikationen



Klassen-spezifikation

Xndkdkd,cx,mklsdkj
Kjsdffjkds
Kjsdffdsdfjkjkds
Sdlkjsdfklj sdflkjsdfljkdsfklj
Sdlkflskdjfk lds sdkjfsl
Sdfoiweyohvnewoi oisd
Oinweoindnvsd
Scztekac.,xvldz

Beziehungs-spezifikation

Xnmklsdkj
Kjsdffjkds
Kjsdffdsdfjkjkds
Sdlkjsdfklj sdflkjsdfljkdsfklj
Sdlkflskdjfklds sdkjfsl
Sdfoiweyohvnewoi oisd
Oinweoindnvsd
Scztekac.,xvldz

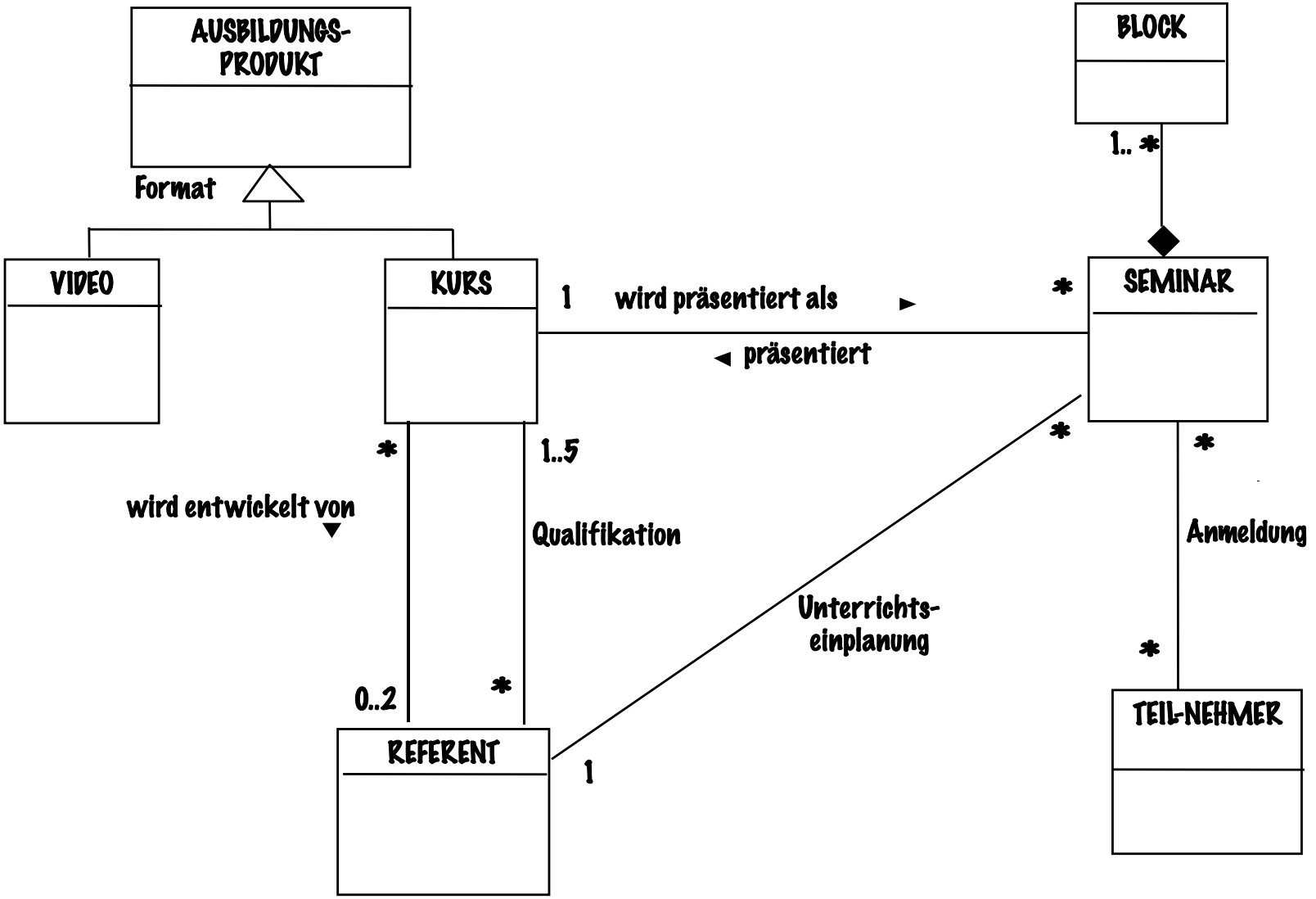
Attribut-spezifikation

Xnmkl sdk dkdsfvcj
Kjsdffjkds
Kjsdffdsdfjkjkds
Sdlkjsdfklj sdflkjsdfljkdsfklj
Sdlkflskdjfklds sdkjfsl
Sdfoiweyohvnewoi oisd
Oinweoindnvsd
Scztekac.,xvldz

Operations-spezifikation

Xnmklsdkj
Sdlkjsdfklj sdflkjsdfljkdsfklj
Sdlkflskdjfklds sdkjfsl
Sdfoiweyohvnewoi oisd
Oinweoindnvsd
Scztekac.,xvldz

Das Seminargeschäft



Tests auf „Entity“...

WENN Sie ein *Ding* für eine Entity halten,
DANN fragen Sie:

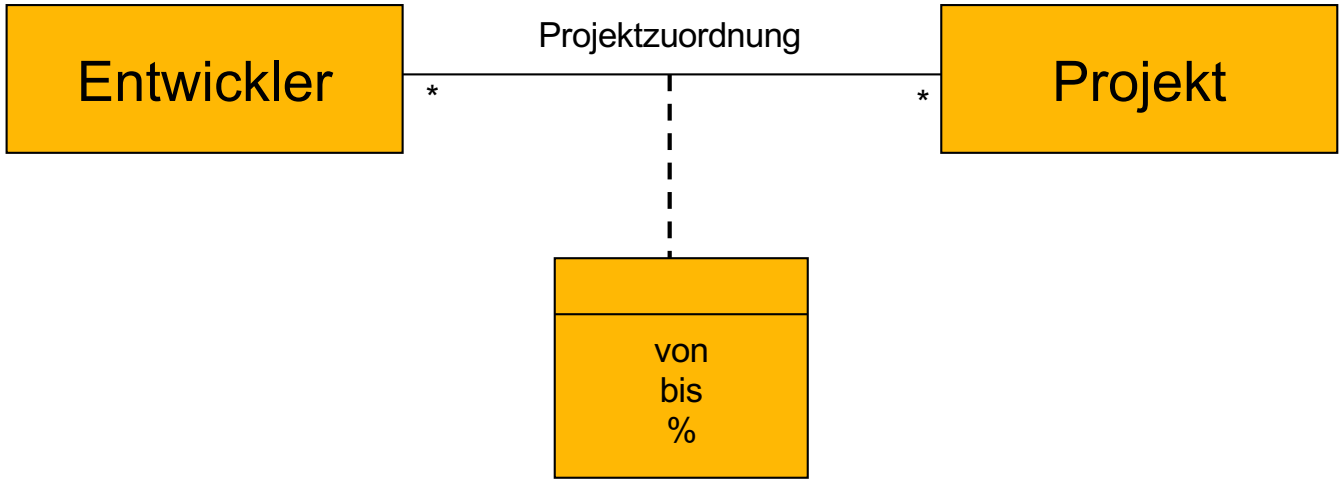
1) Kann man es **identifizieren**? (Macht es Sinn, über ein *bestimmtes Ding* zu sprechen)
2) Finden Sie ein **weiteres Attribut** (außer der Identifizierung)?
3) Gibt es **mehrere** von diesem *Ding*? (Macht es Sinn, über *mehrere Dinge* zu sprechen)
4) **Tut es etwas** oder **wird etwas mit ihm** getan?

Jedes JA als Antwort vergrößert die Wahrscheinlichkeit, dass das *Ding* der Name einer Entity ist.

(Ein einzelnes NEIN führt nicht zur Ablehnung als Entity)

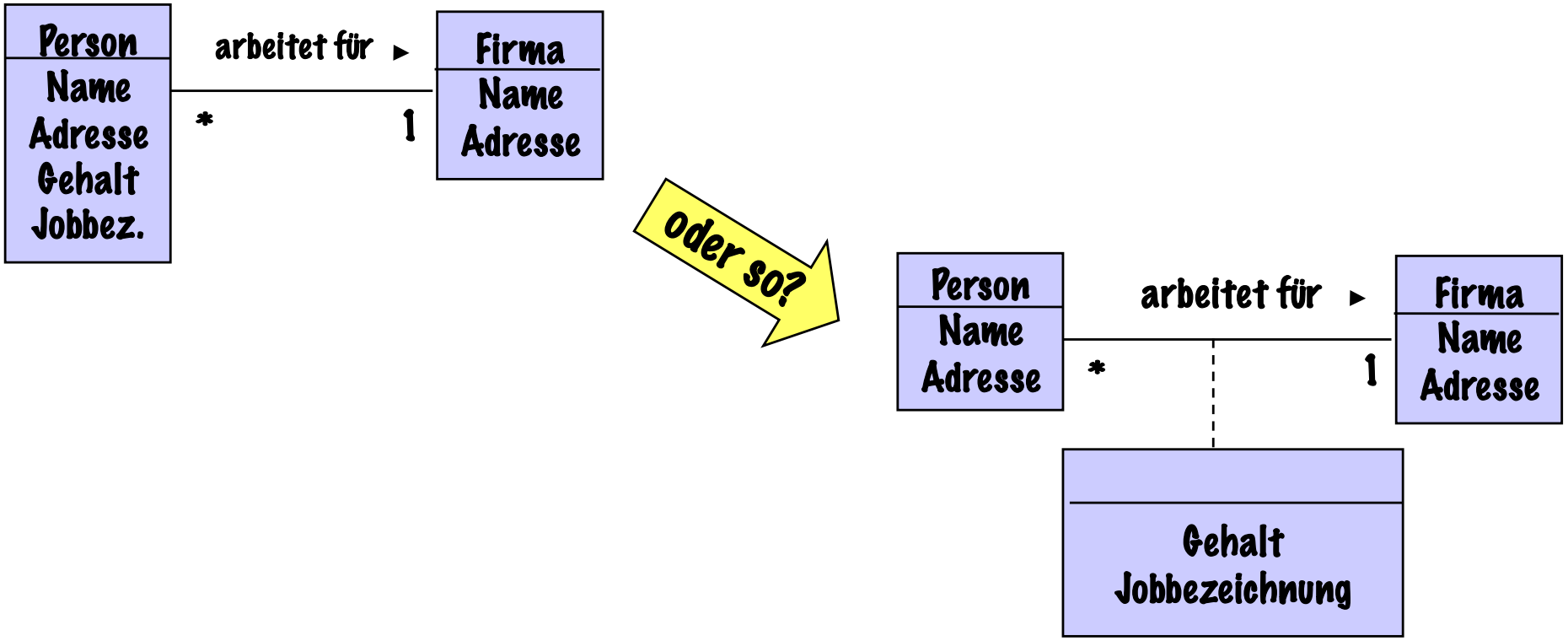
Denken in Beziehungen!

- Analytiker (und Kunden) denken oft in Funktionen
- Architekten sollen in Beziehungen zwischen Entities denken:



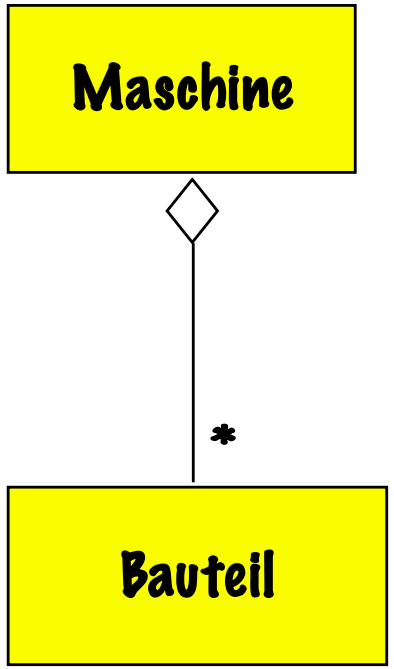
- Beziehungen erlauben es, neue Funktionalität hinzuzufügen ohne die Datenstrukturen zu ändern.

Beziehungsattribute (1)



Teile-Ganze Beziehung

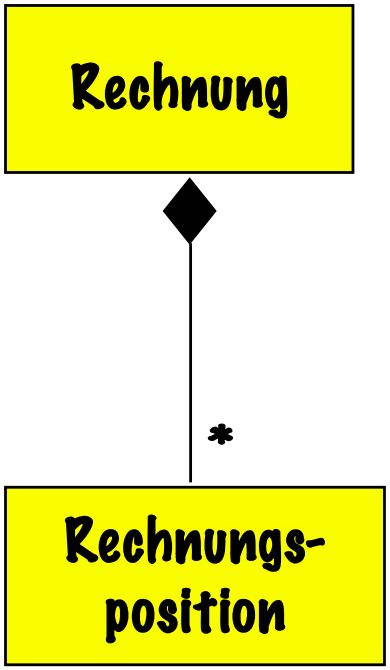
- tritt so oft in der Praxis auf, dass fast alle Methoden dafür eigene Symbole vorsehen:
  - von “oben“ nach “unten“ gelesen: “besteht aus”
  - von “unten“ nach oben” gelesen: “ist Teil von”



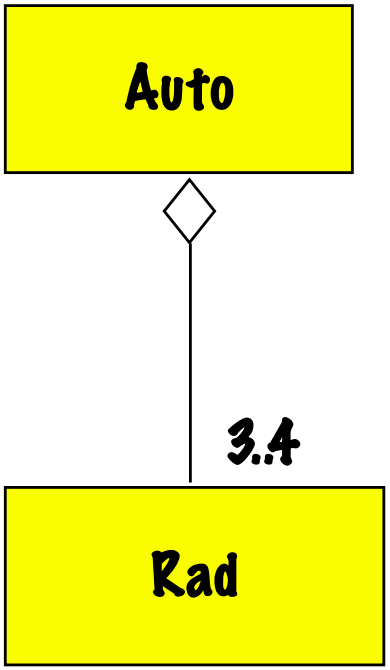
- es ist sehr hilfreich, in größeren Modellen zwischen “normalen” Beziehungen und Teile/Ganze-Beziehungen unterscheiden zu können

Zwei Formen der Teile/Ganze-Beziehung: Komposition ... ... und Aggregation

- Teile sind existenzabhängig vom Ganzen



- Teile sind selbständig lebensfähig

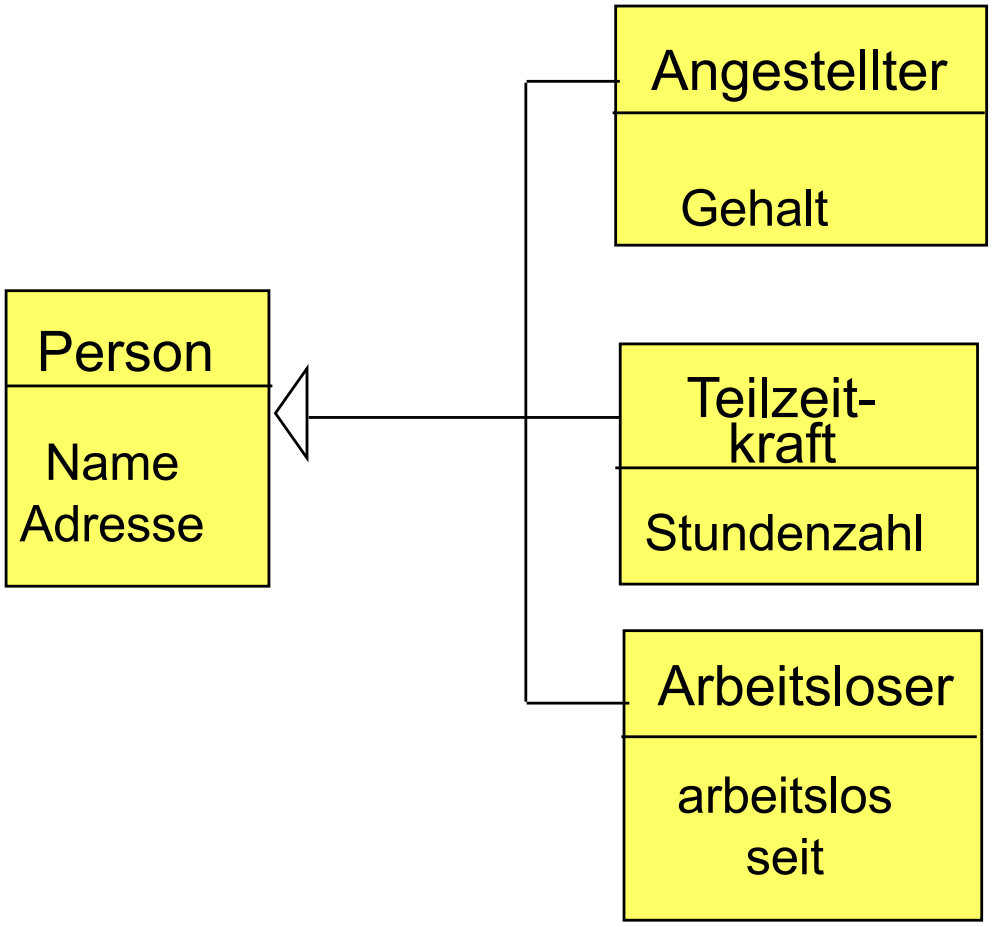


Generalisierung/Spezialisierung

- Gemeinsame Eigenschaften von Unterklassen bei der Oberklasse beschreiben

Unterklassen:

- erben alle gemeinsamen Eigenschaften
- Unterklassen können zusätzliche Attribute haben

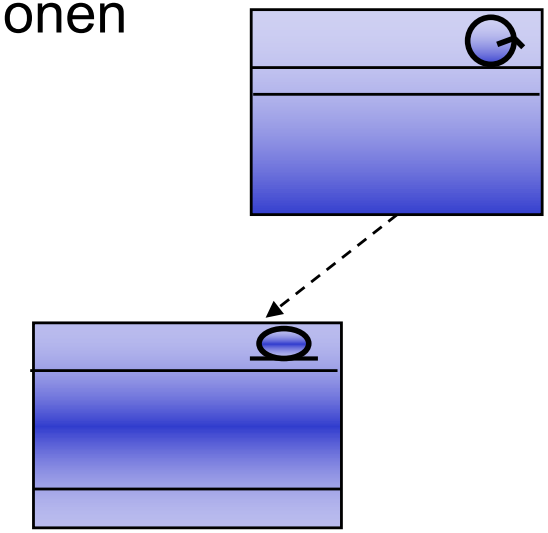


Services
(Steuerungsklassen, Controller)

- Hauptaufgabe der Entities ist die Kapselung von Attributen und das Verfügbarmachen von Zugriffsfunktionen zu den Attributen

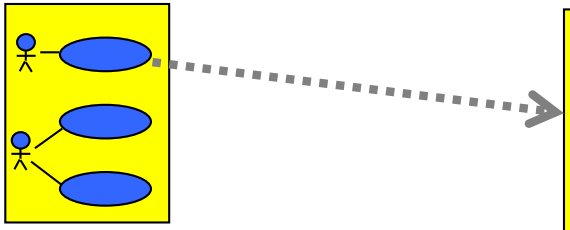
- Hauptaufgabe der Services ist hingegen, die Zusammenarbeit von anderen Bausteinen sicherzustellen, zu steuern und zu überwachen

- Während Entities meist bidirektionale Assoziationen haben, sind Services eher unidirektional mit den zu steuernden Bausteinen verbunden

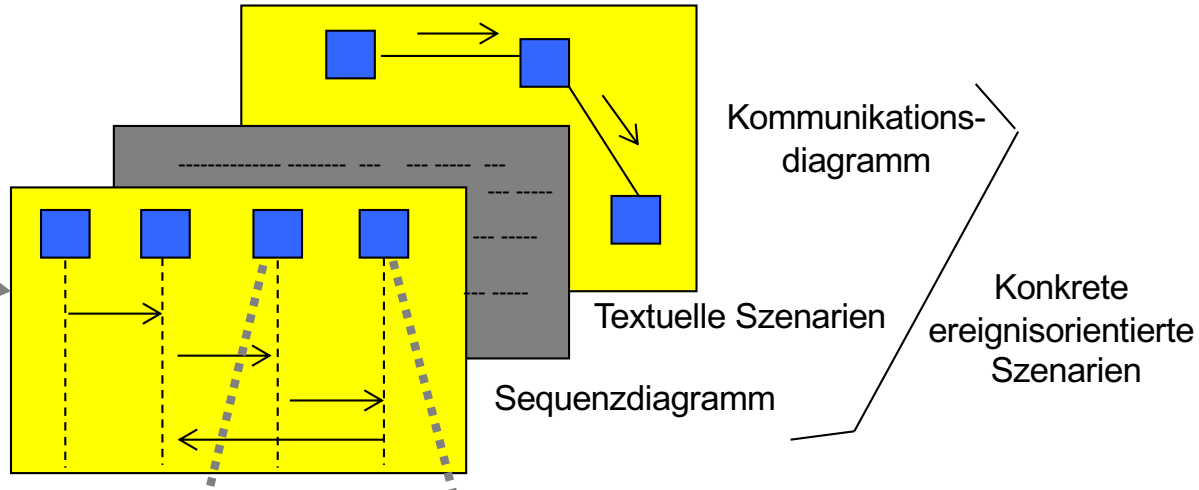


Finden von Services über Szenarien (Event Storming)

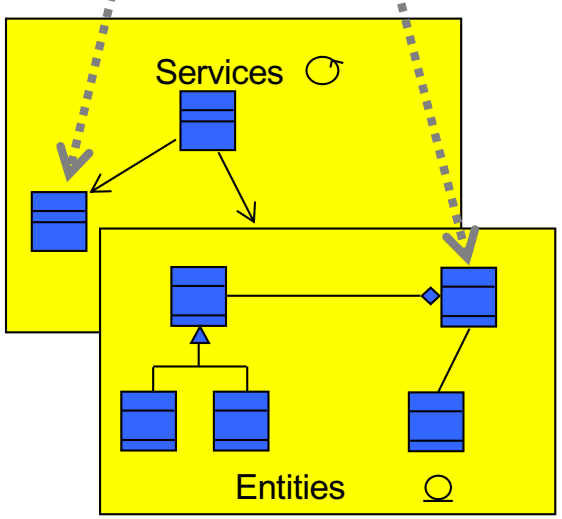
Auslöser = Domain Events



Use-Case-Diagramm



-> besonders leicht, wenn Analytiker bereits Use-Cases spezifiziert haben



Auch Services können Attribute haben

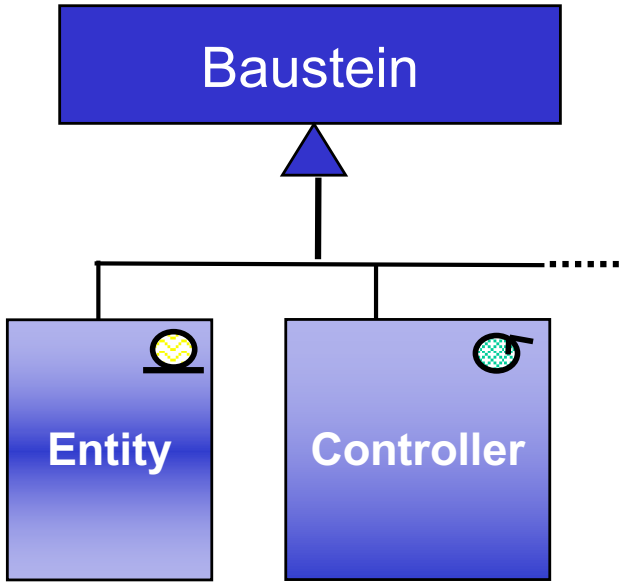
- Services haben typischerweise ihren Schwerpunkt in den Operationen (komplexere Abläufe mit viel Delegation an Untergebene)

- Trotzdem können Services Attribute haben - meist Statusattribute

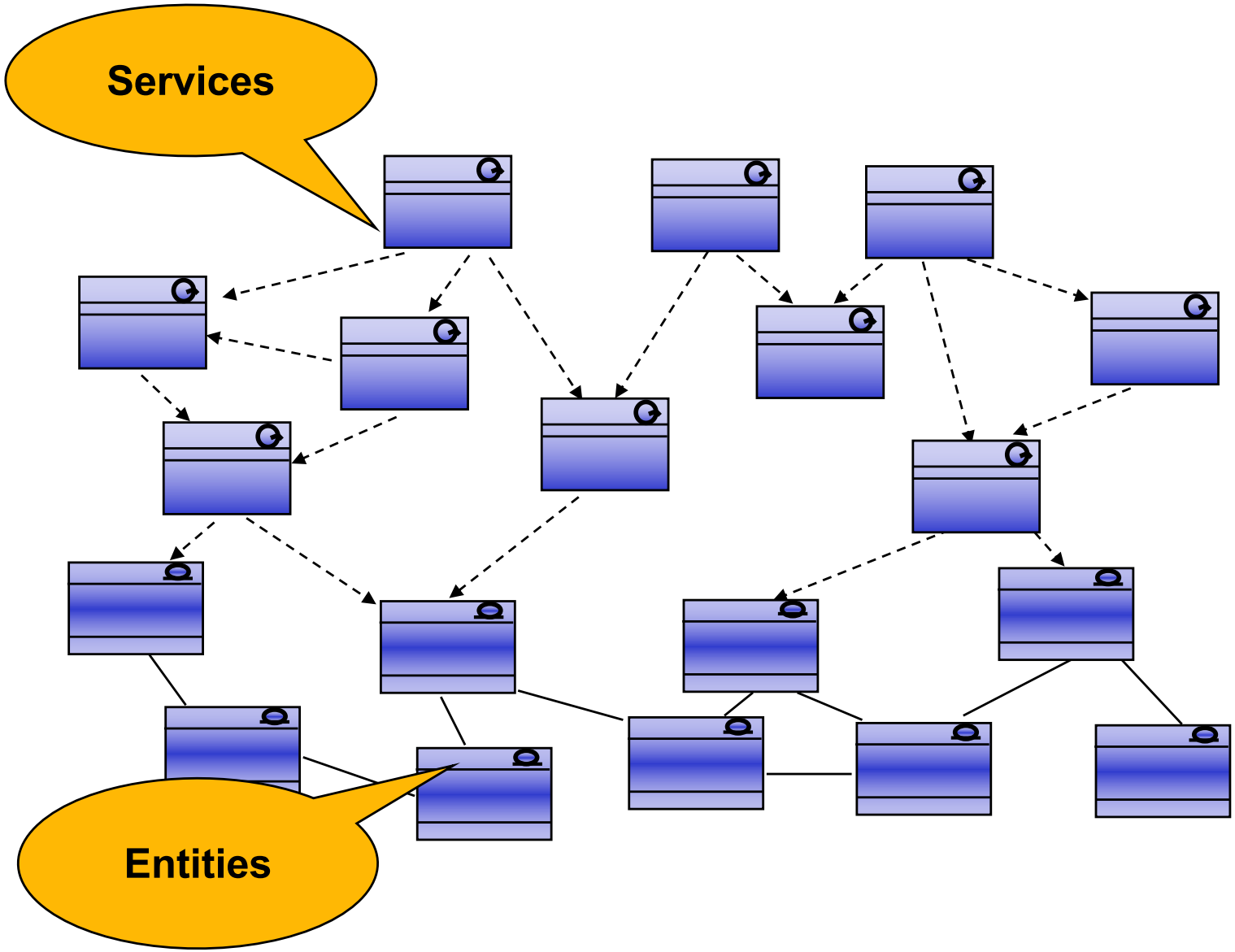
  (Ein Service „merkt sich“ :
  - seinen Zustand,
  - an wen es etwas delegiert hat,
  - bis wann eine Rückmeldung notwendig ist,
  - ....)

Entities & Services: FAT oder LEAN?

- Wann sollte man Operationen schon gefundenen Entities zuordnen?
- Ab wann sollte man eigene Services einführen?



Entities und Services



Entity

- Ein “Ding”, besitzt
  - Identität und Eigenschaften
  - Lebenszyklus
- Das “Modell” muss erklären, was “Gleichheit” bedeutet

Value Object

- Ein Objekt oder Dinge, das einen Teil der Domäne beschreibt, jedoch keine Identität besitzt.
- Nur die Werte der Attribute sind interessant.
- Immutable (kann nicht verändert werden, höchstens durch ein anderes komplett ausgetauscht)

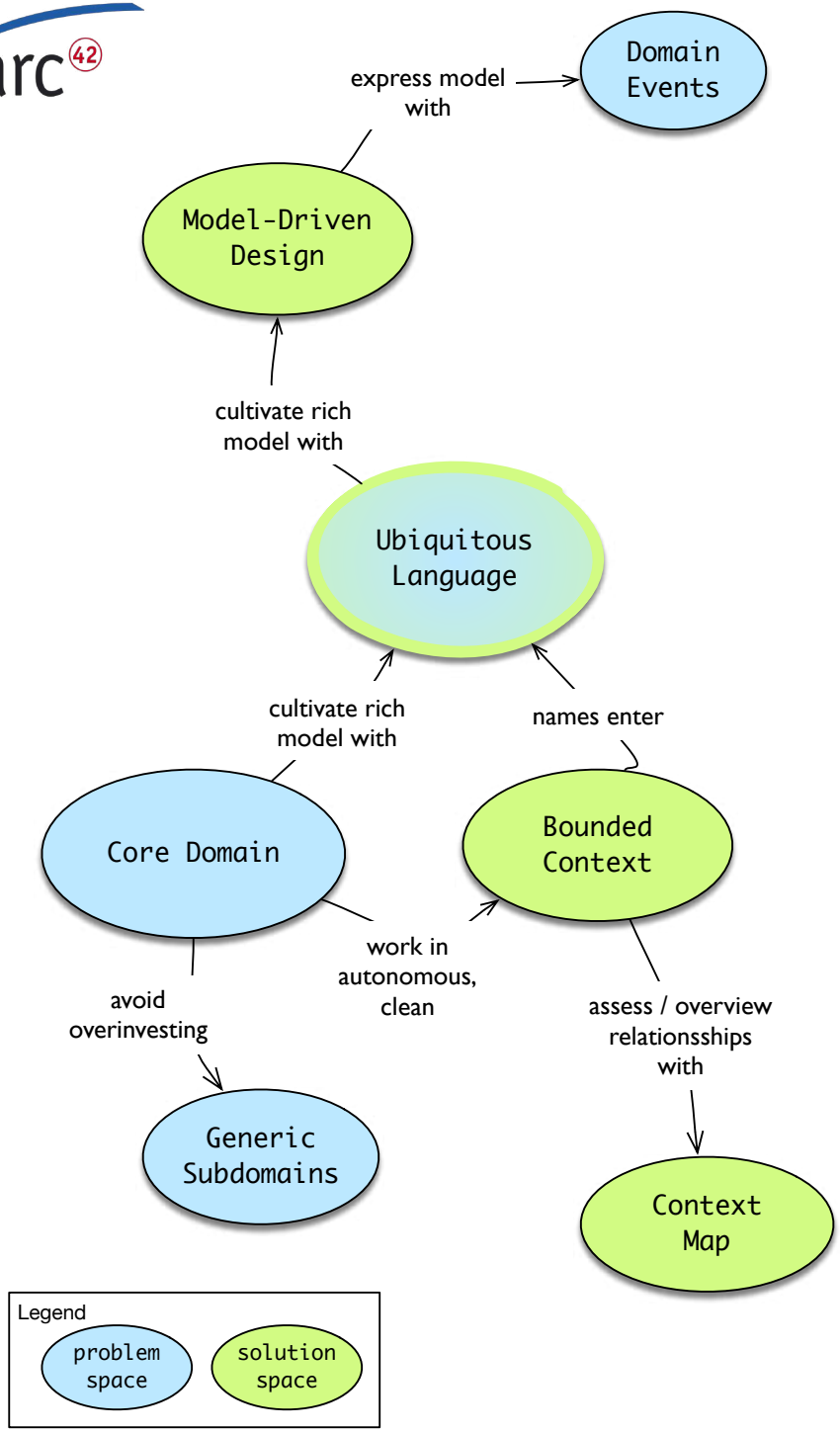
Aggregate

- Fasst mehrere ENTITIES und VALUE OBJECTS zusammen
- Eine **ENTITY als Einstiegspunkt (root)** für jedes AGGREGATE,
  - Zugriff nur über root
  - Externe Objekte dürfen Referenzen nur auf die Root halten

Repository

- Repräsentiert alle Dinge eines bestimmten Typs als “Menge”.
- Baue ein REPOSITORY für jedes Objekt (ENTITY, AGGREGATE), auf das Zugriff benötigt wird
- Entwickle Methoden add, remove und find, die den (technischen) Zugriff auf den (physischen) Datenspeicher kapseln.

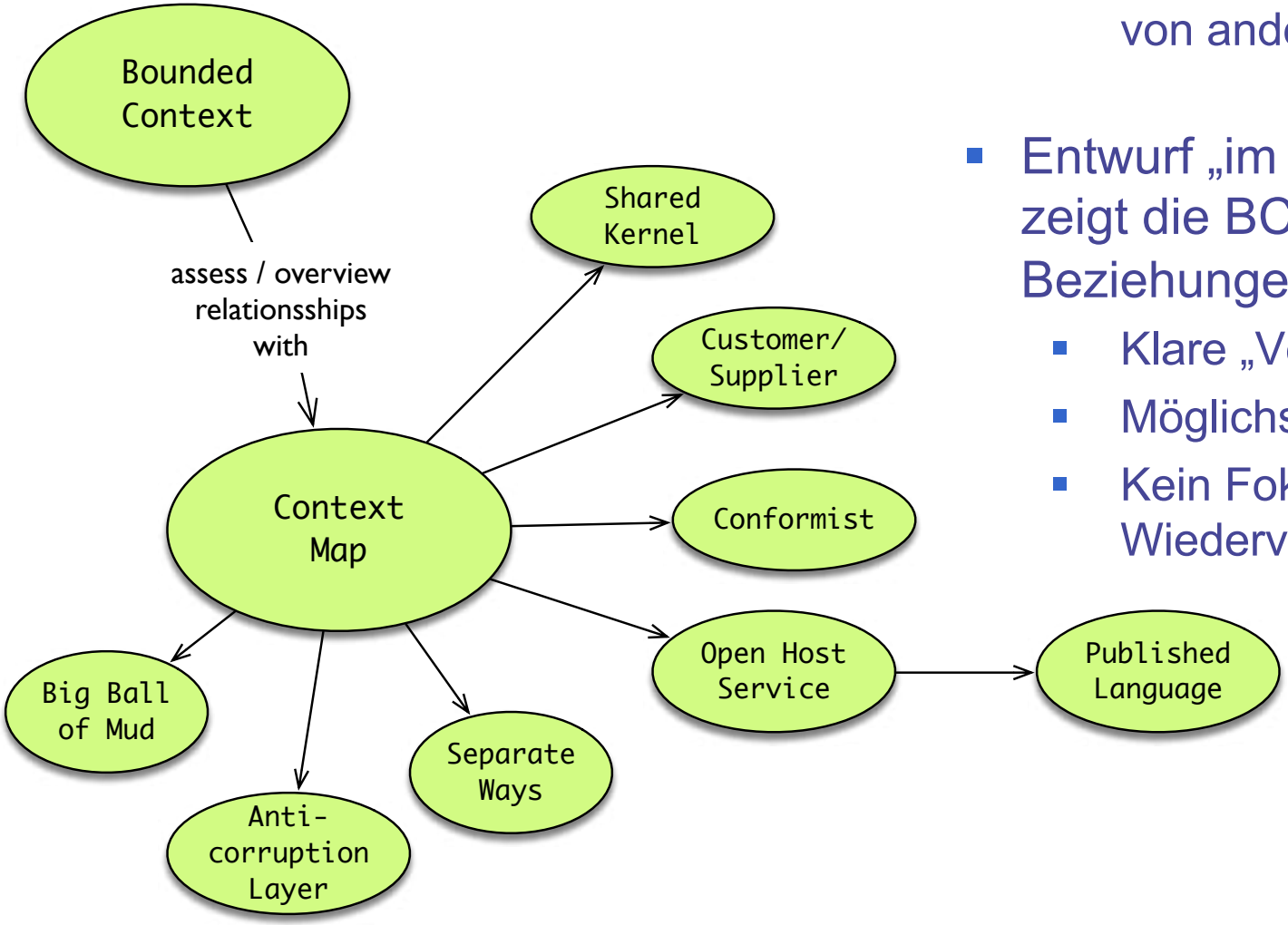
Domain Driven Design



- Analysiere fachlich: „**Core Domains**“ und „Subdomains“
  - Hilfsmittel: „**Domain-Events**“ mit Event-Storming
- Isolierte Bereiche identischer Semantik und hoher fachlicher Kohäsion („**Bounded Context**“), die durch IT unterstützt werden sollen.
- Entwickle darin jeweils spezifische Terminologie („**Ubiquitous Language**“)
- „**Context Map**“ als Landkarte

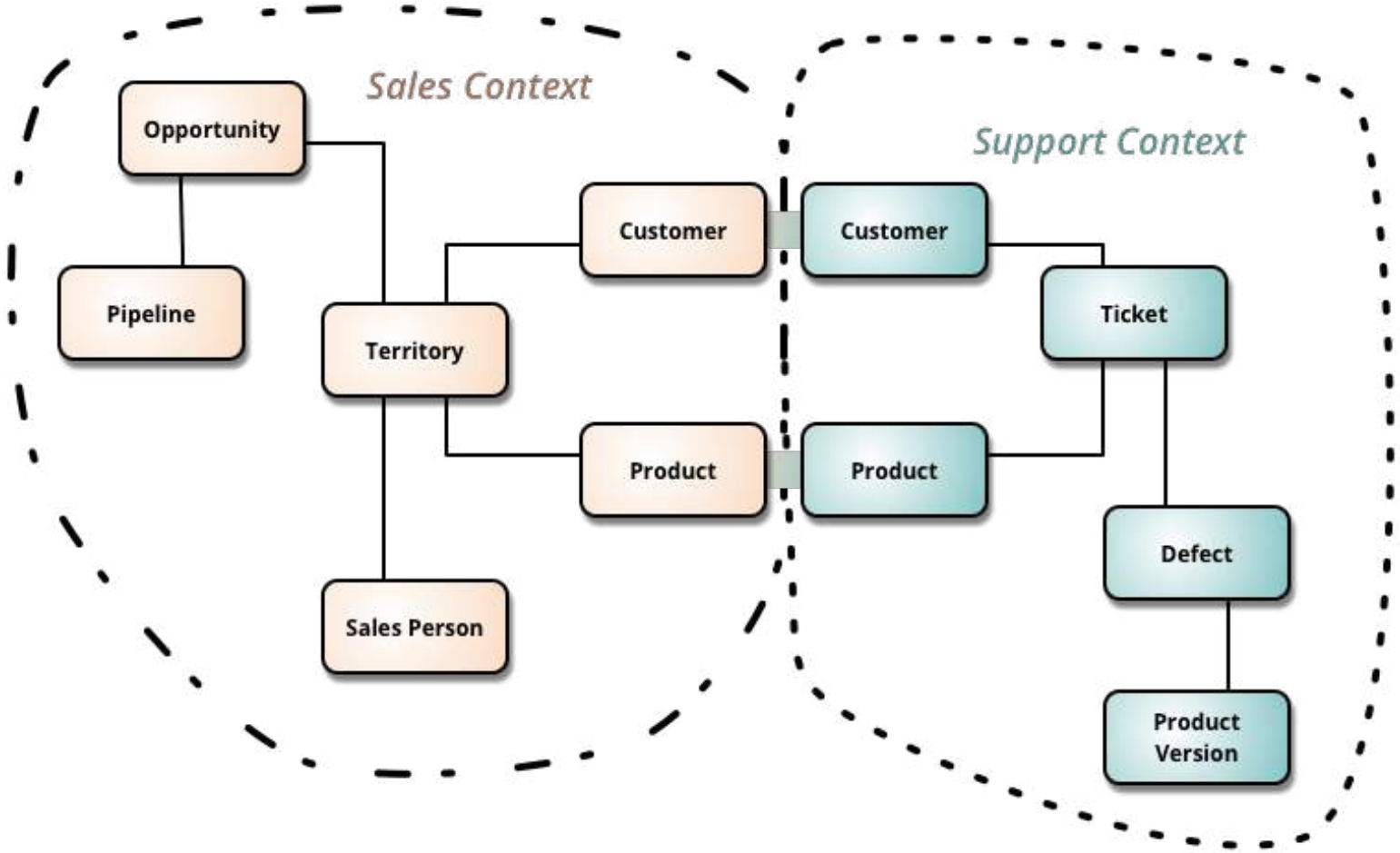
DDD: „Strategic Design“

- ***Bounded Context*** (BC): Subsystem des Lösungsraums
  - klare Grenzen differenzieren ihn von anderen Subsystemen
- Entwurf „im Großen“: ***Context Map*** zeigt die BC und deren Beziehungen.
  - Klare „Verträge“ zwischen BC
  - Möglichst lose Kopplung
  - Kein Fokus auf Wiederverwendung



Beispiel: Bounded Contexts

- inhaltlich zusammengehörige Teile
- Geltungsbereich für fachliche Begriffe + Regeln

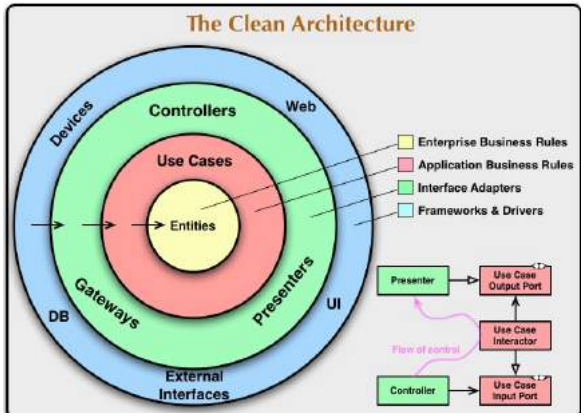


Umsetzung DDD

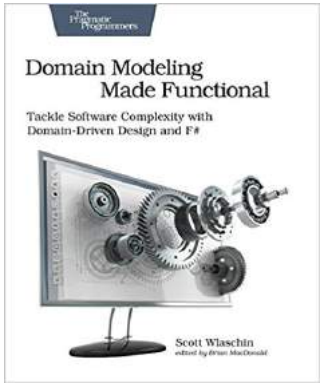
Idee: Fachlicher Kern ohne Abhängigkeiten zur Technik

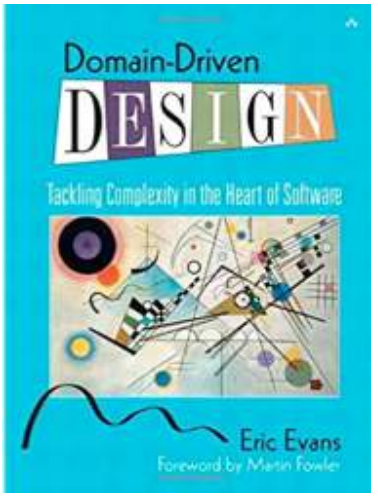
- Aufrufbeziehungen immer von „außen“ nach innen

- Onion-Architecture: Jeffrey Palermo
  - https://jeffreypalermo.com/2008/07/the-onion-architecture-part-1/
- Hexagonal Architecture: Alistair Cockburn
  - https://java-design-patterns.com/blog/build-maintainable-systems-with-hexagonal-architecture/
- Clean Architecture: Robert Martin
  - http://blog.cleancoder.com/uncle-bob/2012/08/13/the-clean-architecture.html



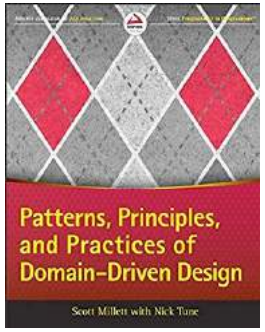
Literatur zu DDD



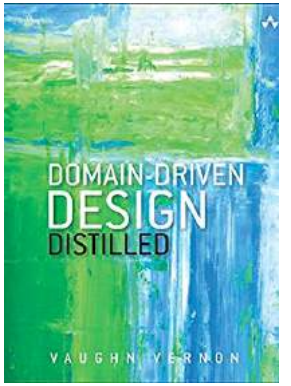


Eric Evans: Domain Driven Design (DDD)
(650+ Seiten)

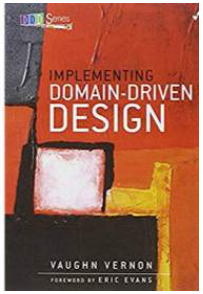
Scott Wlaschin: Domain Modeling made Functional
(300 Seiten, 2017)



Vaughn Vernon: DDD distilled (160 Seiten),
auf Deutsch bei dpunkt-Verlag, 2017



Scott Millet: Patterns, Principles and Practices of DDD
(600+ Seiten)

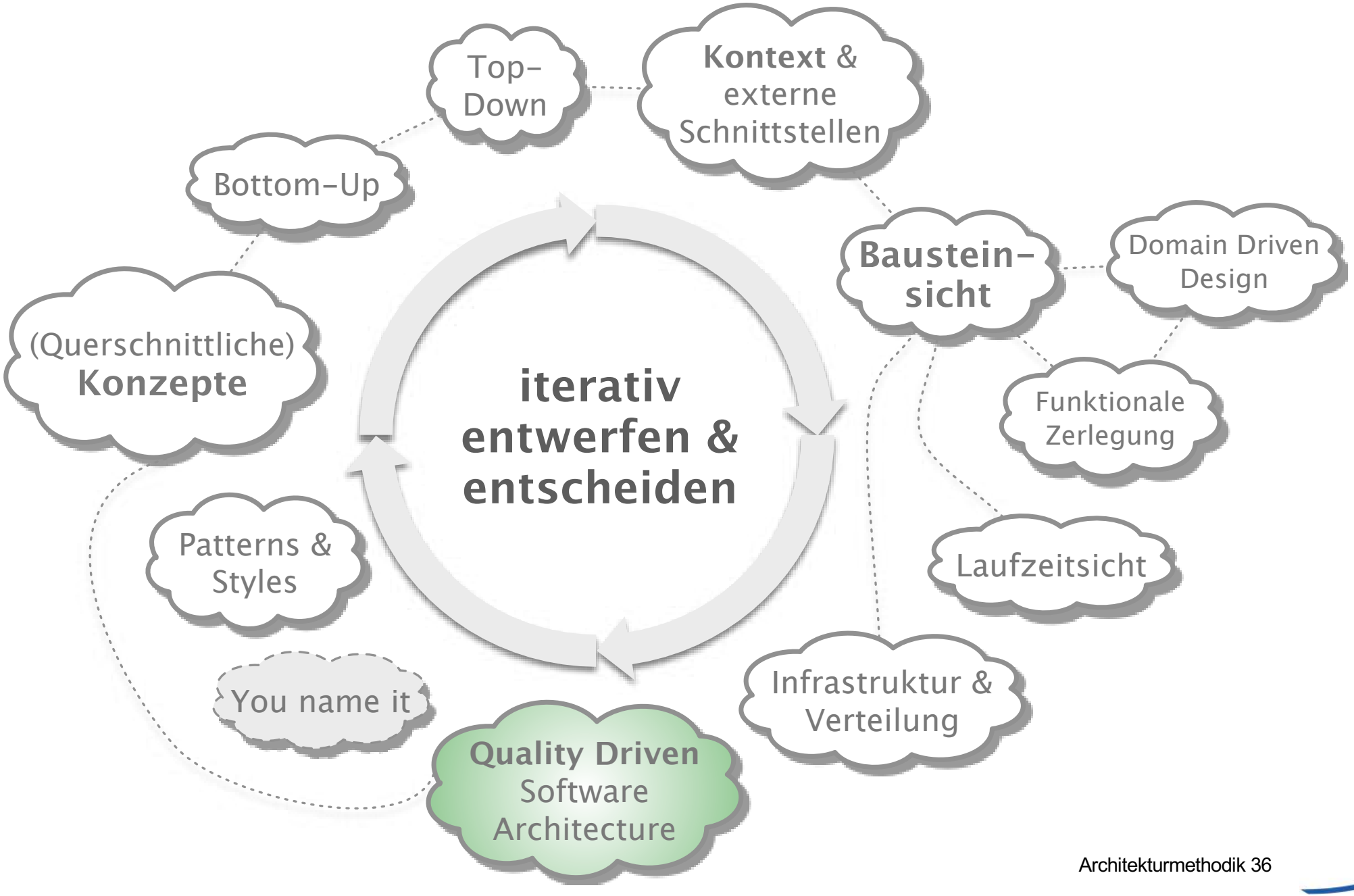


Vaughn Vernon: Implementing Domain-Driven Design (650+ Seiten)

InfoQ: Domain-Driven-Design Quickly (kostenfreies pdf, 50 Seiten)
www.infoq.com/minibooks/domain-driven-design-quickly

Michael Plöd: DDD by Example (Leanpub)

Quality Driven Software Architecture



Quality-Driven Software Architecture

A.k.a. „Global Analysis“ ([Hofmeister+99])

Analysiere relevante Qualitätsanforderungen.
Entwickle Strategien zu ihrer Lösung.

1. Beschreibe Qualitätsanforderungen,
   konkret entscheid- oder messbar

2. Entwickle Strategien zur Erreichung dieser Anforderungen
   - Beachte Wechselwirkungen!

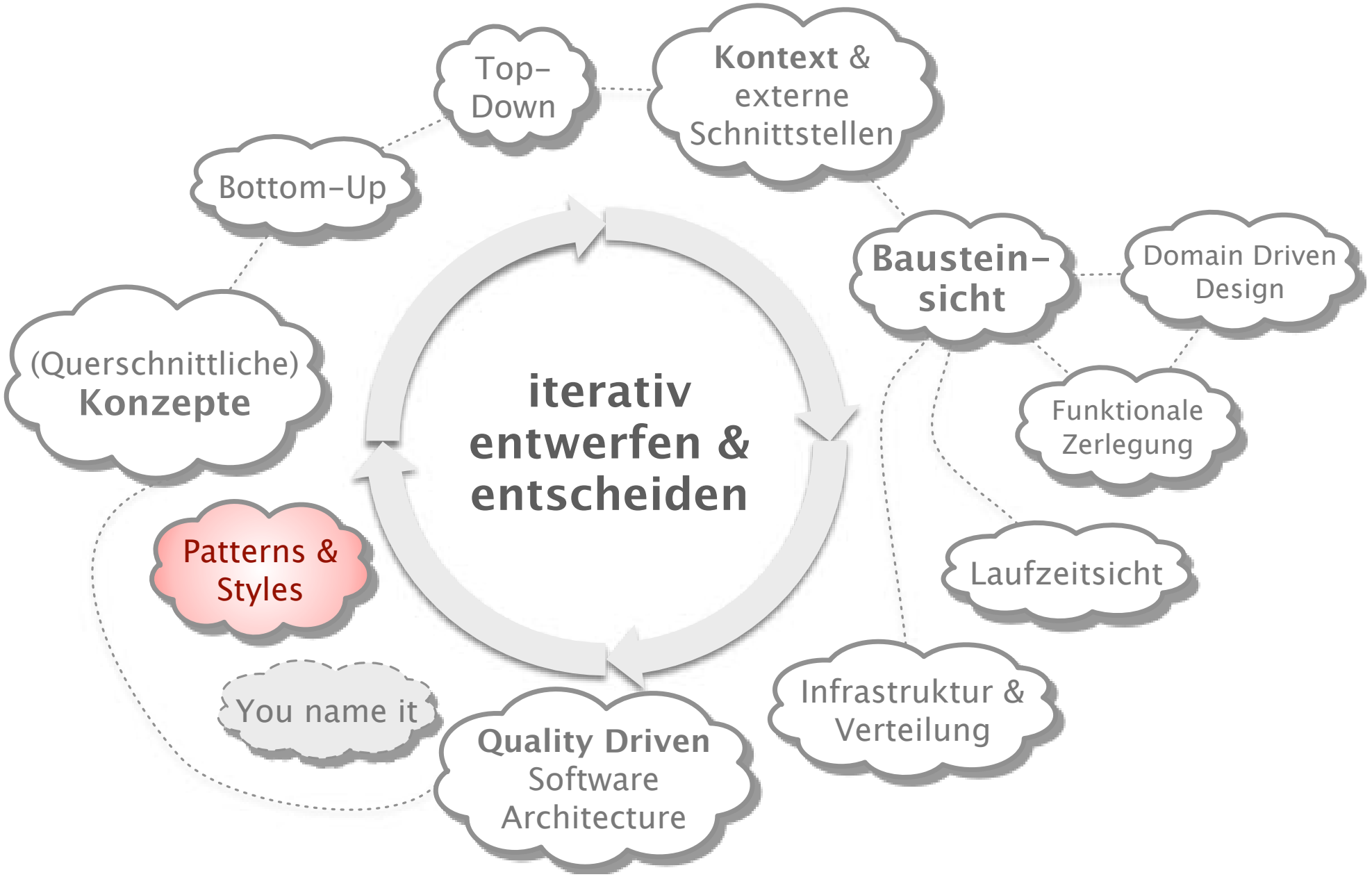
[Hofmeister+99] Applied Software-Architecture – A Practical Guide. Addison-Wesley.

QDSA: Formuliere Maßnahmen explizit

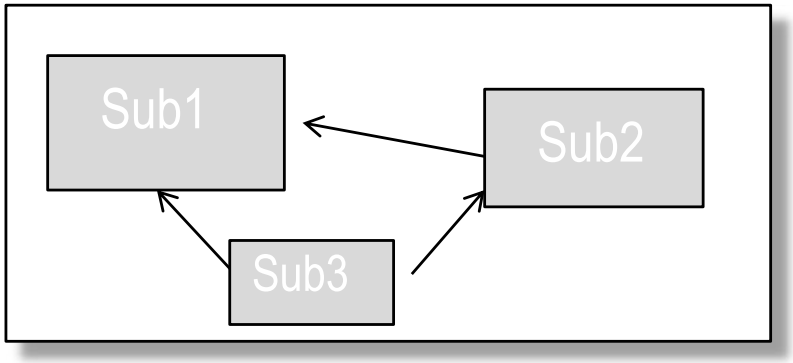
- Ergänze Q-Anforderungen durch konkrete Maßnahmen(vorschläge) pro Szenario!
- Kann Anforderungen, Entwurf, Implementierung, Betrieb und Organisation umfassen!

Prio	Q-Ziel	Konkrete Anforderung/ Szenarien	**Taktiken / Maßnahmen / Entscheidungen**	**Konsequenzen**
1	Performance	Import 250T jpg Bilder inkl. Metadaten (je 10MB) inkl. Verarbeitungsentscheidung In < 24h	• Geschäftsregeln in Jboss-Drools (statt Python) • importiere Bilder nur ins Dateisystem, lediglich File-URL in DB, zusammen mit Metadaten • Täglicher Lasttests	• Risiko: inode Größe Filesystem
2	Flexibilität	Konfiguration csv-Eingabeformates zur Laufzeit	• Entwickeln einer DSL für csv-Formate • Syntaxgesteuerter Editor für diese DSL	
3	Robustheit	.....		
3	Security	Credentials per Hardware HSM generieren	??	Risiko: Kein Know-How im Team

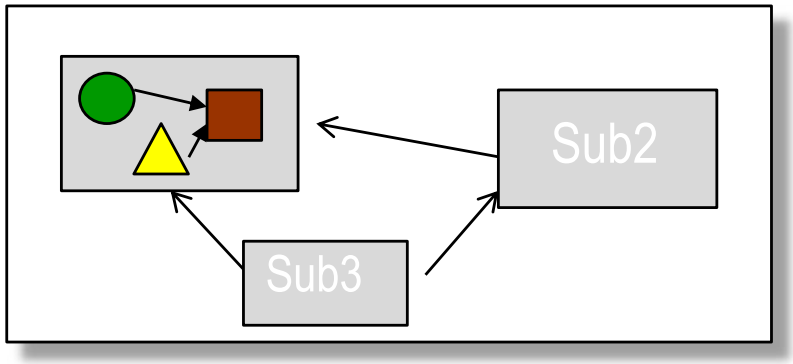
Systematisch abschreiben....



Architektur- und Entwurfsmuster, Idiome

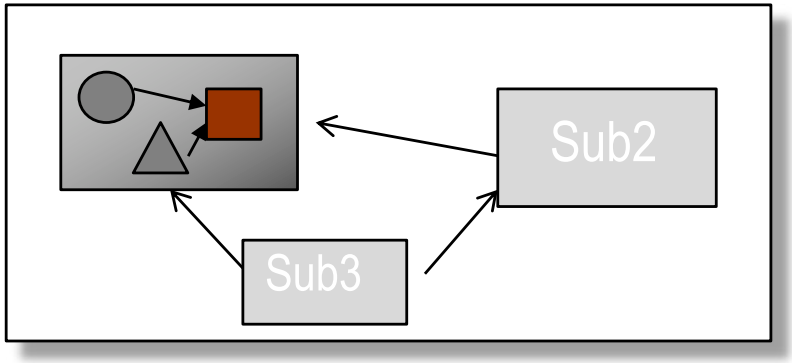


- Architekturmuster:
  Struktur des Gesamtsystems

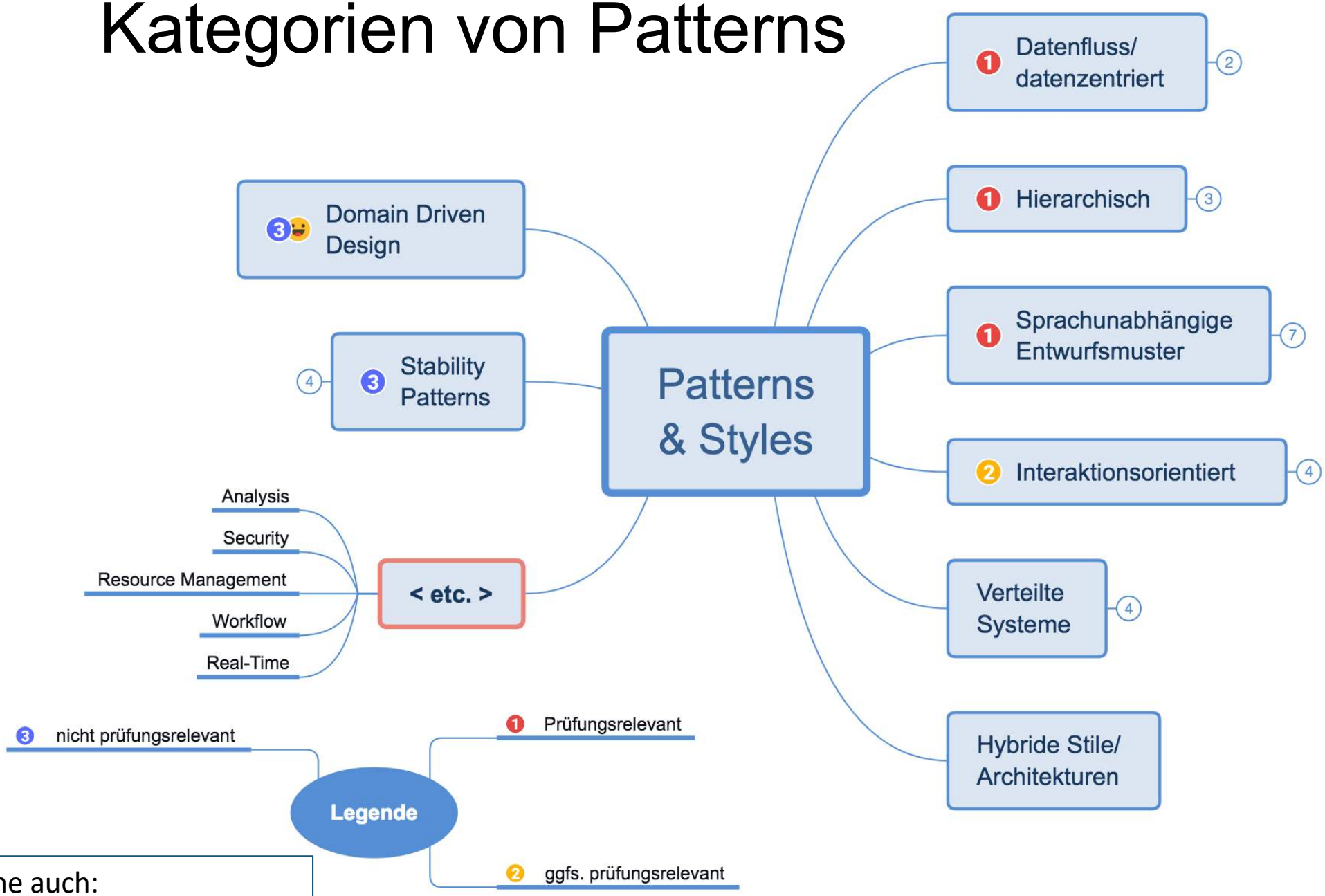


- Entwurfsmuster:
  Struktur von Teilsystemen oder Klassenverbünden
  (mehr dazu bei “Designprinzipien”)

- Idiome:
  sprachenspezifische Lösung von Detailproblemen



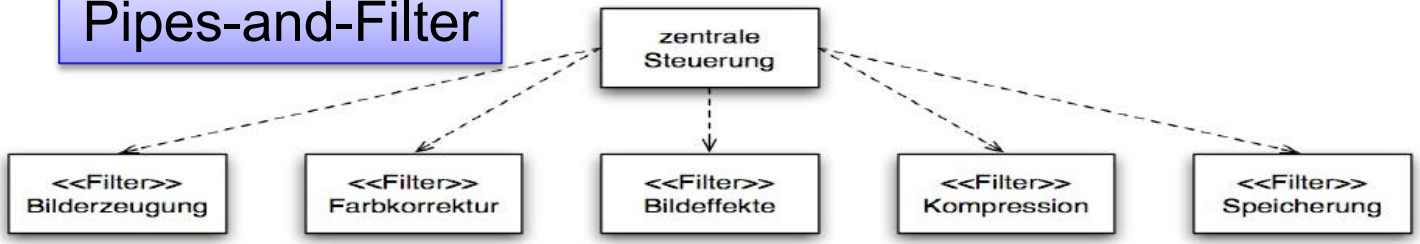
Kategorien von Patterns



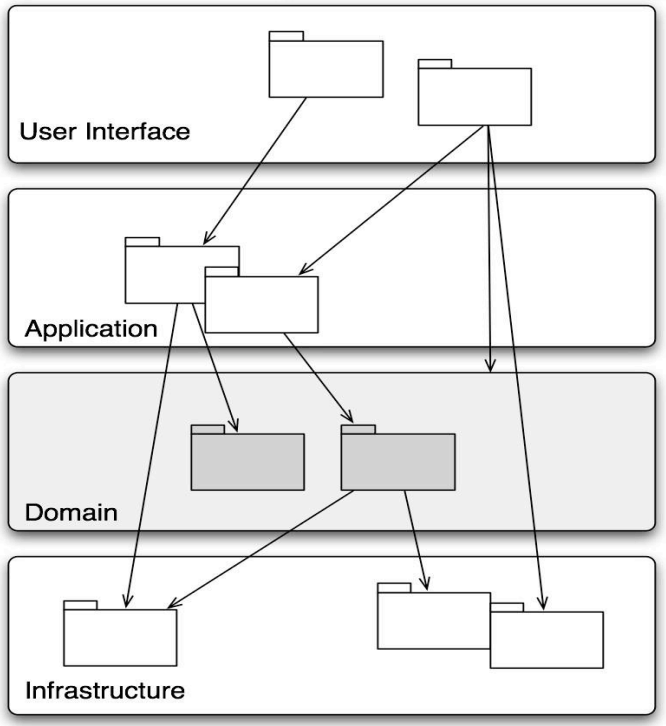
Siehe auch:
http://patterns.arc42.org

Systematisch abschreiben: Architekturmuster

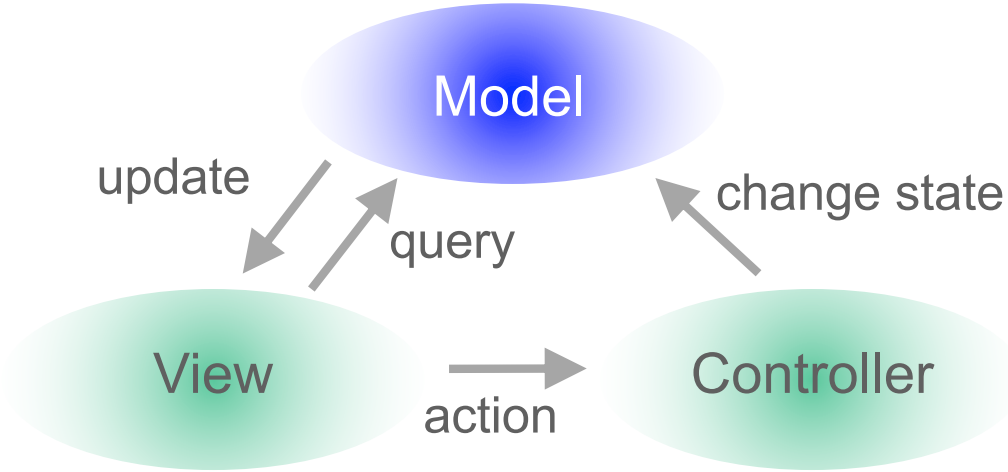
Pipes-and-Filter



Schichten/Layer

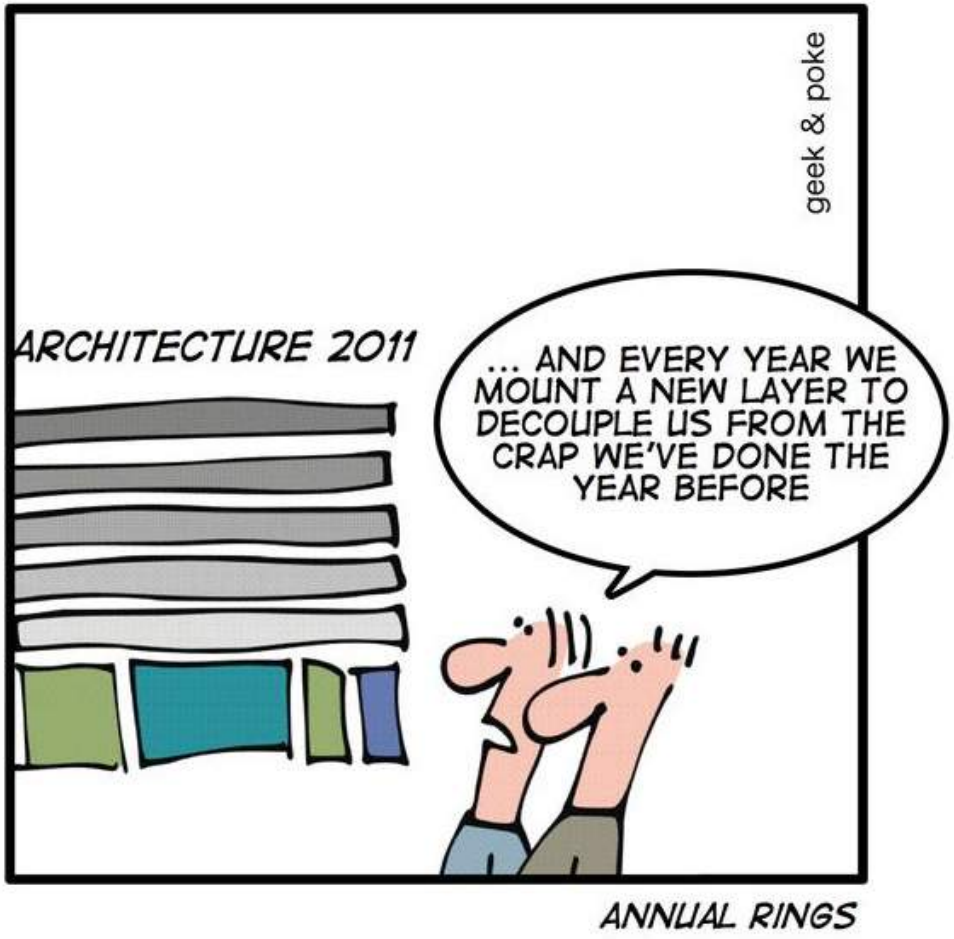


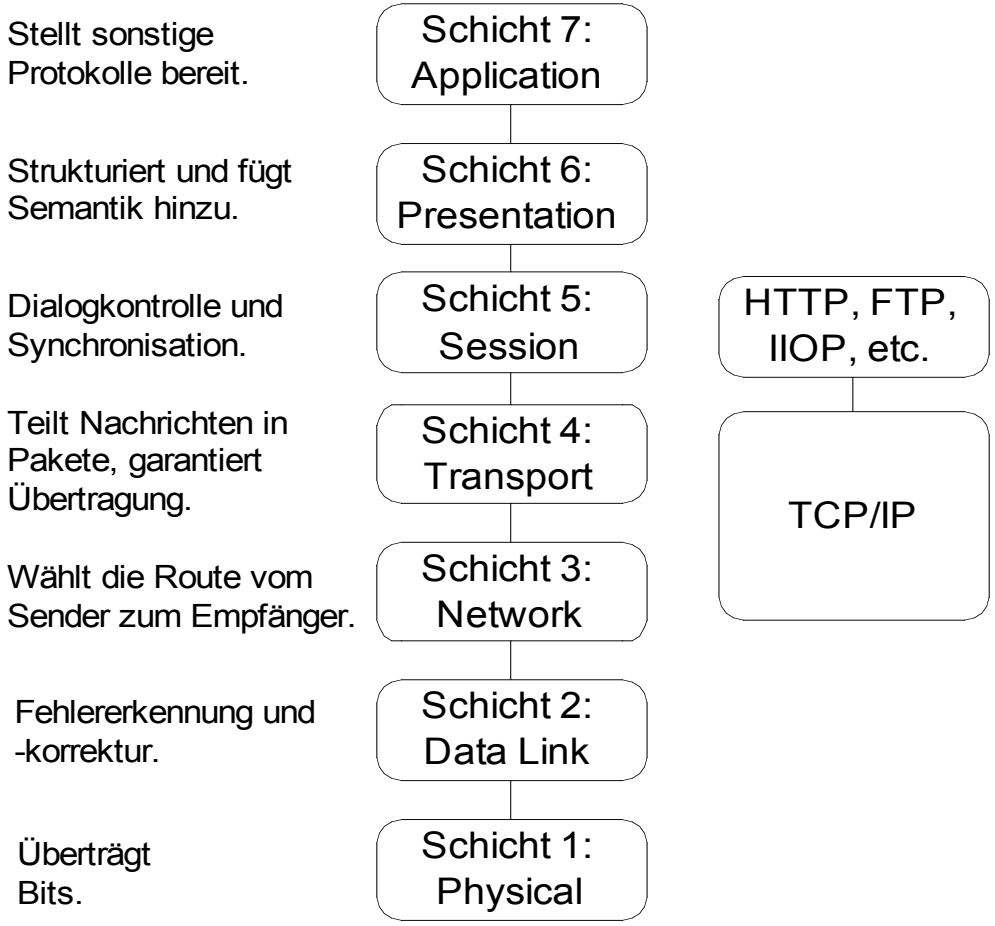
Model-View-Controller



Architekturmuster: Schichten

- Siehe auch: Hierarchische Zerlegung (in „benutzt“-Beziehungen)
- Untere Schicht bildet reine Blackbox für höhere Schicht





Schichten <> Layered Architecture

UserInterface:

- Informationsanzeige
- Interpretation der Benutzerkommandos

Application:

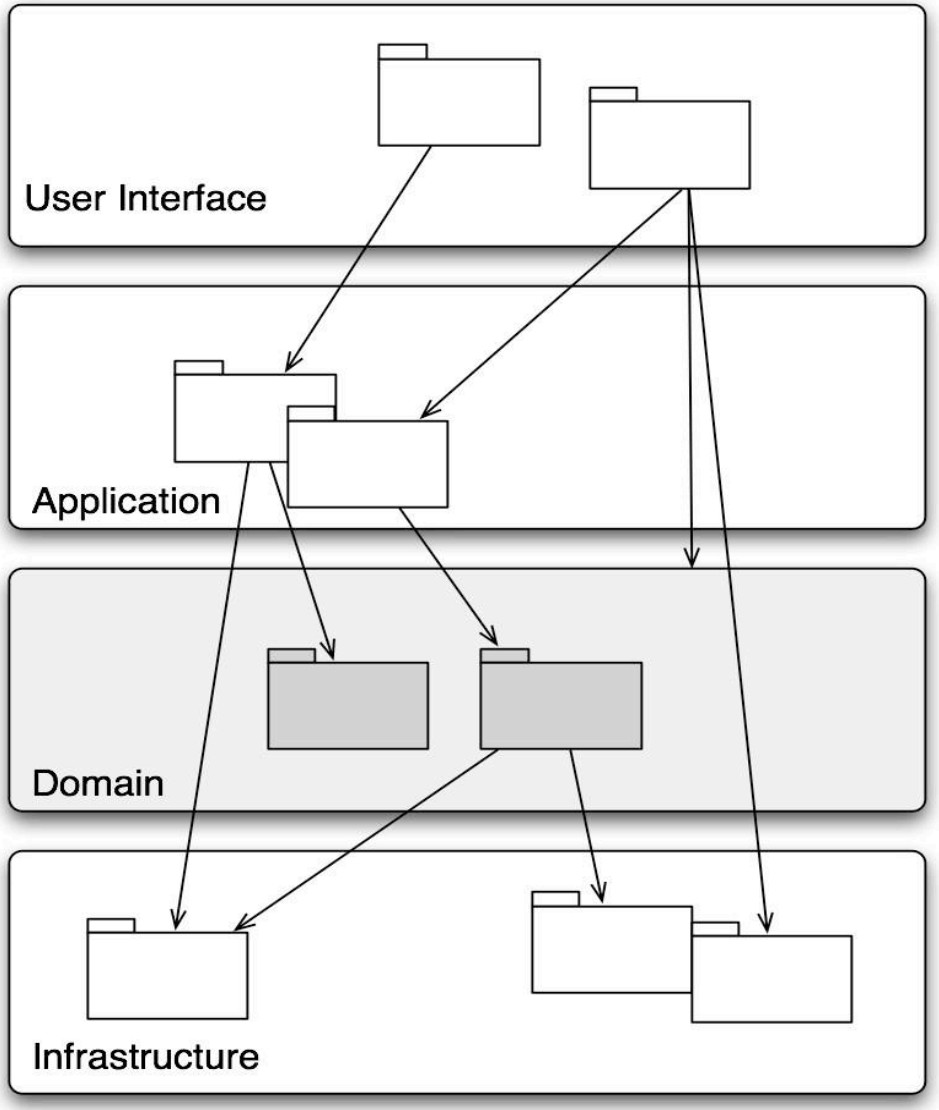
- Steuert Geschäftsprozesse
- Koordiniert Domain-Objekte
- Kann Zustand von Benutzer oder Anwendung verwalten

Domain:

- Repräsentiert die Konzepte der Problemdomäne
- Verwaltet Zustand der Domäne
- Persistenz delegiert an Infrastruktur

Infrastructure:

- Allgemeine technische Services



Hier nach [Evans]
„Domain-Driven-Design“

Schichten haben Vor- und Nachteile

- Vorteile:
  - Geringe / keine Abhängigkeiten bei Erstellung und Betrieb
  - 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.

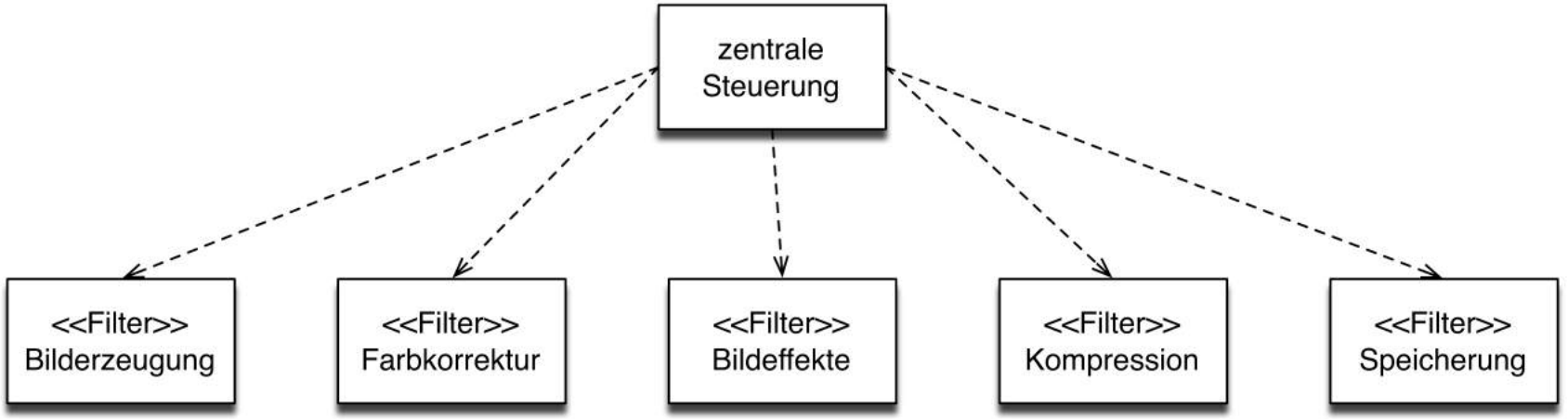
Architekturstil „Pipes and Filters“

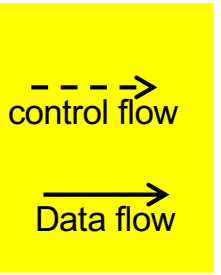
- Bekannt aus Unix-Kommandozeilen:
  - „|“ repräsentiert eine Pipe
  - Beispiel (aus Cygwin-Bash:)

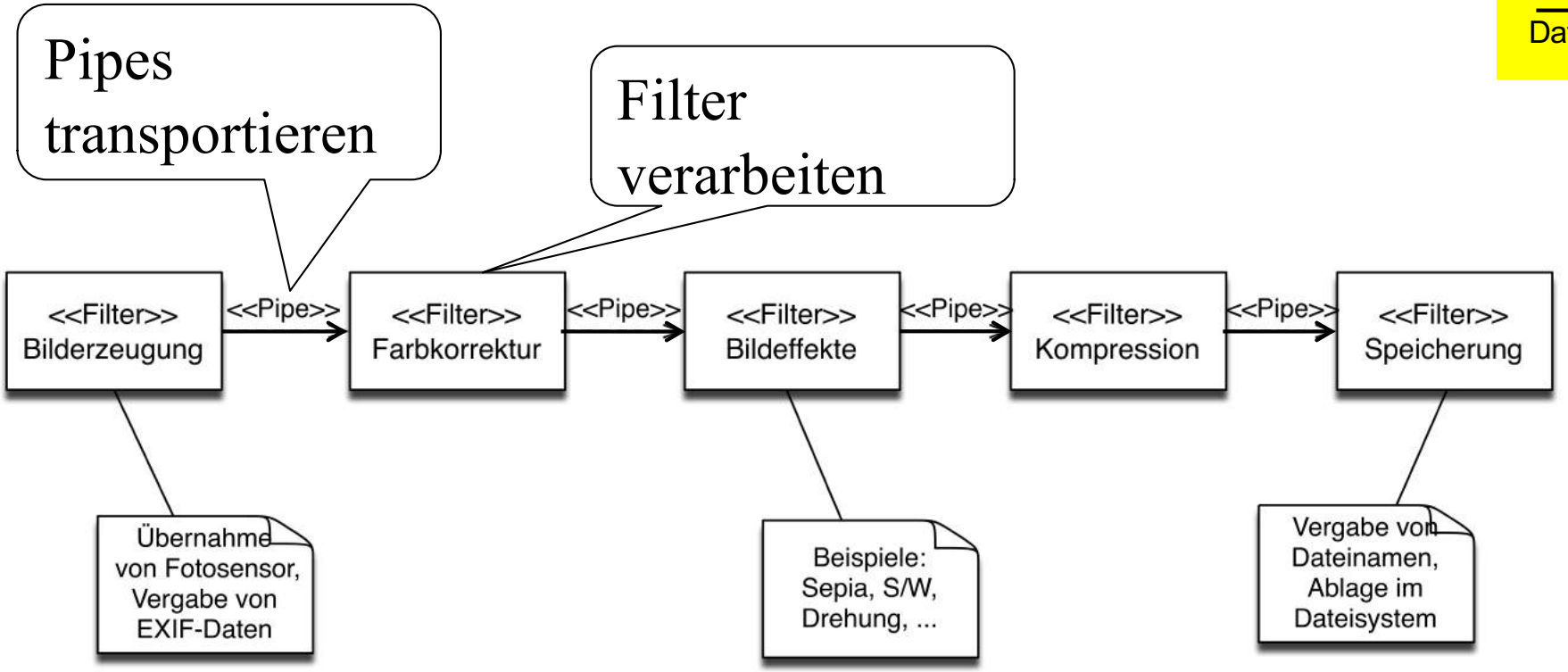
```
$ cat maven.log | sort -u | wc
```

- Filter
  - Implementiert einen Verarbeitungsschritt auf seinen Eingabedaten
    - Push filter
    - Pull filter
    - Push + Pull
- Pipe
  - Transportiert und puffert Daten

Darstellungsalternativen von Pipes und Filter







Bewertung: Pipes und Filter

Kontext & Einsatzbereich

- Individuelle, eindeutig unterscheidbare Verarbeitungsschritte
  - Beispiel: Datentransformation

Voraussetzungen

- Daten sind in getrennten Paketen (“Chunks”) verarbeitbar
  - Keine Abhängigkeiten zwischen verschiedenen Chunks
  - Beliebige Reihenfolge der Bearbeitung von Chunks

Vorteile

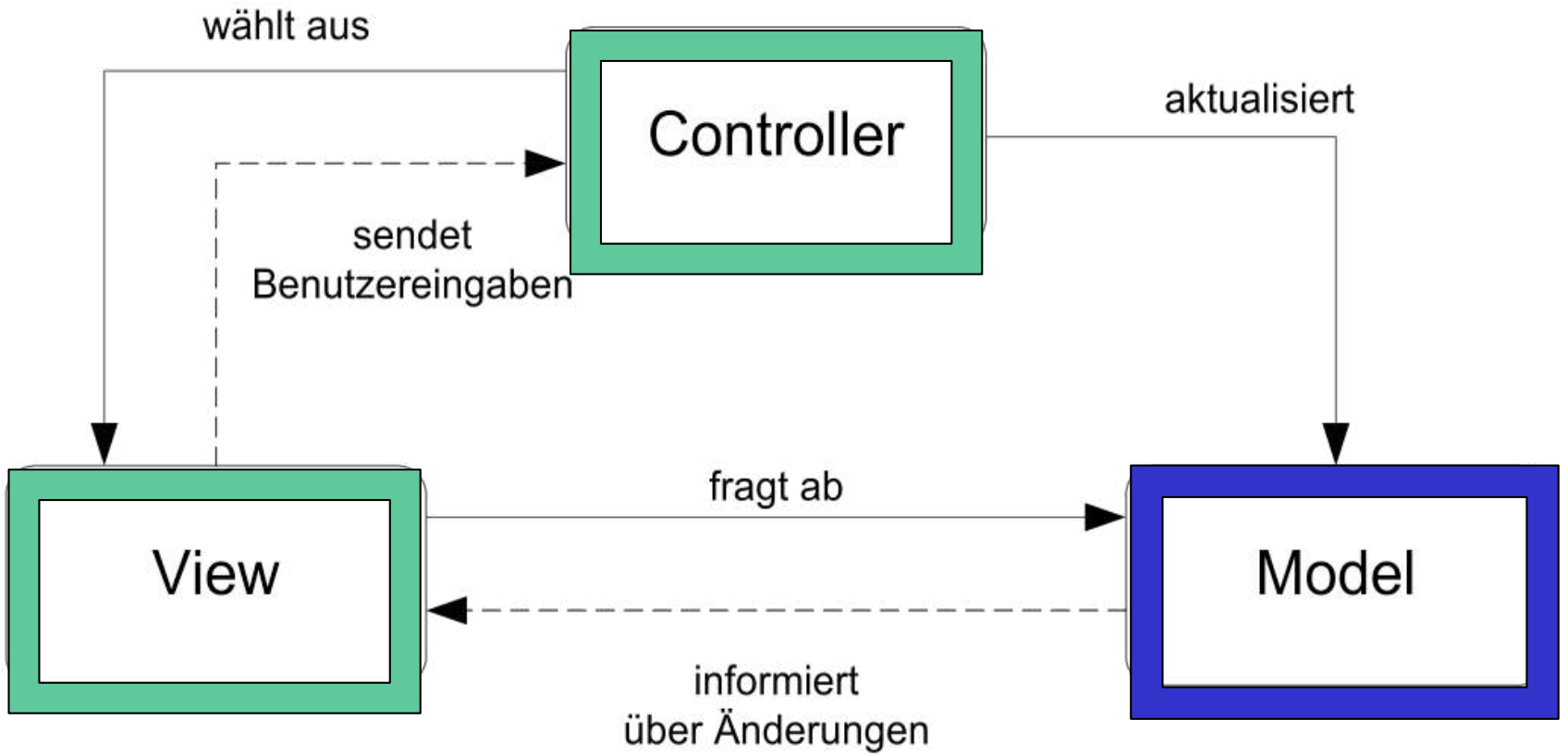
  - Einfache Struktur
  - Modular

- Nachteile
  - Fehlerbehandlung schwer (Unix: stderr)

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



Wo gibt‘s mehr Informationen…

- Literatur
  - Gamma et. al: Design-Patterns, *der Klassiker*…
  - POSA-Serie bei Addison-Wesley, u.a.:
    - Architecture Patterns
    - Patterns for Networked and Concurrent Objects
    - Patterns for Resource Management
- Websites
  - www.hillside.net
  - www.patterns-kompakt.de
  - Quelle: http://msdn.microsoft.com/architecture/patterns/

- Konferenzen
  - Verzeichnis einiger Konferenzen auf ww.hillside.net
  - PloP, EuroPloP, ChiliPloP, KoalaPloP, VikingPloP

Wo in arc42 kommen Ergebnisse hin

1. Einführung und Ziele
1.1 Aufgabenstellung
1.2 Qualitätsziele
1.3 Stakeholder

2. Randbedingungen
2.1 Technische Randbedingungen
2.2 Organisatorische Randbedingungen
2.3 Konventionen

3. Kontextabgrenzung
3.1 Fachlicher Kontext
3.2 Technischer- oder Verteilungskontext

4. Lösungsstrategie

5. Bausteinsicht
5.1 Ebene 1
5.2 Ebene 2
….

6. Laufzeitsicht
6.1 Laufzeitszenario 1
6.2 Laufzeitszenario 2
….

7.Verteilungssicht
7.1 Infrastruktur Ebene 1
7.2 Infrastruktur Ebene 2
….

8. Konzepte
8.1 Fachliche Struktur und Modelle
8.2 Typische Muster und Strukturen
8.3 Persistenz
8.4 Benutzeroberfläche
….

9. Entwurfsentscheidungen
9.1 Entwurfsentscheidung 1
9.2 Entwurfsentscheidung 2
….

10. Qualitätsszenarien
10.1 Qualitätsbaum
10.2 Qualitäts-/Bewertungsszenarien

11. Risiken

12. Glossar

Zusammenfassung

Es gibt keinen deterministischen Prozess zur Konstruktion komplexer Systeme

Hilfreiche Methodiken:

- Entwickeln einer ersten Systemidee („Ideen für Technik“)
- Entwickeln von Lösungsstrategie (grundsätzliche Lösungsideen)
- Von Anfang an auf Qualitätsmerkmale konzentrieren
- Domain-Driven: Fachliche Architekturen zuerst entwerfen
- Top-Down und Bottom-Up
- Architekturmuster kennen und anwenden

- Iterativ und evolutionär arbeiten
  - Risiken und Zielerreichung kontinuierlich prüfen
  - Durchgehend (wesentliche) Entscheidungen festhalten