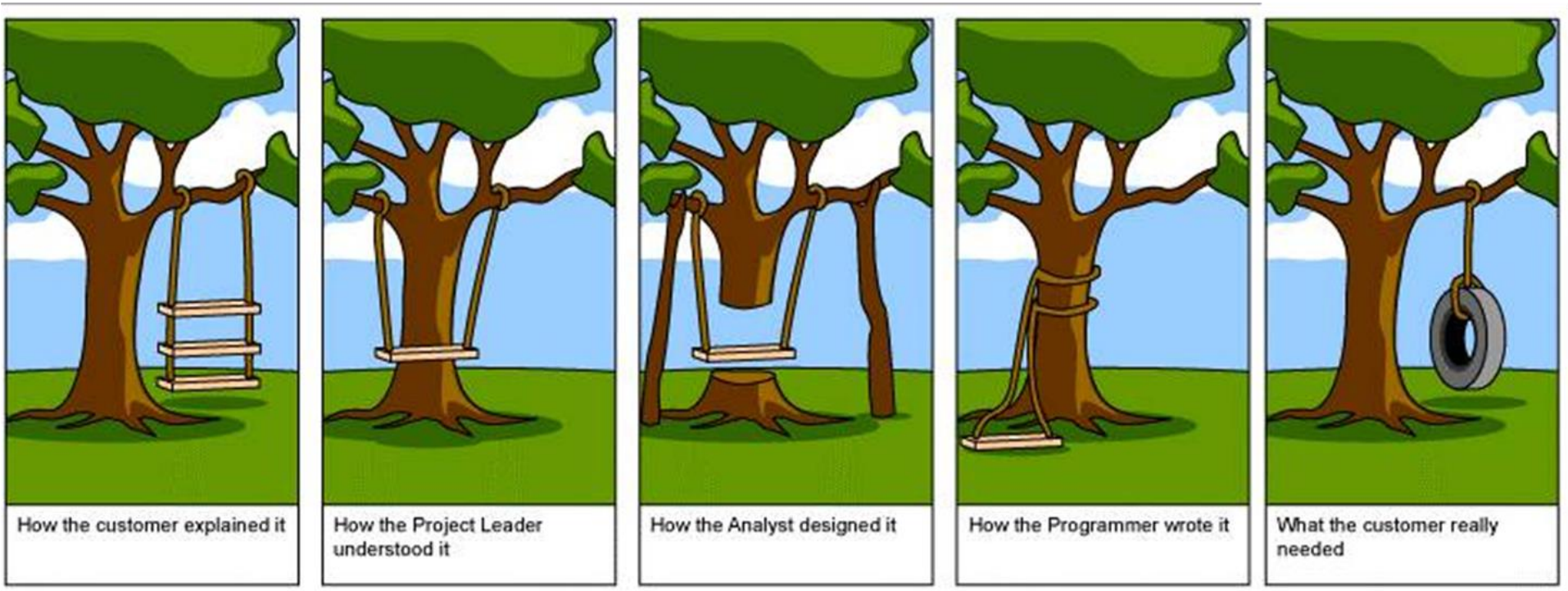
Das Design

Phase III

Programmieren ist das Wichtigste – oder doch nicht?

- Schaffen ist das Eine, **desirable products** schaffen das Andere (siehe iPhone und Co)
- Die Klassiker seit Generationen:



Design Phase

Transforms detailed requirements into complete, detailed Systems Design Document. Focuses on how to deliver the requirements document.

- Vorweg (und Wiederholung): Wie viel Arbeit sollten wir in das Design stecken?
- 3 gängige Möglichkeiten:
  - Architecture-indifferent design: Man schenkt der Architektur wenig Aufmerksamkeit. Eventuell Vorlagen aus Normen oder anderen Domänen.
  - Architecture-focused design: Es wird eine Architektur entworfen, welche den Anforderungen gerecht wird.
  - Architecture hoisting: Hier wird die Architektur so gestaltet, dass die Anforderungen garantiert eingehalten werden.

Welche Möglichkeit ist die Richtige?

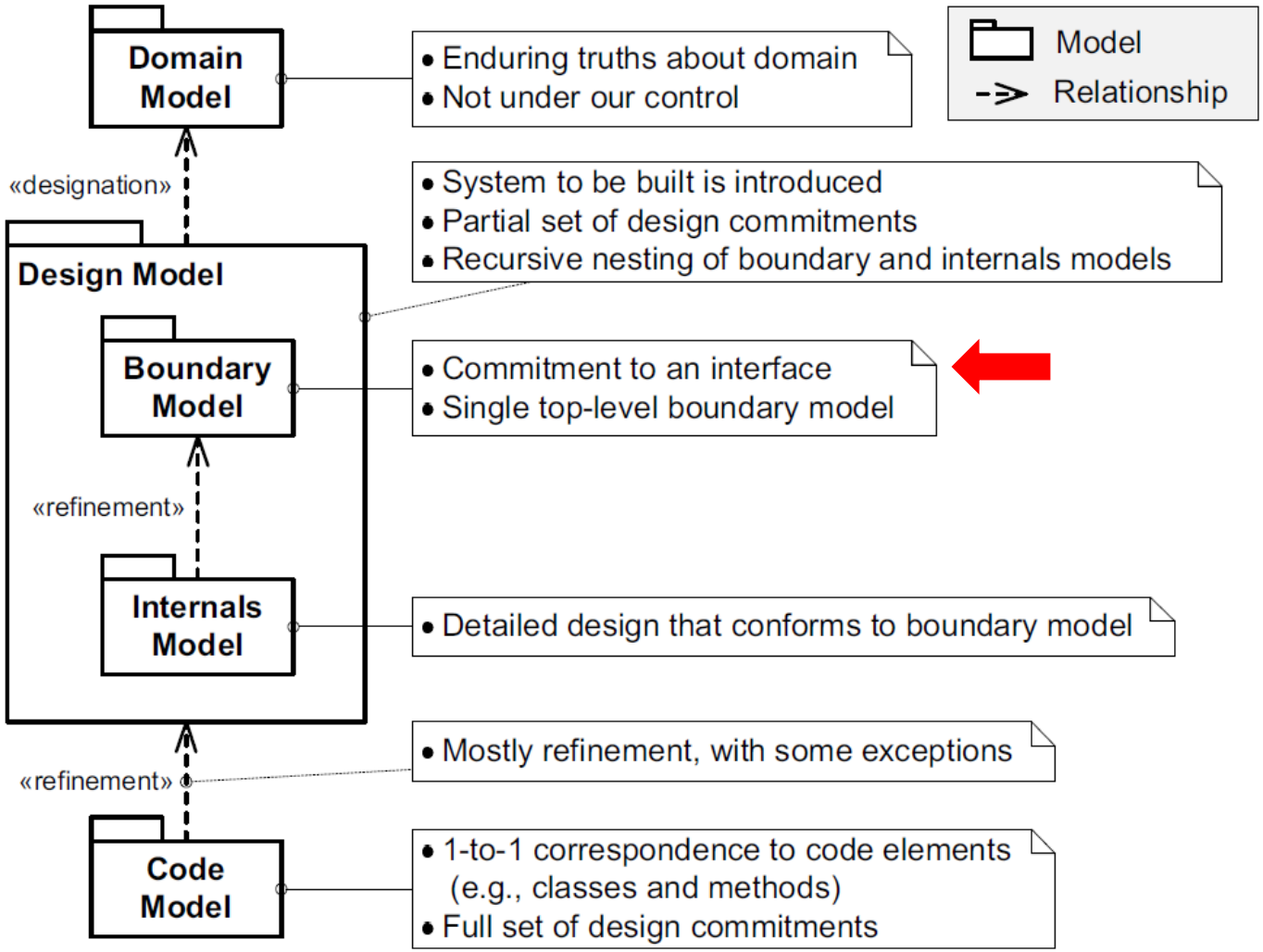
- Alle – und auch keine
- Sollte man sich für eine Möglichkeit für JEDES Projekt entscheiden? Nein
- Also: Risiko ist hier wieder unser Indikator der Wahl
  - Risiko = Aufwand in der Design-Phase
- Wir wollen das Risiko am Ende des Tage reduzieren
  - Money, Ressources, …

Lass uns modellieren – aber was wollen wir eigentlich modellieren?

Die Domäne und das Design

- Es empfiehlt sich die Domäne (die Realität) und das Design (die Computer-Repräsentation) zu trennen
  - Die Analyse der Domäne kann wichtige Fragen und Unklarheiten beseitigen, da sie den technischen „Noise“ nicht beinhaltet und daher einfach(er) ist
  - Das Design wird im Regelfall der Domäne ähnlich sein, aber es wird Ausnahmen und Einschränkungen geben
  - Wir konzentrieren uns auf das Design

Die Domäne und das Design cont.



Tipp: Man muss das nicht bei jedem Projekt in diesem Detailgrad machen (Stichwort: Risiko)

Fairbanks, G. (2010). Just enough software architecture: a risk-driven approach. Marshall & Brainerd.

Erklärung

- Designation: Ähnliche Dinge in unterschiedlichen Modellen bedeuten was ähnliches
- Refinement: Ein Detailgrad höher
- Boundary Model: Was andere im System sehen (e.g. behavior, interchange data, und quality attributes)
- Internal Model: Boundary Model + mehr Details (e.g. Szenarien)

FH University of Applied Sciences TECHNIKUM WIEN

Domain-Driven-Design

- Vereinfacht gesagt: Versucht, die Realität im Code zu modellieren (in Fairbanks-Modell: Domain-Model → Code-Model)
- Tools*:
  - Bounded context: *„Explicitly define the context within which a model applies“*. E.g. das Wort „Kunde“ kann innerhalb eines Projektes unterschiedlich in Systemen verstanden werden.
  - Eine Ubiquitous Language (eine einheitliche Verwendung von Begriffen) sollte dabei helfen, dass alle das gleiche Auto sehen, wenn man von einem Auto im Projekt spricht
  - Tools im Code: Value-Object, Aggregate, Domain-Event, uvm.

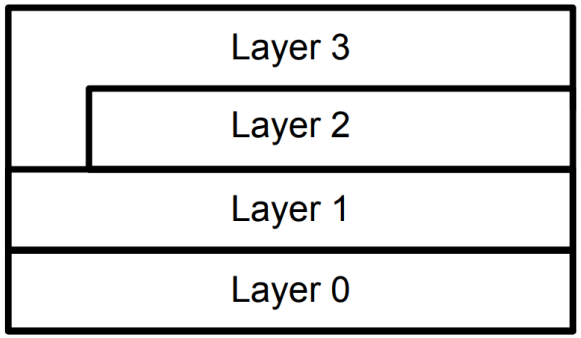
* https://en.wikipedia.org/wiki/Domain-driven_design

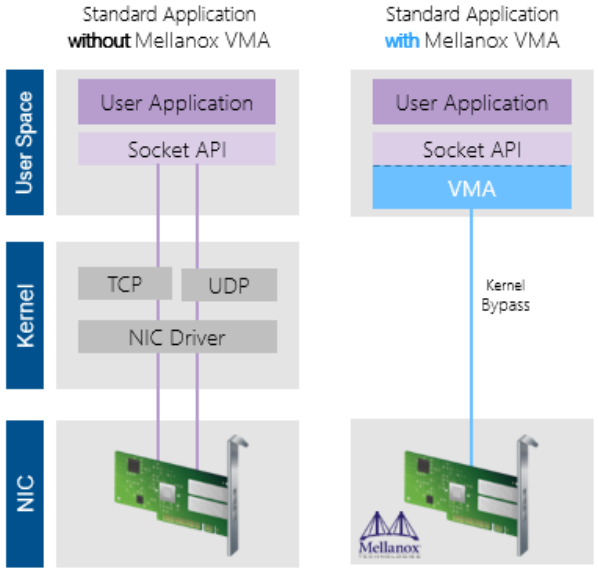
Design ist ein großes Kapitel …

- Da die Vorlesung sich auf Komponenten konzentriert, werden wir uns nur die wichtigsten **Architecture-Styles** anschauen
- Es gibt auch eine Unterscheidung zwischen Pattern und Style – aber für die folgenden Ideen unwichtig
  - Daher auch einige Punkte die auch eher zu Patterns gehören
- Architecture-Styles helfen
  - Architekturen zu kategorisieren
  - Gemeinsame Eigenschaften zu definieren
  - Um die Architekturen anschließend vergleichen zu können
- Komponenten spielen in den folgenden Aufzählungen eine essentielle Rolle (Zur Erinnerung: Interfaces, Ports, Connectors)

Layered style

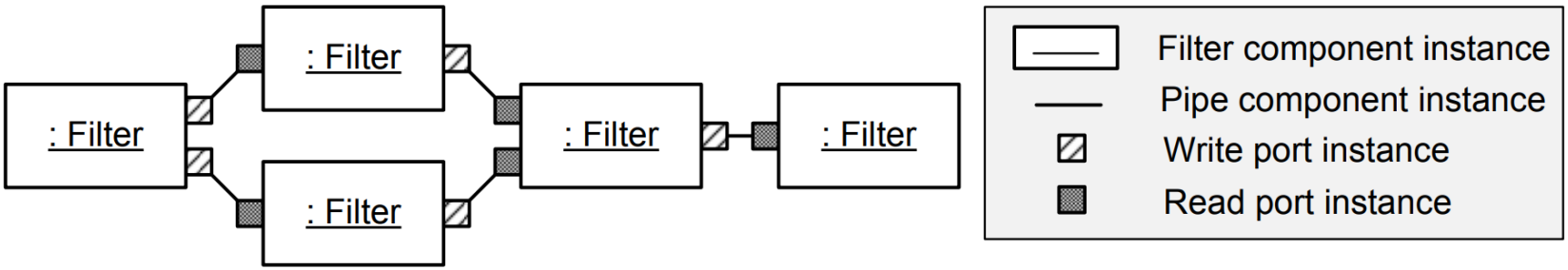
- Layer höherer Ebene können mit Layer tieferer Ebene kommunizieren
- In den Layern befinden sich e.g. Services in Form von Komponenten
- Konzentriert sich auf die Qualitätsattribute Modifiability, Portability und Reusability
- Layer tieferer Ebene können leicht ersetzt werden oder emuliert werden
- Layers können unter gewissen Umständen auch einen By-Bass machen
- Beispiel: Betriebssystem mit Network Kernel-By-Pass





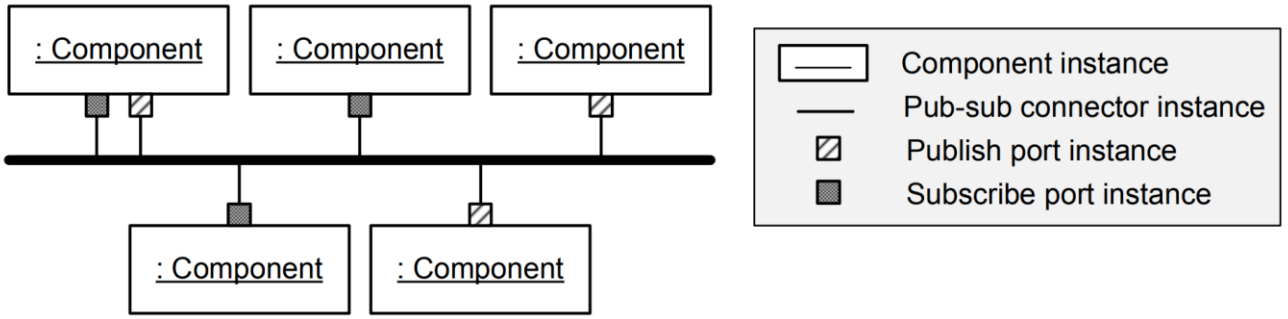
Pipe-and-filter style

- Daten werden „strommäßig“, kontinuierlich bearbeitet
- Elemente:
  - Pipe: Transportieren die Daten in den nächsten Filter (ohne sie dabei zu ändern)
  - Filter: E.g. reichert Daten an oder transformiert sie. Filter operieren voneinander unabhängig und teilen sich auch keinen State. Vorteil: Filter können wiederverwendet werden
- Vorteile: Das System ist sehr flexibel



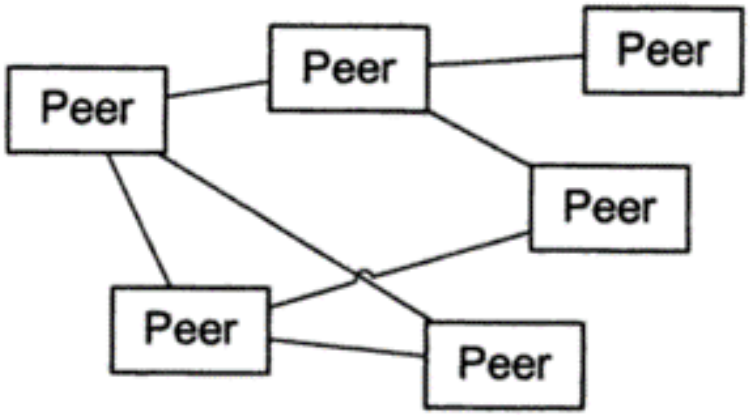
Publish-subscribe style

- Publisher-Komponenten senden „Events“ (e.g. `CustomerCreated`)
- Subscriber-Komponenten können sich auf diese Events subskribieren (=Interesse zeigen)
- Vorteil:
  - Die Komplexität, dass Events auch bei Subscribern ankommen, kann ausgelagert werden
  - Publisher und Subscriber sind vollkommen entkoppelt und voneinander unabhängig



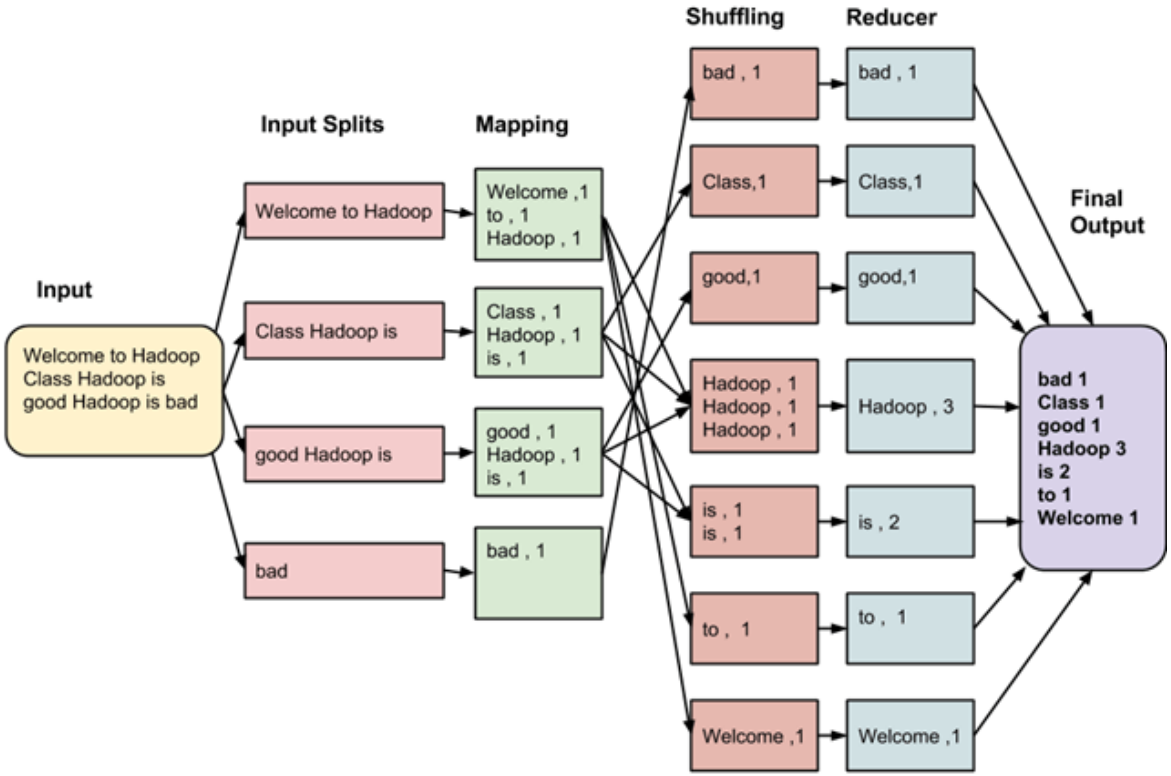
Peer-to-peer style

- Komponenten können als Client und/oder Server agieren
- Im Kontrast zu Client-Server kein Single-Point-Of-Failure, da alle Komponenten gleichberechtigt
- Knoten können sich wie im realen Leben finden: „Der kennt den, und der kennte den“



Map-reduce style

- Für große Datenmengen (e.g. Suchmaschinen)
- Berechnungen / Aufgaben können auf mehrere Maschinen verteilt (map) werden – das Ergebnis der Einzelschritte, wird dann wieder eingesammelt (reduce)



FH University of Applied Sciences TECHNIKUM WIEN

Wie können meine Komponenten miteinander kommunizieren? Connectors & Styles

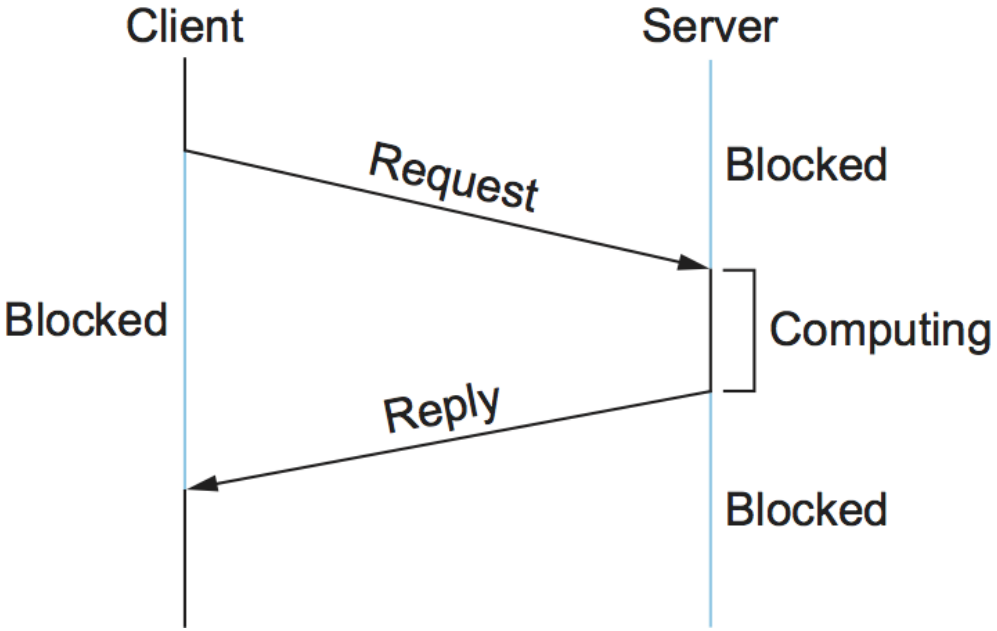
Remote-Procedure-Call (RPC)

- *„**Synchronous** Remote Procedure Calls (RPCs). RPCs enable a client to invoke the services of a remote server as if they were implemented by local procedure calls.”*
- **Vorteil**: Programm bleibt „prozedural“ – d.h. Details über die Kommunikation mit der anderen Komponenten bleiben verborgen (Abstraktion), Optimierung (siehe WCF IPC vs. TPC per Konfiguration), stört Refactoring wenig
- **Nachteile**: Erwartungen eventuell „versteckt“ (Netzwerkaufruf kann schief gehen), wenn schlecht implementiert „blocking“ (siehe async / await)

Schmidt, Douglas, Stal, M., Rohnert, H., & Buschmann, F. (1996).
Pattern-oriented software architecture, volume 1: a system of patterns.

Remote-Procedure-Call (RPC)

- Was passiert bei einem Timeout auf Applikationsebene?
  - TCP Keep-Alive ist so ne Sache …



https://blog.stephencleary.com/2009/05/detection-of-half-open-dropped.html
https://250bpm.com/blog:22/

REST (Representational State Transfer)

- Wichtigsten Eigenschaften:
  - Client-Server: User-Interface wird vom Storage getrennt
  - Stateless: Die Request muss die vollständige Information enthalten – States werden nur im Client gehalten. Vorteil: U.a. kann leicht skaliert / gecached werden
- Kernabstraktion von REST ist eine Ressource (e.g. ein Dokument, ein Bild, eine Person, das aktuelle Wetter in Wien, …)
- Oft verwendet mit HTTP – aber ist kein Muss

REST über HTTP: Verben

- Safe: beim Aufruf sind keine Seiteneffekte („Änderungen“) zu erwarten
- Idempotent: Es macht keinen Unterschied, ob ich den Request 1 oder 1000 Mal aufrufe – das Ergebnis bleibt das gleiche
  - Ebenfalls im Payload wichtig: `increment value by 1` vs. `store value 238`

Verb	Safe	Idempotent
GET	Yes	Yes
HEAD	Yes	Yes
OPTIONS	Yes	Yes
PUT	No	Yes
DELETE	No	Yes
POST	No	No

REST Beispiele: Anlegen eines Benutzers

- URL: `POST /api/v1/users`
- Content:


```
{
   "firstname":"Karl",
   "secondname":"Maier",
   "address":{
      "postalcode":1010,
      "street":"SomeStreet 12"
   },
   "dayofbirth":"1984-01-01"
}
```


- Return HTTP Status Code: `202 Created`
- Content: Id + Referer zur Ressource (`/api/v1/users/273`)

Weitere REST Beispiele

- `PUT /api/v1/users/273` um den User zu aktualisieren
- `GET /api/v1/users/273` um den User zu abzufragen
- `DELETE /api/v1/users/273` um den User zu löschen
- `PATCH`: Update wenn große Entität und man nur einen Teil updaten will
  - RFC 6902 http://jsonpatch.com/
  - RFC 5789 https://tools.ietf.org/html/rfc5789

REST-over-HTTP via JSON immer sinnvoll?

- Auf die Constraints achten
  - Macht e.g. stateless in meinem Context Sinn?
  - Macht es Sinn, Ressourcen-orientiert zu denken?
- REST-over-HTTP via JSON oft auch für Public-APIs (e.g. https://developer.github.com/v3/, https://developer.twitter.com/en/docs) → muss aber nicht Sinn machen für interne Kommunikation (siehe e.g. https://grpc.io/ - was viel performanter ist)

RPC & REST

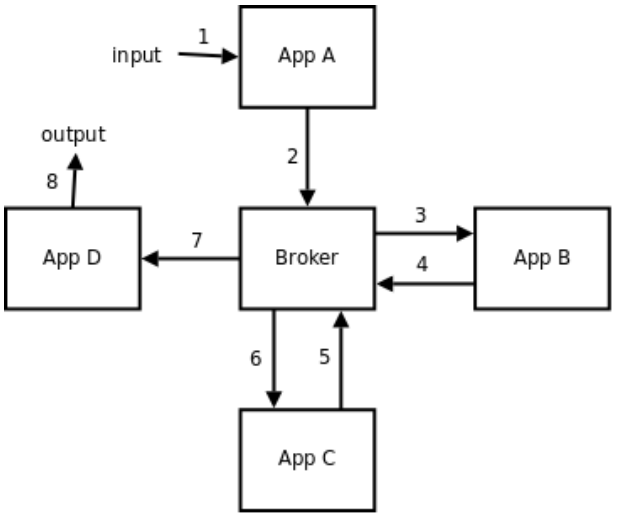
- Wichtige Gemeinsamkeit: **zeitliche Abhängigkeit**
- Die andere Komponente muss beim Aufruf verfügbar sein
- Macht in vielen Situationen Sinn, weil ich die Antwort sofort brauche
- Was aber, wenn ich die Anwendung nicht sofort brauche? E.g. die Bestellung bearbeiten, die Durchschnittstemperatur aus den Werten der letzten 24h berechnen?
- Broker-Style ist eine Möglichkeit

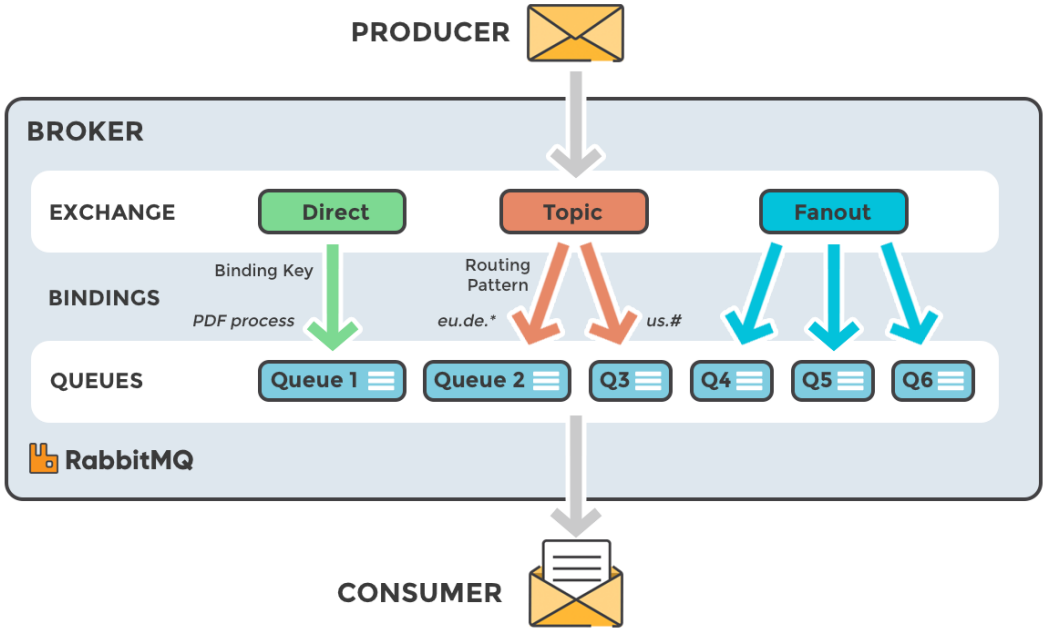
REST – aus der Praxis

- Beispielentität: Person mit Adressen (Lieferadresse, Rechnungsadresse)
- Es wird über die API folgendes geändert:
  - Person Nachname
  - Lieferadresse
- Aber was ist passiert?
  - Hat die Person geheiratet?
  - Handelt es sich um eine andere Person mit gleichem Vornamen?
  - Ist die Person umgezogen?
  - Wurde der Account übernommen?
- Zu beachten: der Kontext geht oft verloren → im API Design berücksichtigen

Broker-Style

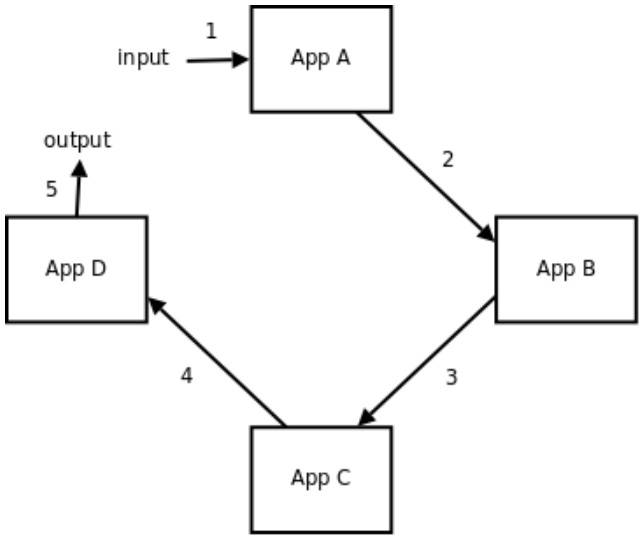
- Broker agiert als eine Art Vermittler
  - Dadurch Entkoppelung
- Oft in Verwendung mit dem Messaging-Style
  - Beispiel RabbitMQ





Brokerless-Style

- Broker ist „Nadelöhr“ (Skalierung & Verfügbarkeit erhöhen möglich)
  - Applikationen müssen sich kennen / finden
- Aber es geht auch ohne Broker
- Beispiel zeroMQ, MSMQ



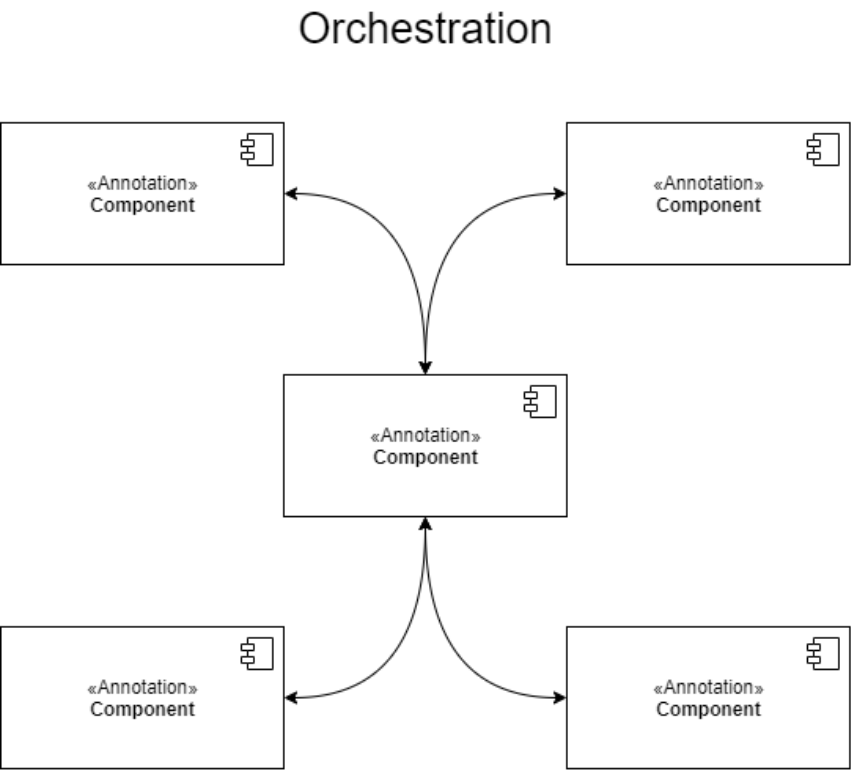
Wie kann ich meine Services untereinander koordinieren?

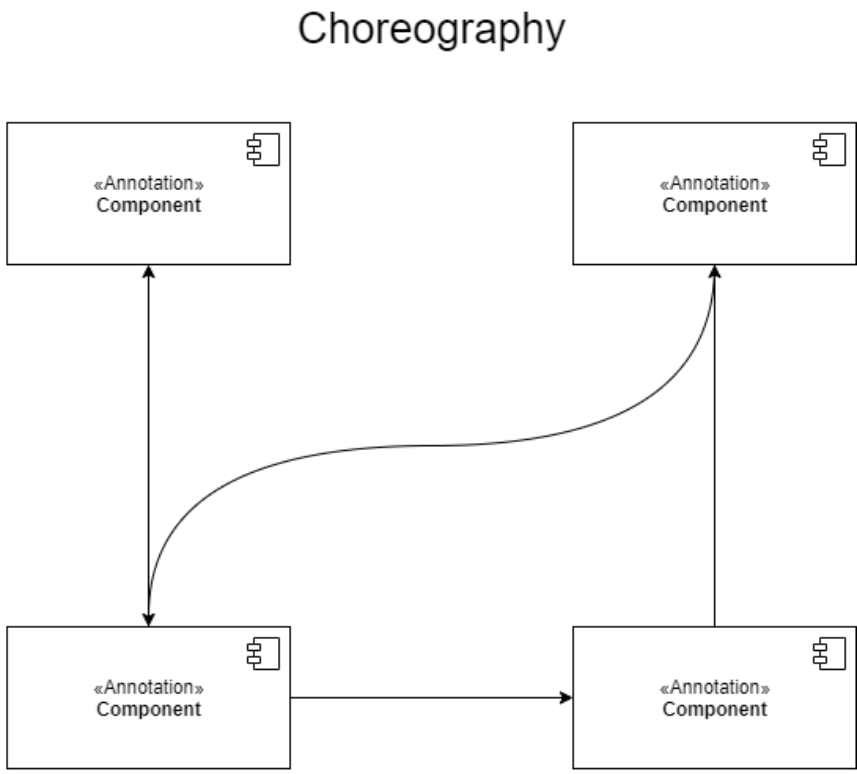
Problemstellung

- 3 Services
  - Mietwagen buchen: Erlaubt es Mietwagen bei diversen Anbietern (AVIS, Sixt, Europcar) zu bestellen
  - Flug buchen: Erlaubt die Buchung bei diversen Fluggesellschaften
  - Hotel buchen: Kann Hotels bei großen Ketten wie IBIS usw. buchen
- Es kommt eine Anfrage mit folgenden Informationen:
  - Mietwagen: von 01.01.2020 – 10.01.2020
  - Flug: Hin 01.01.2020, Retour 10.01.2020
  - Hotel: 01.01.2020 – 10.01.2020
- Wie koordinieren sich die Service untereinander?
  - E.g. es gibt keinen Flug mehr für diese Zeit, Hotel aber Auto schon

Exkurs: Orechestration vs. Choreography

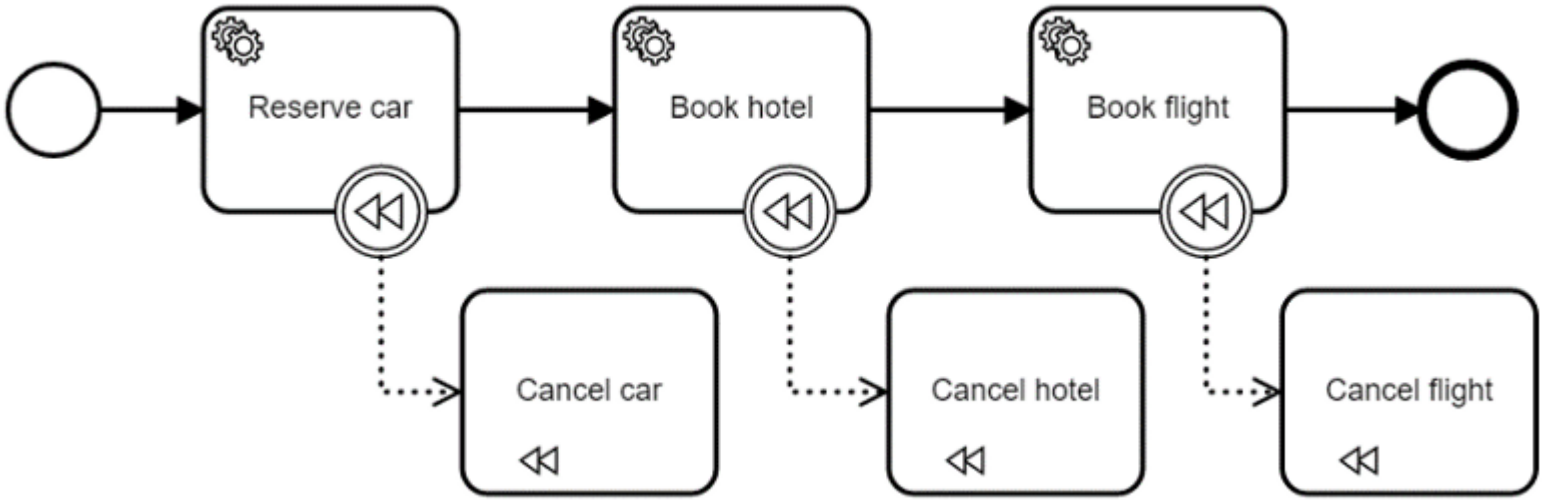
- Hier geht es eher um Prozesse – trotzdem ähnlich





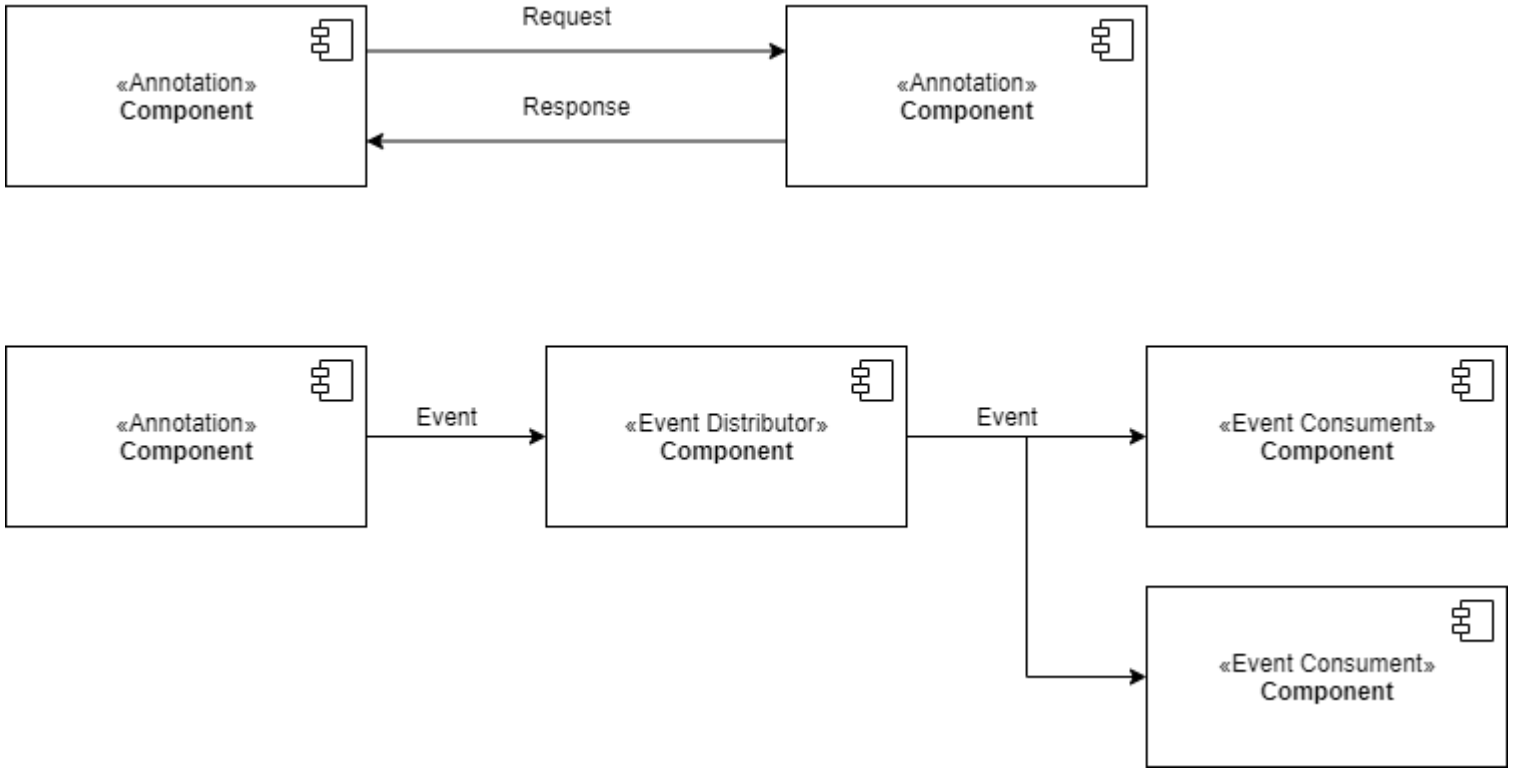
Beispiel: Cammunda BPMN

- SAGA Style: kompensierende Operationen wurde angegeben
  - Wenn das Auto reserviert wurde → Storno



Event-Driven-Style

- Auch hier kennt die Komponente seine Empfänger (im Regelfall) nicht



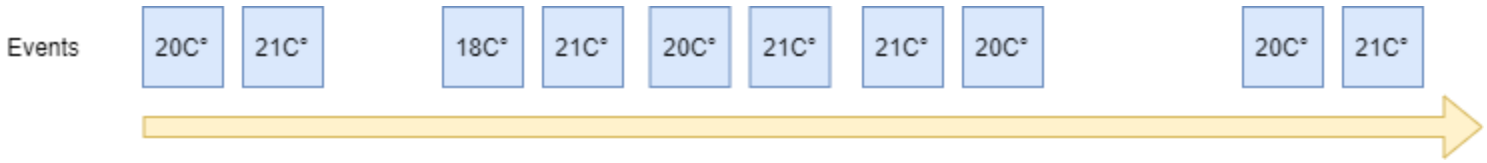
Was ist ein Event?

- *“An event is an occurrence within a particular system or domain; it is **something that has happened**, or is contemplated as having happened in that domain. The word event is also used to mean a programming entity that represents such an occurrence in a computing system”*
- Was kann man mit Events machen?
  - Beobachten: Der Kunde kauft schon zum 3. Mal Produkt X – Dauerauftrag vorschlagen?
  - Reagieren: Ich habe Event X bekommen – muss etwas tun – und Event Y auslösen
  - Fehlerdiagnostik: Wie ist es dazu gekommen, dass der Wert in der Komponente auf einmal negativ ist?!
  - Events vorhersagen: Der Kurs steigt?
  - …

Etzion, O., & Niblett, P. (2011). Event processing in action. Manning.

Events Zeitachse

- Da Events zu einem gewissen Zeitpunkt passieren entsteht ein Stream aus Events
- Auf Events können „Rechenoperationen“ ausgeführt werden
- Oder es kann auf gewisse Events reagiert werden (oder wenn ein gewisses Muster passiert e.g. 20C° → 21C° → 20C°)



FH University of Applied Sciences TECHNIKUM WIEN

Komponenten gruppieren

Microservice

- *„The high-level definition of microservice architecture (microservices) is an **architectural style** that functionally **decomposes** an application into a set of services. Note that this definition doesn't say anything about size. Instead, what matters is that each service has a **focused, cohesive set of responsibilities**."*
- Daher:
  - Ein Microservices hat auch wohl definierte Schnittstellen
  - Ein Microservices kann mehrere Komponenten beinhalten
  - Eine Komponente kann in mehreren Microservices vorkommen

Richardson, C. Microservices Patterns With examples in Java (2018). URL: https://www. manning. com/books/microservices-patterns.

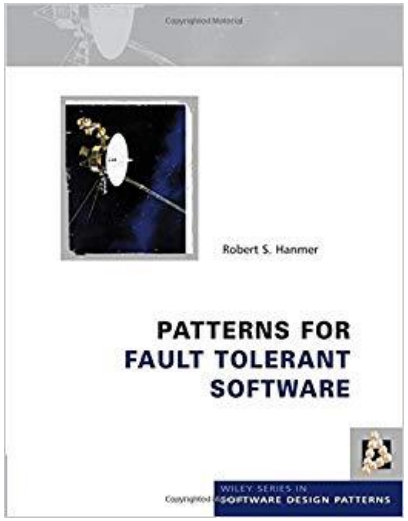
Weitere Eigenschaften

- Independent deployability: Einzelne Funktionalitäten können geändert werden (e.g. Billing) ohne dass andere Microservices auch deployed werden müssen (e.g. OrderManagament)
- Keine Shared Databases: Im Regelfall halten Microservices ihre eigenen Daten in einem eigenen Storage.
  - Über Schnittstellen → Datenaustausch
- Microservice gehört einem Team
- Viele Patterns im Dunstkreis der Microservices: https://microservices.io/i/MicroservicePatternLanguage.pdf

Komponentenzuverlässigkeit erhöhen

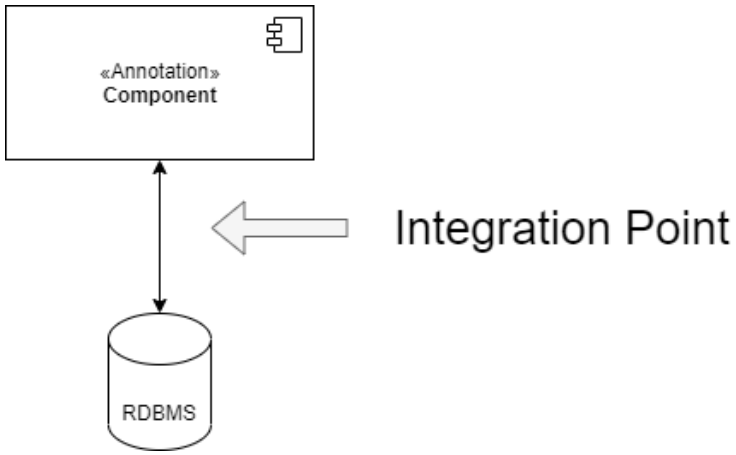
Software Fault-Tolerant designen

- *„Fault tolerance is the property that enables a system to continue operating properly in the event of the failure of (or one or more faults within) some of its components.”*
- Schritte: Error Detection → Error Recovery → Error Mitigation → Error Treatment



Aufpassen: Integration Points

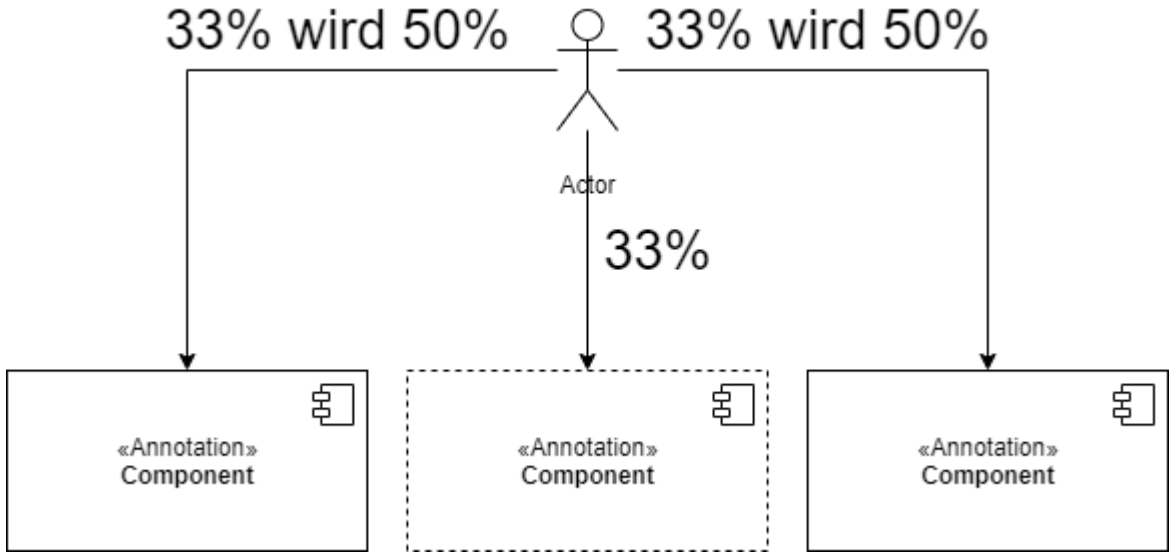
- Integration-Points sollten immer kritisch betrachtet werden → können Versagen
- Fehler von Netzwerk-Fehler bis Semantische-Fehler
- Fehlerausbreitung vermeiden („Defensives Programmieren“)
- Patterns: Circuit Breaker, Timeouts, Decoupling Middleware, Handshaking



Mehr dazu später

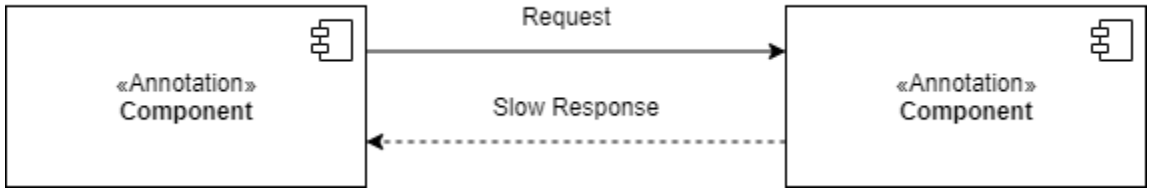
Aufpassen: Chain-Reactions

- 3 Komponenten – je 33% Last. 1 Server fällt aus – andere Komponenten sollten Arbeit übernehmen
  - Gefahr groß, dass diese auch ausfallen
  - Circuit Breaker … Lieber 33% User weniger bedienen



Aufpassen: Slow-Responses

- Komponente antwortet langsam …
- Meist Indikator für „zu wenig im Vorfeld überlegt“
  - Meist scheitert es an Ressourcen
- Patterns „Someone in Charge“, „Fault Observer“, „Maintance Interface“



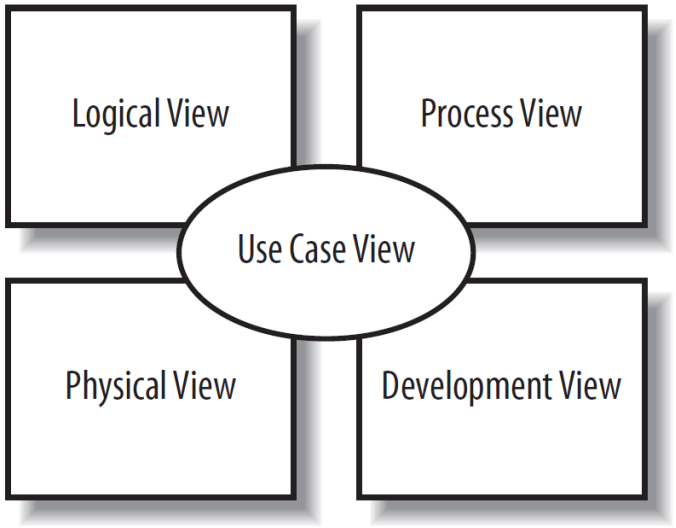
Weitere wichtige Themen

Wie lange sollte man nach einer Lösung suchen?

- Ändert man e.g. ein Quality-Attribute an einem Ende, so kann ein Eigenschaft des Systems an einem anderen Ende beeinflusst werden
  - Wenn man e.g. an der Security Schraube dreht, kann sich die Usability verschlechtern
- Es gilt nicht die beste Lösung zu finden (sprich: so lange die Schrauben zu drehen, bis alles optimal ist)
  - Wir reden immer von einer **acceptable solution**
  - Und nicht von einer optimal / best solution

Komponenten dokumentieren

- UML 2 Nicht Hauptaugenmerk dieser Vorlesung
- Man sollte es aber einmal gehört haben: Kruchten’s 4+1 view model



Miles, R., & Hamilton, K. (2006). Learning UML 2.0. " O'Reilly Media, Inc.".

Die Views

- Logical View: Funktionalität des Systems
  - Klassendiagramme, Kommunikationsdiagramm, Sequenzdiagramm
- Development View: Beschreibt System aus Sicht des Entwicklers
  - Komponentendiagramm, Paketdiagramm
- Process View: Prozesse im System hinsichtlich Laufzeitverhalten
  - Aktivitätsdiagramm
- Physical View: aus Sicht des Systemarchitekten (Verteilung der Softwarekomponenten)
  - Verteilungsdiagramm (Deployment diagram)

Architectural Decision Records

- Context: In this section of the ADR we will add a short one- or two-sentence description of the problem, and list the alternative solutions.
- Decision: In this section we will state the architecture decision and provide a detailed justification of the decision.
- Consequences: In this section of the ADR we will describe any consequences after the decision is applied, and also discuss the trad
- https://cognitect.com/blog/2011/11/15/documenting-architecture-decisions