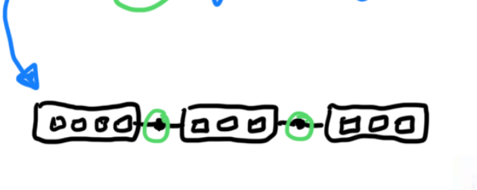
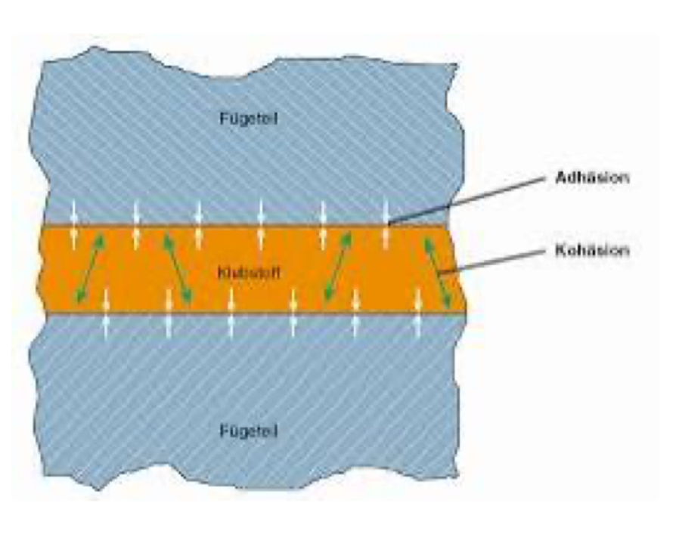
1. KOPPLUNG UND KOHÄSION

Was ist Architektur?

→ Wie strukturiert man Dinge,
2 Grundregeln:

① Koppelung
② Kohäsion
} Bei Klebstoff: Adhäsion vs. Kohäsion





Kopplung

a) Funktionale Kopplung

```
Public    func Foo() {
              bar()
          }                } Kopplung

Private   func bar(){}
```

Diese beiden können nicht ohne einander, sie sind recht eng gekoppelt.

Größeres Beispiel wäre statt funcs (micro)-Services.

b) Datenkopplung

```
const steuersatz = 23

func Brutto(netto float64) float64 {
  brutto := netto * (1 + steuersatz/100)
  return brutto
}

func Steuersatz() int {
  // Kopplung an Konstante
  return steuersatz
}

func PferdeAufKoppel() int {
  return steuersatz
}
```

→ Plötzlich ist der Steuersatz von der Anzahl der Pferde abhängig! → bad idea.

c) Temporale Kopplung

d) ...

⇒ Man kann Kopplung nicht verhindern, sollte aber bewusst damit umgehen.

* So lose wie möglich, so eng wie nötig.

Kohäsion

↳ z.B. Alltag
- Gewürze beieinander, Nudeln, Steuerunterlagen → statt Lebensmittel nach Farbe
  - ↳ Inhaltliche Zusammengehörigkeit = cool

↳ z.B. Ordnerstruktur
- Controller beieinander, Models, Views, ...
  - ↳ Keine fachliche Zusammengehörigkeit = uncool
  - → Ist aber halt auch klar, weil die Frameworks ja für alle möglichen Apps Boilerplaten wollen, daher macht die Vorgabe der technischen Gleichartigkeit naheliegend.

IDEALFALL:

Niedrige Kopplung ❤ Hohe Kohäsion

- weil eine Änderung nicht zig Änderungen nach sich zieht
- Änderungen lokal und überschaubar

- Änderungen sind fachlich / inhaltlich beieinander

Konzepte

DRY – Don't Repeat Yourself

↳ Wird zu oft als nie Copy/Paste ausgelegt

↳ in Wahrheit, vgl. Pragmatic Programmer, beschreibt man an genau einer Stelle die Logik / fachliche

z.B. Versandhandel:
Business rules:

1) Bestellung darf nicht mehr als 3 Artikel umfassen } if number of articles > 3 { fachlicher Kont }

2) Es dürfen nicht mehr als 3 Artikel in einem Paket versendet werden } if number of articles > 3 { fachlicher Kontex }

⚠ Obwohl copy/paste, völlig ok, sogar richtig, weil fachlich komplett unterschiedlicher Kontext.

Würde man das in eine Funktion auslagern, die sich beide Bereiche teilen, würde bspw. eine Portoänderung katastrophale Folgen haben!

Identischer Code muss nicht identisch sein!

→ wir hätten dann ein super tightes Coupling & niedrige Kohäsion ☹

→ Fachliche Doppelung ≠ technische Doppelung

ENTKOPPELUNGSMECHANISMEN

- Asynchrone statt synchrone Kommunikation
- Fehler einplanen (Timeouts, Latenz, Ausfall, ...)
- Indirekte statt direkte Kommunikation

Direkt:

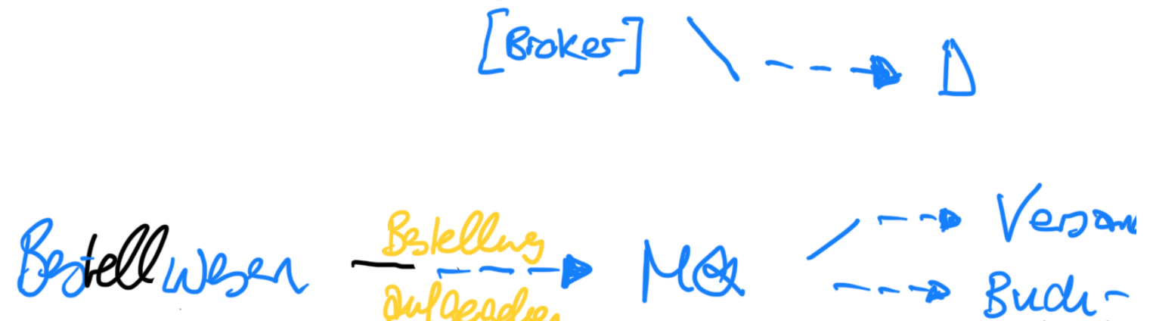
$$A \xrightarrow{\text{Cola}} B \quad \text{Kellner}$$

Indirekt:

$$\begin{tikzcd} & \\ A \arrow[ur, dashed] \arrow[r, dashed, "\text{Cola}"] \arrow[dr, dashed] & B \\ & \end{tikzcd}$$

- Events verwenden
  - um ein Bedürfnis zu Signalisieren
  - Events sind nicht zielgerichtet, sie gehen an die Allgemeinheit
  - Events erwarten keine synchrone Antwort

Pub/Sub



→ muss nichts über Receiver wissen, Receiver müssen einander nicht kennen, loosely coupled!

Abhängigkeiten:
* MQ an sich
* MQ-Protokolle ('amqp://', ...)
* Event-Format (wobei man das über Adapter lösen kann)

~~~
```golang
func HandleEvent(event Event) {
  if event.Type == "bestellung-aufgegeben" {
    kommunikationSvc.SendCommand("send-whatsapp", {
      // ...
    })
  }
}
~~~
```

Teams

- Vertikale Teams : Buchhaltung, Reklamation, ...
- Horizontale Teams : Frontend, Backend, ...
```