

# **Verteilte Systeme und Komponenten**

## **Einführung**

Martin Bättig

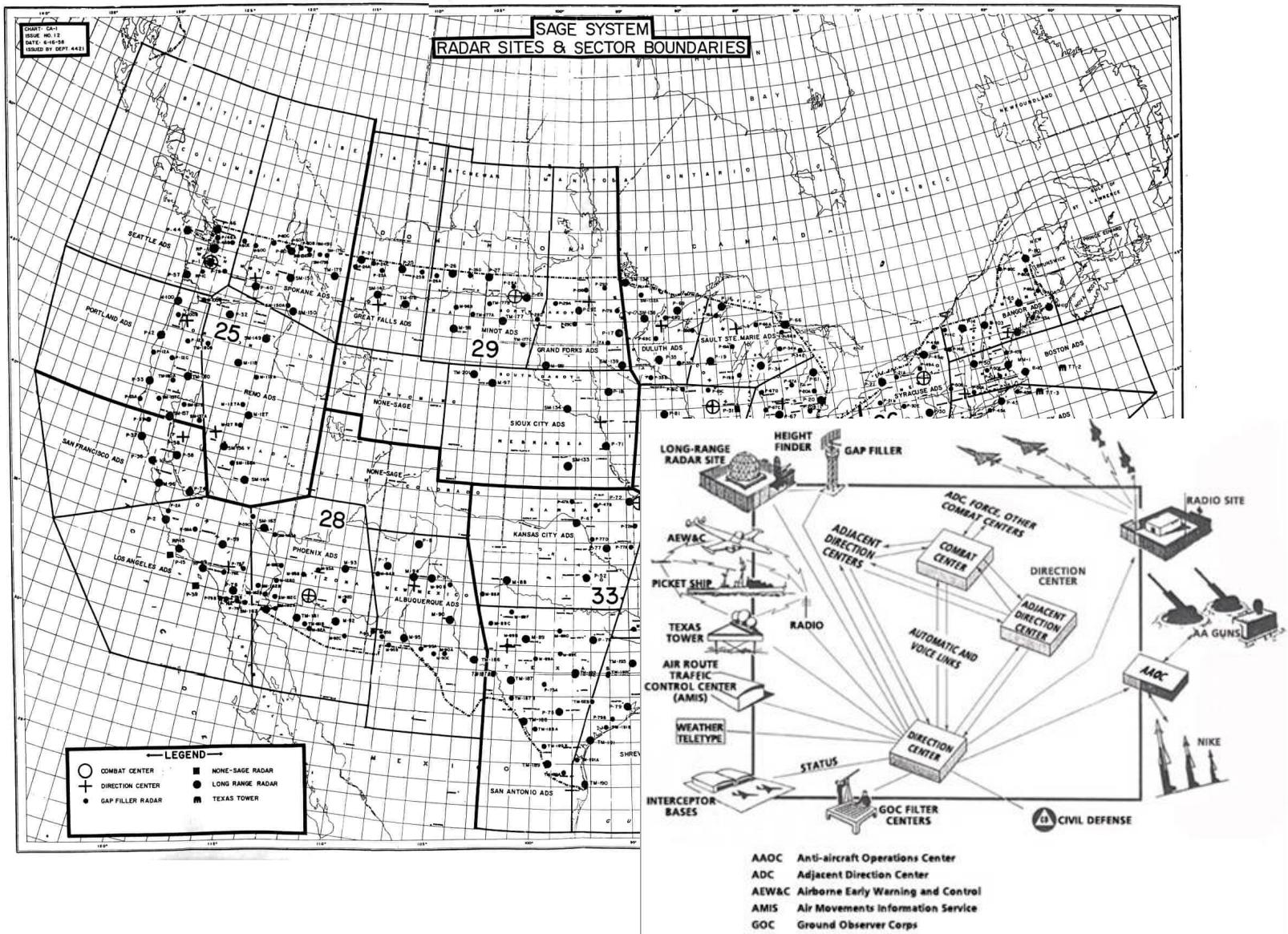


# Inhalt

- Einführung in das Modul
- Projektmanagement und -durchführung
- Einführung in verteilte Systeme
- Netzwerk-Einmaleins

# **Einführung in das Modul**

# Einleitung



# Organisatorisches - Plattformen

- **ILIAS:** Inputs & Projektmaterial

[https://elearning.hslu.ch/ilias/goto.php?target=crs\\_5501092](https://elearning.hslu.ch/ilias/goto.php?target=crs_5501092)

- **Forum:** Für Fragen und Diskussionen

[https://elearning.hslu.ch/ilias/ilias.php?ref\\_id=5569017&cmd=showThreads&cmdClass=ilrepositorygui&cmdNode=10h&baseClass=ilrepositorygui](https://elearning.hslu.ch/ilias/ilias.php?ref_id=5569017&cmd=showThreads&cmdClass=ilrepositorygui&cmdNode=10h&baseClass=ilrepositorygui)

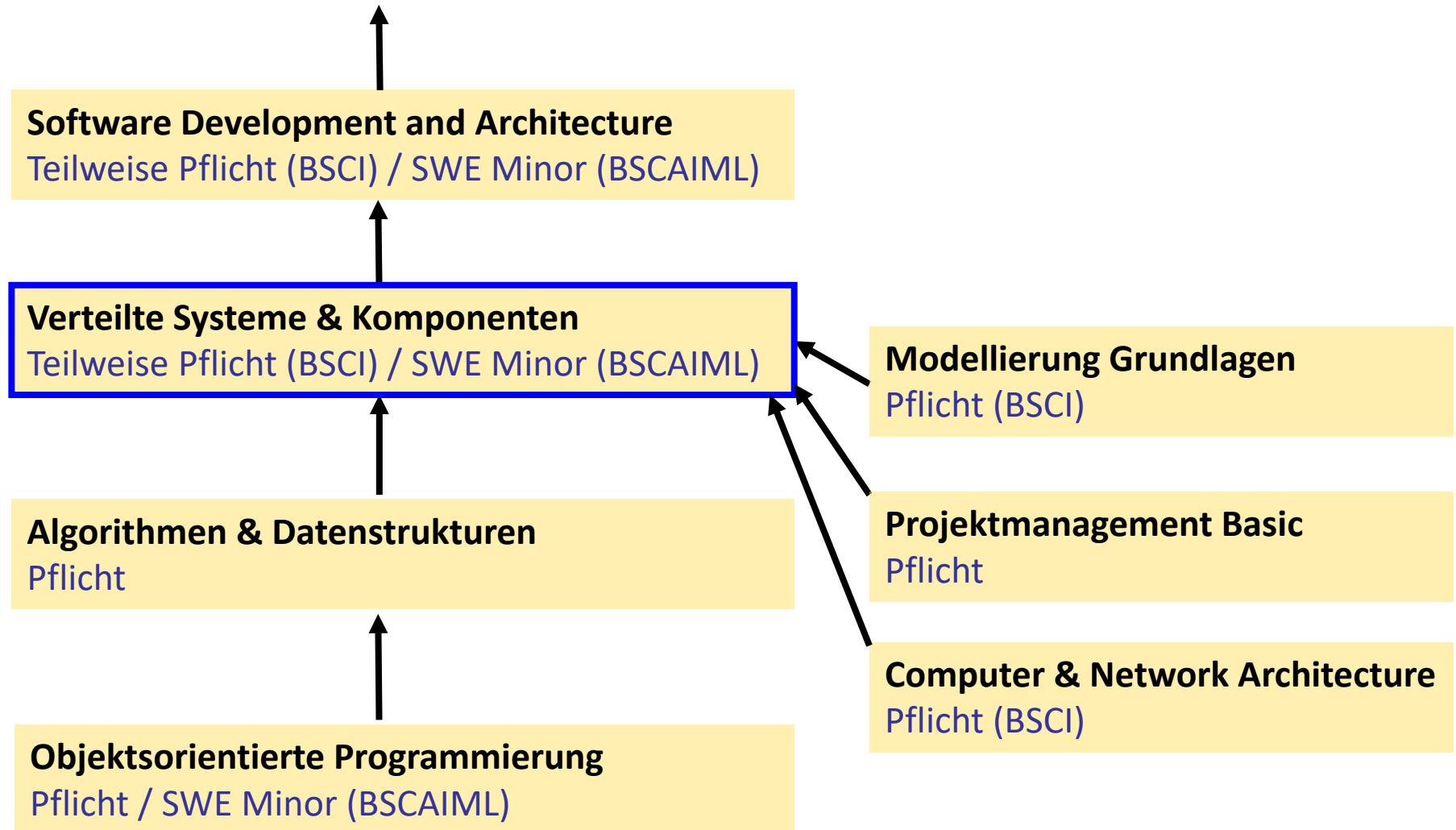
- **Zoom:** Streaming von Inputs für Ausnahmefälle (Krankheit, Termine, usw.)

<https://hslu.zoom.us/j/94103562967?pwd=VU9jRU5XSXZ2bnNFNXNkeFc3VEhuQT09>

(Meeting-ID: 941 0356 2967 / Kenncode: 146601)

**Hinweis:** Das Modul ist auf «vor Ort»-Unterricht ausgelegt.

# Einbettung im Curriculum



# Agenda

**Block 3 - Donnerstag: 15:30 - 17:50**

Thema	Raum
SW01 KW38 PR: Einführung (Modul, Projekt, Gitlab, Netzwerke) [Bam] Raum: S12_013	S1A_220
SW02 KW39 EP: SCM und Buildautomation (CI) [Gro]	S1A_220
SW03 KW40 VS: Message-orientierte Kommunikation [Bam]	S1A_220
SW04 KW41 KO: Schnittstellenabstimmung & -Freigabe / Architekturbeschreibung [Bam]	S1A_220
SW05 KW42 PR: Sprintreview und Sprintplaning	S1A_220
SW06 KW43 KO: Integrations- und Systemtest [Bam]	S1A_201
SW07 KW44 EP: Continuous Integration [Gro]	S1A_220
SW08 KW45 PR: Sprintreview und Sprintplaning	S1A_330
SW09 KW46 VS: Konsistenz und Replikation [Bam]	S1A_220
SW10 KW47 VS: Sicherheit in verteilten System [Bam]	S1A_220
SW11 KW48 PR: Sprintreview und Sprintplaning	S1A_220
SW12 KW49 Feiertag: Maria Empfängnis [Kein Unterricht]	
SW13 KW50 VS: Koordination verteilter Systeme [Bam]	S1A_220
SW14 KW51 PR: Sprintreview und Bereinigung	S1A_220

*Hinweis: Raumangaben sind ohne Gewähr. Bitte beachten Sie MyCampus oder die Anzeigetafeln für allfällige kurzfristige Anpassungen.*

## Studienelemente:

- VS: Verteilte Systeme
- KO: Komponenten
- EP: Entwicklungsprozess
- PR: Projekt

**Block 2 - Freitag: 12:50 - 15:10**

Thema	Raum	Projekt
VS: Serverprozesse mittels Sockets und RPC [Bam]	S1A_220	<b>Einführung</b> 29. Sep. - 20. Okt.
KO: Komponenten- & Schnittstellendesign [Bam]	S1A_220	
EP: Dep.-Management / Buildserver (CI) [Gro]	S1A_220	
KO: Modularisierung und Schichtenarchitektur [Bam]	S1A_220	
KO: Containervirtualisierung [Gro]	S1A_220	<b>Sprint 2</b> 20. Okt. - 10. Nov.
EP: Testing [Gro]	S1A_201	
VS: Fehlertoleranz und Resilienz [Bam]	S1A_220	
EP: Entwurfsmuster [Gro]	S1A_220	
EP: Deployment [Gro]	S1A_201	<b>Sprint 3</b> 10. Nov. - 1. Dez.
VS: Skalierung und Verteilung [Bam]	S1A_201	
EP: Code-Qualität [Gro]	S1A_220	
Freie Projektarbeit [Kein Unterricht]		
KO: Komponentenmodelle [Bam]	S1A_220	<b>Sprint 4</b> 1. Dez. - 22. Dez.
PR: Abschluss (Prüfungsinfo, Wettbewerb, Retrospektive)	S1A_220	

## Dozententeam:

- Martin Bättig [Bam] (Modulverantwortung)
- Roland Gisler [Gro]

# Konzept

- Inputs mit drei komplementären Tracks:
  - Entwicklungsprozess (EP)
  - Verteilte Systeme (VS)
  - Komponenten (KO)
- Übungen in **3er** oder **4er** Gruppen (Selbststudium)
  - Projekt "verteiltes Logger-System".
  - fünf auf einander aufbauende Arbeitsaufträge.
  - Bearbeitung ist Testatpflichtig.
- Leistungsnachweis:
  - Mündliche Prüfung basierend auf Ergebnissen des Projekts

# Testatbedingung

- Bearbeitung der fünf Arbeitsaufträge.
  - Schnittstellendefinition und Sprints 1-4.
  - Abgabe der geforderten Zwischenresultate pro Sprint.
  - Erwartet wird ein ernstgemeinter Lösungsversuch.
- Aktive Projektmitarbeit in der Gruppe.
  - Ausgewiesene Teilaktivitäten pro Gruppenmitglied.
- Obligatorische Teilnahme **aller Teammitglieder** an folgenden Terminen:
  - Schnittstellenabstimmung.
  - Sprint-Reviews 1-4 mit den Coaches.

# GitLab – auf EnterpriseLab (ELAB)

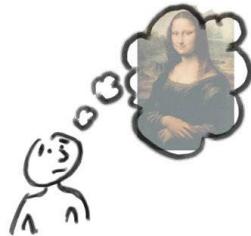
- Wir verwenden die Versionsverwaltung der GitLab der HSLU:  
<https://gitlab.enterpriselab.ch>
- Bitte ELAB-Konto erstellen, falls noch nicht vorhanden:  
<https://eportal.enterpriselab.ch/> (Registration -> Registration SwitchAAI)
- Anschliessend bitte **einmal** beim enterprise.gitlab.com einloggen
  - Benutzer ist dann in GitLab erfasst und wir können die Rechte vergeben.
- EnterpriseLab FAQ: <https://el-core.pages.enterpriselab.ch/faq/>
- Andere Probleme: <https://ticket.enterpriselab.ch/otrs/customer.pl>

# Prüfung

- Format: Mündlich.
- Dauer: 25 Minuten.
- Basiert auf der Projektarbeit (Architekturbeschreibung und Programmcode).
  - «Closed Book Exam»: Material von uns bereit gestellt
- Prüfer: Martin Bättig und Roland Gisler.

# **Projektmanagement und -durchführung**

# Sequential vs. Iterativ



nicht sequenziell.....



... sondern iterativ



aus Jeff Patton (2014), User Story Mapping

# Scrum: Agiles Produktentwicklungsframework

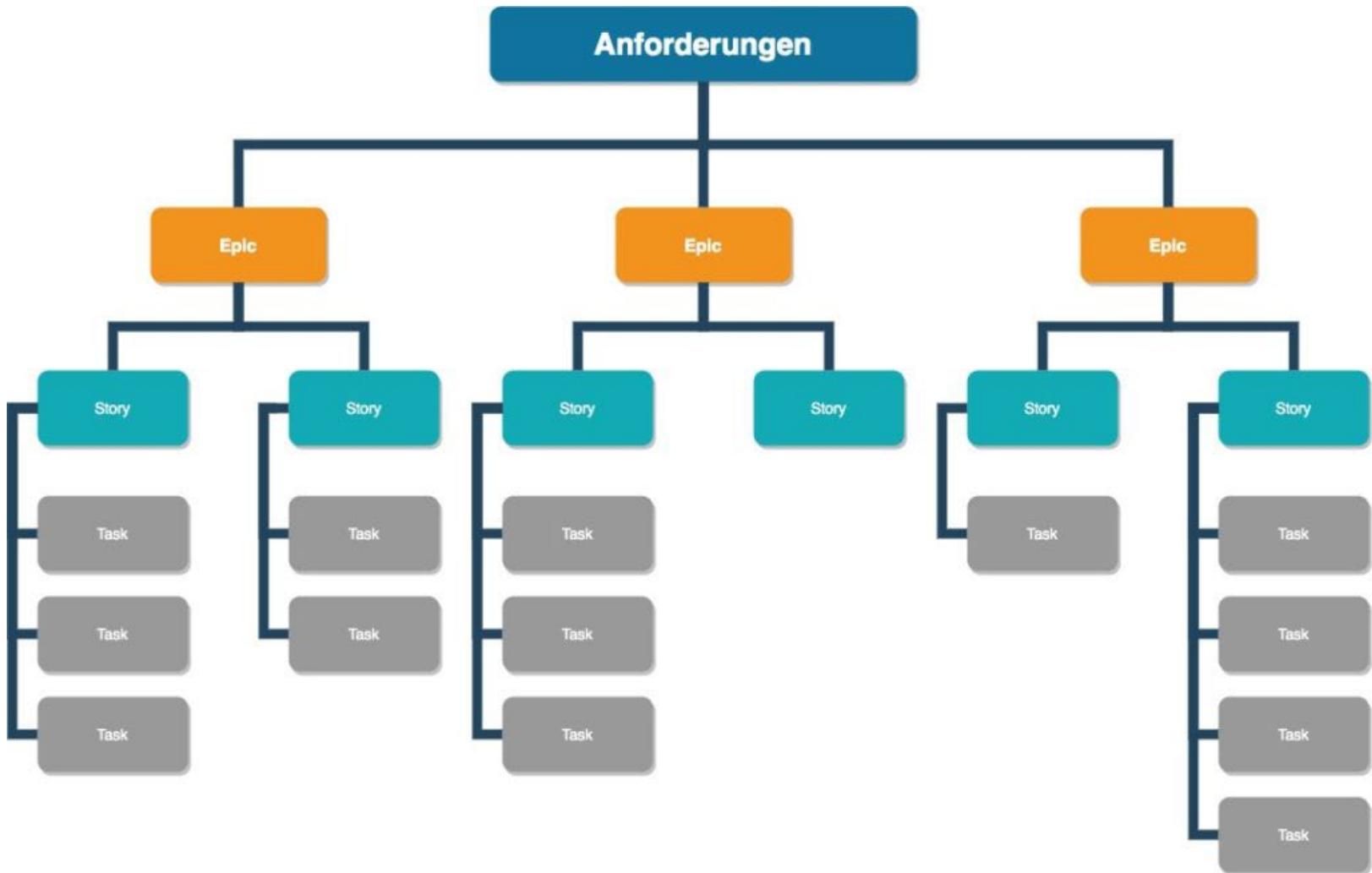


Agiles Vorgehen ermöglicht:

- Start der Konzeption und Realisation mit unvollständigen Anforderungen.
- Änderungen der Anforderungen während der Konzeption und Realisation.
- Priorisierung der Anforderungen (u.U. werden nicht alle erfüllt).

**In einer geordneten Art und Weise.**

# Gefässe in Scrum-Vorgehensmodell



# Sprintplanung

- Sprintziel festlegen: Vom PO zu erarbeitendes und mit Team abzustimmendes Ziel, das mit dem Sprint erreicht werden soll.  
=> beantwortet die Frage nach dem Sinn des Produkt-Inkaments, das im Rahmen dieses Sprints erstellt wird.
- Schätzbare Items aus dem Produkt-Backlog werden in den Sprint aufgenommen: Einträge mit hoher Priorität zuerst.
- **Eine Aufnahme in den Sprint verpflichtet zur Abarbeitung.**
- Ob schätzbar oder nicht entscheidet Scrum-Team.
  - Konfliktpotenzial zwischen Product-Owner und Scrum-Team.
- Schätzung typischerweise durch alle Teammitglieder.
- Einheit nach Scrum: Storypoints.
  - Storypoints nur bei eingespielten Teams sinnvoll.

# Sprintplanung im GitLab – Begriffe

- GitLab kann zur Verwaltung der Stories im Backlog und zur Planung und Controlling der Sprints eingesetzt werden.
- Abweichende Terminologie in GitLab bzgl. Standard Scrum-Terminologie:

Scrum-Artifakt	GitLab-Feature
User story	Issue
Task	Task lists
Epic	Epics
Points and estimation	Time tracking & Weights
Produkt-Backlog	Issue-Liste mit Prioritäts-Labels
Sprint / Iteration	Milestone
Burndown-Chart	Burndown-Chart

# Sprintplanung im GitLab – Vorbereitung

**Konzept:** jedes Team ist eine Gruppe in GitLab.

- Epics, Boards, Labels, Milestones sind auf **Ebene Gruppe**
- Issues sind auf **Ebene Projekt:** Produkt-Backlog
- Priorisierung von Issues im Produkt-Backlog:
  - Zusätzliche Labels wie z.B. Prio\_A, Prio\_B, Prio\_C einführen
  - Labels entsprechend den Issues zu ordnen.
  - Issue-Board kann danach gefiltert werden.

# Sprintplanung im GitLab – Neuen Sprint planen

## Auf Gruppenebene:

- Einen entsprechenden Milestone mit sprechendem Namen (Sprint 01, Sprint 02..) und Daten wählen.
- Board öffnen, ohne Milestone Filter -> gesamtes Backlog sichtbar.
- Stories aus Backlog auswählen und Labels (ToDo etc.) zuweisen.
- Alle zugewiesenen Issues auf den aktuellen Milestone setzen (Sprint-Backlog):
- Issue auf Board anwählen -> rechts erscheint Sidebar-Menü -> dort Milestone wählen
- Abschluss, nicht erledigter Stories: Milestones weg und remove from Board (d.h. Label weg).

# Projektcontrolling

**Zentrale Frage:** Erreichen wir das Projektziel? (Zeit, Kosten):

- Wieviel ist noch zu tun und viel Zeit und Ressourcen haben wir noch?
- Wie gut wird geschätzt? Wie kann man es bessern?
- Controlling im VSK wird mittels Gitlab gemacht.

# Sprintplanung im GitLab – Controlling

- Um den geschätzten Aufwand für eine User-Story festzuhalten im Kommentarfeld des betreffenden Issue `/estimate` eingeben und die geschätzte Zeit in Wochen, Tagen, Stunden und Minuten angeben.  
**Beispiele:** 1w 3d, 5h 40m oder 4h 30m
- Um die Zeit einzutragen, wie lange man bis jetzt an der User-Story gearbeitet hat, `/spend` plus die Zeit im Kommentarfeld des Issue eingeben.
- Im Issue rechts unter `time tracking` sieht man die geschätzte Zeit und die bereits aufgewandte Zeit für die User-Story.
- Anleitung um ein Issue Template im GitLab zu erstellen:  
[https://gitlab.com/help/user/project/description\\_templates.md](https://gitlab.com/help/user/project/description_templates.md)

## Sprintplanung im GitLab – Bemerkungen

- Auswertungen erfolgen auf Ebene "Milestone" -> diese verwenden (pro Sprint!).
- Boards sind sowohl auf Gruppen als auch Projektebene die gleichen.
- Milestones sind entweder auf Projekt oder Gruppenebene vorhanden / sichtbar. Können zu Gruppen-Milestones promoted werden.

# Code-Reviews nach Google

Googles Code-Review-Guidelines sind öffentlich:

<https://google.github.io/eng-practices/review/reviewer/>

## How to do a code review

The pages in this section contain recommendations on the best way to do code reviews, based on long experience. All together they represent one complete document, broken up into many separate sections. You don't have to read them all, but many people have found it very helpful to themselves and their team to read the entire set.

- [The Standard of Code Review](#)
- [What to Look For in a Code Review](#)
- [Navigating a CL in Review](#)
- [Speed of Code Reviews](#)
- [How to Write Code Review Comments](#)
- [Handling Pushback in Code Reviews](#)

See also the [CL Author's Guide](#), which gives detailed guidance to developers whose CLs are undergoing review.

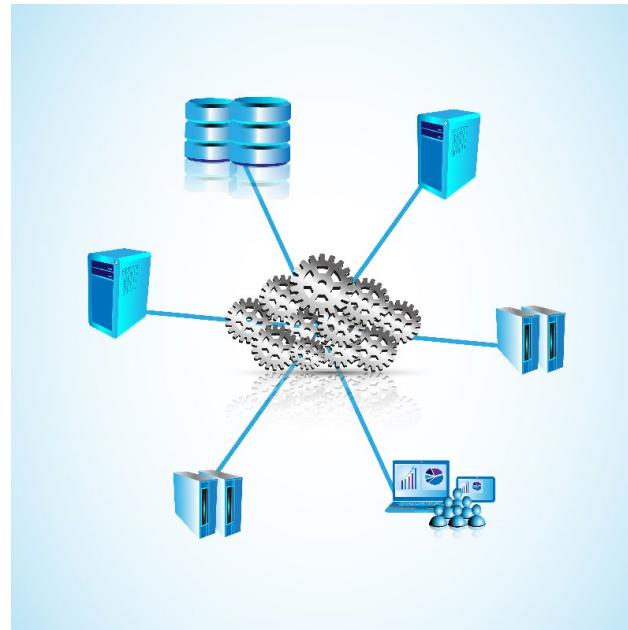
## Einige Auszüge (übersetzt, vereinfacht):

- Eine Änderung muss nicht perfekt sein, muss das System in seiner Gesamtheit jedoch verbessern.
- Der Review sollte sobald als möglich nach Anfrage gemacht werden (z.B. nach einen Kontext-Wechsel). Maximal ein Arbeitstag.
- Falls eine Änderung zu gross ist: Eine Aufteilung der Änderung verlangen. Erkennungsmerkmal: Man hat keine Zeit für den Review. Bei der Linux-Kernel-Entwicklung gibt es eine ähnliche Praxis.

# **Einführung in verteilte Systeme**

# Übung: Nutzen und Fallstricke verteilter Systeme

- **Gruppe A:** Überlegen Sie sich, welche Fallstricke in einem verteilten System auftreten können.
- **Gruppe B:** Überlegen Sie sich, welchen Nutzen man aus einem verteilten System ziehen kann.

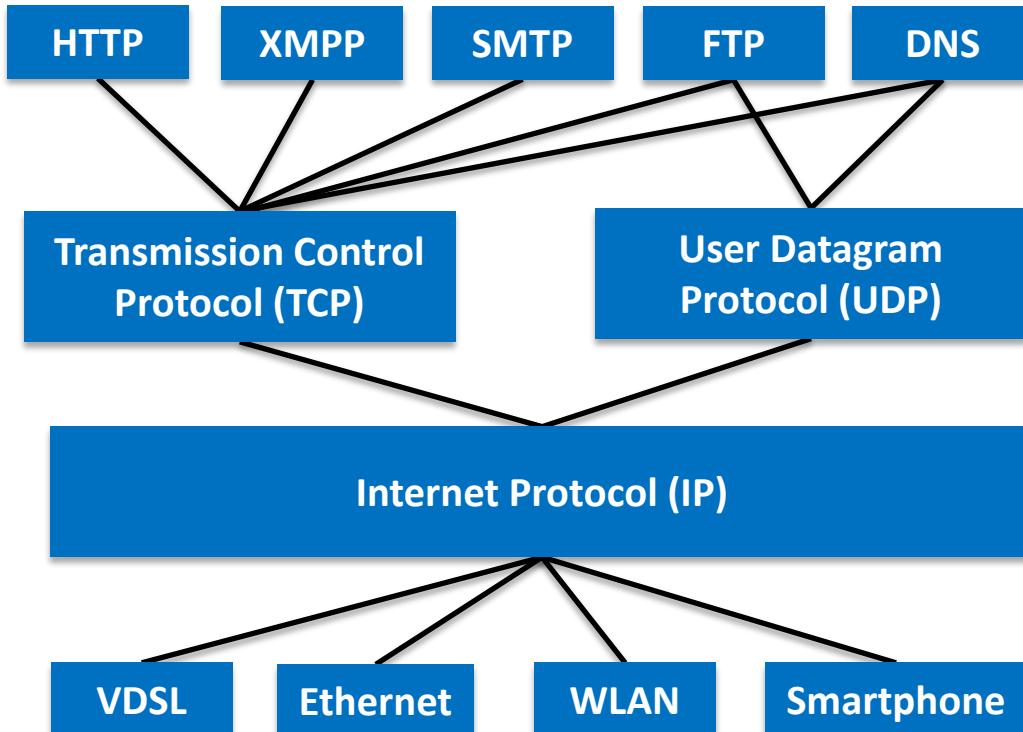


# **Netzwerk-Einmaleins**

\* Im Modul "Computer & Network Architecture"  
werden die Netzwerk Konzepte ausführlich behandelt.

# Internet-Schichtenmodel

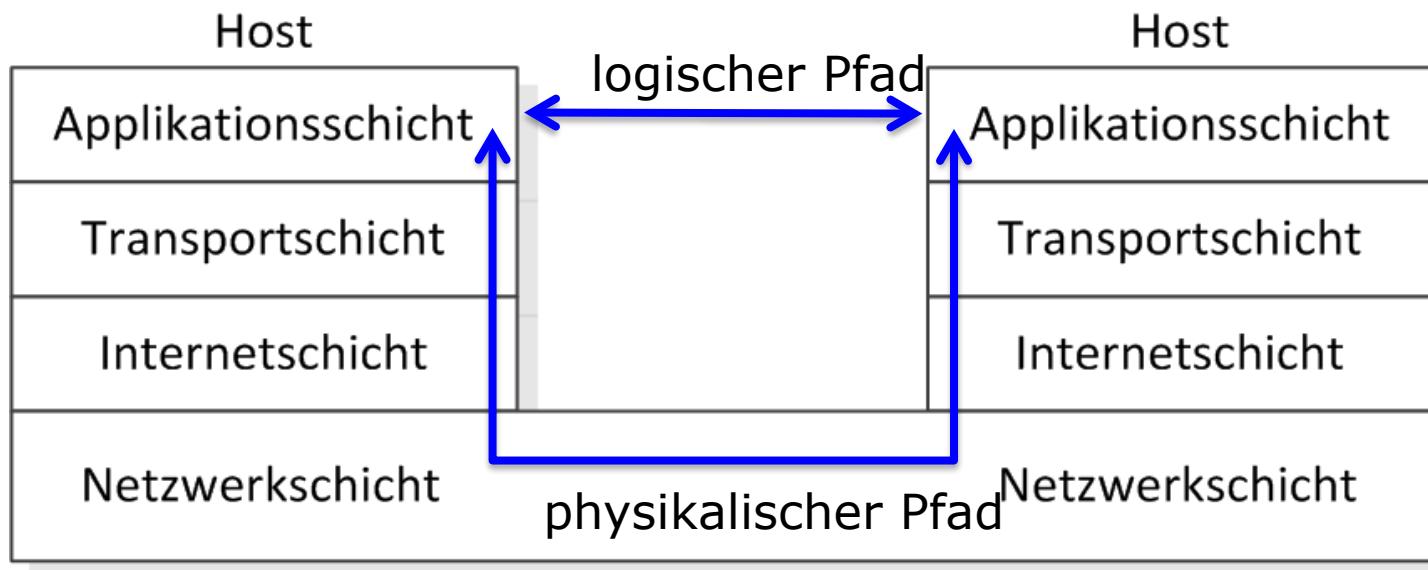
- **Applikationsschicht**  
(application layer)
- **Transportschicht**  
(transport layer)
- **Internetschicht**  
(internet layer)
- **Netzwerkschicht**  
(network layer)



Beispiele von Protokollen in den verschiedenen Schichten

# Kommunikation

- Physikalisch gehen die Daten der Host-zu-Host Kommunikation durch alle Schichten.
- Logisch gehen die Daten von Applikation zu Applikation.
- Die Kommunikationsdetails sind für die Applikation **transparent**:



# Begriffsdefinitionen

- **Host:** Ein am Netzwerk angeschlossener Rechner.
- **IP-Adresse:** Jeder Host bekommt eine im Netzwerk eindeutige IP-Adresse.
- Versionen von IP-Adressen sind:
  - **IPv4** eine 32-Bit-Zahl
  - **IPv6** eine 128-Bit-Zahl
- **Hostnamen:** Statt IP-Nummern verwendet.
- **Domain Name Service (DNS):** Dienst zur Zuordnung zwischen Name und IP-Adresse.

<https://de.wikipedia.org/wiki/IPv4>

<https://de.wikipedia.org/wiki/IPv6>

[https://de.wikipedia.org/wiki/Domain\\_Name\\_System](https://de.wikipedia.org/wiki/Domain_Name_System)

# Teilnehmer in einem Netzwerk

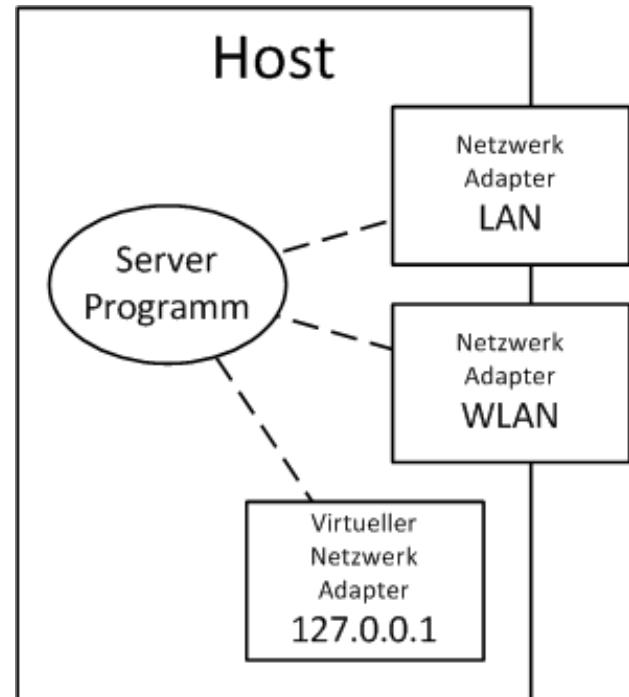
- **Server (dt. Bediener):** Dienstleister, der in einem Computersystem Daten oder Ressourcen zur Verfügung stellt.
  - System kann aus einem Computer oder einem Netzwerk mehrerer Computer bestehen.
  - Mehrdeutiger Begriff: Serverprogramm (Dienstanbieter) vs. Servercomputer (auf welchem Serverprogramme laufen).
- **Client (dt. Kunde):** Dienstnehmer, der in einem Computersystem Dienste von Servern benutzt.

# Network Interface Controller (NIC)

- Verbindung zwischen Computer und Netzwerk (privaten oder öffentlich).
- Physisch (On-board, PCI, USB, ...) oder virtuell (Loopback, Docker, etc.)
- Loopback-Netzwerk: Virtuelles Netzwerk, welches nur eigenen Host umfasst:
  - Kein Zugriff von aussen
  - Hostname: localhost (default)
  - IPv4: 127.0.0.1
  - IPv6: ::1

# NICs im Geräten

- Oft mehr als ein NIC pro Gerät, z.B. für:
  - LAN (Local Area Network)
  - WLAN (Wireless LAN)
  - Virtuelles Netzwerk
- Serverprogramm muss wissen, auf welchen Netzwerkadapters es auf Verbindungen warten soll.



# IP-basierte Netzwerke

- **IP-Adresse** identifiziert Teilnehmer in IP-basierten Netzwerken:
  - Stufe Internetschicht.
  - Kein Host im Internet hat die gleiche IP-Adresse (\* Spezialfall: NAT).
- **Portnummer** identifiziert Serverprogramm auf einem Teilnehmer:
  - Stufe Transportschicht.
  - Leitet empfangene Daten an Serverprogramm weiter.
- **Socket:** Kommunikationsendpunkt
  - Spezifiziert mittels Tupel (IP-Adresse, Port-Nummer).

# Host- und IP-Adressen

- Datenaustausch im Internet geschieht durch IP-Pakete.
- Empfänger der Pakete wird durch eine Kennung, die numerische IP-Adresse, identifiziert.
- Diese Zahl ist schwer zu behalten und z.T. volatil, weshalb oft der Hostname Verwendung findet.
- Die Konvertierung von Hostnamen in IP-Adressen übernimmt ein Domain Name Server (DNS).

# **Wichtigste Protokolle der Transportschicht**

## **Transmission Control Protocol (TCP):**

- Zuverlässiges, verbindungsorientiertes, Bytestrom-Protokoll.
- Hauptaufgabe: Bereitstellung eines sicheren Transports von Daten durch das Netzwerk.

## **User Datagram Protocol (UDP):**

- Unzuverlässiges, verbindungsloses Protokoll.
- Unzuverlässig: Daten kommen möglicherweise nicht bei Ziel-Host an. Daten, welche ankommen, sind jedoch korrekt.

# Übung: Mini-Logger mittels \*ix-Tools (1)

Übung mittels Online-Programmierumgebung:

<https://replit.com/@mbaettig/Logger-using-SOCAT>

**Kein HSLU-Dienst:** Neues Konto anlegen, wenn Sie die Übung machen möchten.

- Logger-Server starten: `./logger_server.sh <tcp|udp> <port>`
- Logger-Client starten: `./logger_client.sh <tcp|udp> <host> <port> <msg>`

## Aufgabe:

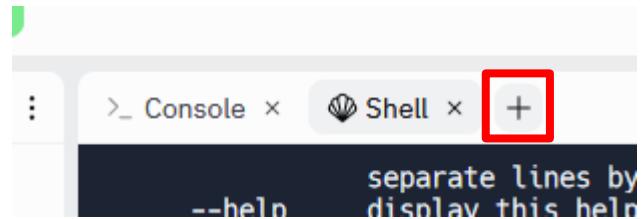
- Experimentieren Sie mit dem Mini-Logger. Ideen auf nächstem Slide.
- Was funktioniert? Was nicht?
- **Notieren Sie sich Ihre Beobachtungen. Wir diskutieren im Plenum.**

# Übung: Mini-Logger mittels \*ix-Tools (2)

**Machen Sie folgendes:**

- Starten Sie einen Logger-Server.
- Senden Sie einige Nachrichten an diesen Server. Machen Sie dafür eine neue Shell auf (siehe Bild oben).
- Starten Sie einen zweiten (oder noch mehr) Server.
- Senden Sie jeweils eine Log-Message, an die unterschiedlichen Server.
- Senden Sie auch mal eine Message an den Host: grey.baettig.ws / Port: 12345.
- Kommen Ihre Nachrichten an?
- Variieren Sie Protokoll und Port.

- **Hinweis:** Bei repl.it kann der Server nur ans Loopback-Netzwerk binden.



# **Fragen?**

# Literatur und Quellen

- Der offizielle ScrumGuide von Ken Schwaber and Jeff Sutherland, 2017:  
<https://www.scrumguides.org/scrum-guide.html>
- How to use GitLab for Agile software development von Victor Wu, 2018:  
<https://about.gitlab.com/blog/2018/03/05/gitlab-for-agile-software-development/>

Verteilte Systeme und Komponenten

# **Serverprozesse mittels Sockets und RPC**

Martin Bättig



# Inhalt

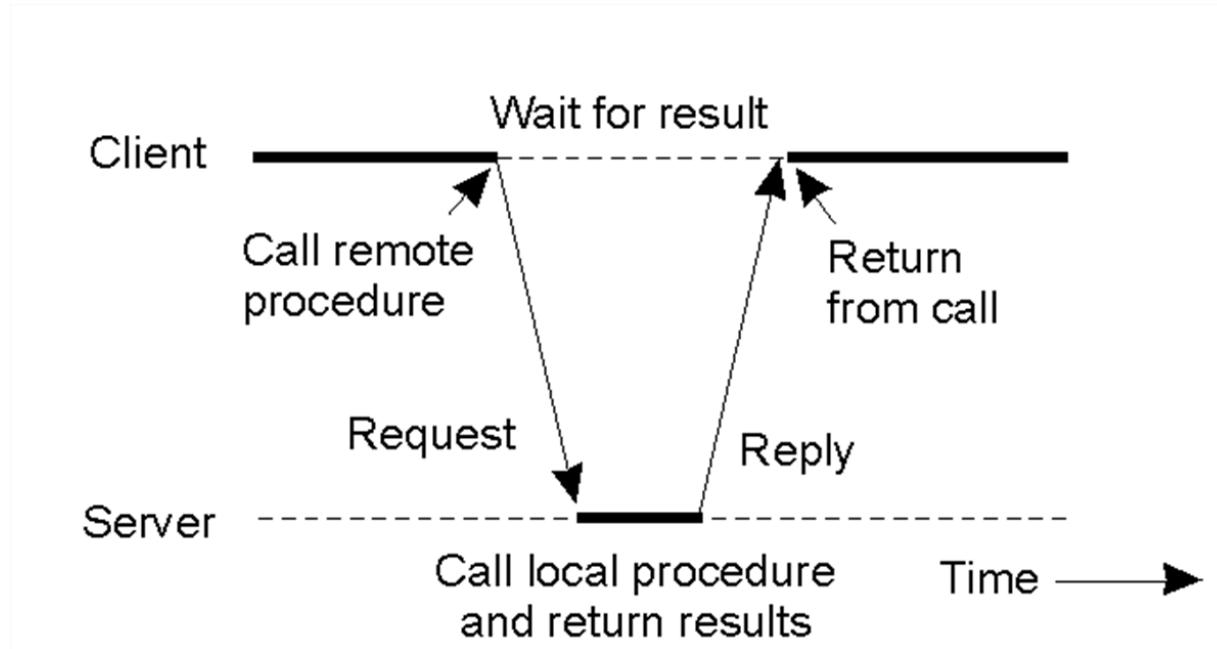
- Remote-Procedure-Call (RPC)
- TCP Client- und Serverprogramme
- gRPC
- Zusammenfassung

# Lernziele

- Sie verstehen das Prinzip des Remote-Procedure-Calls.
- Sie kennen die verschiedenen Arten um ein Serverprogramm zu implementieren und wissen, wann welche einzusetzen.
- Sie wissen, welche Aktionen nötig sind, um Daten verbindungsorientiert senden zu können.
- Sie wissen was ein Kommunikationsprotokoll ist und können ein einfaches Protokoll in Java umsetzen können.
- Sie kennen den Lebenszyklus eines TCP-Servers und können die einzelnen Elemente mit einem Java-Programm in Beziehung stellen.
- Sie können sowohl Java Client-Programme sowie Java Server-Programme analysieren und implementieren.
- Sie kennen die generelle Funktionsweise von gRPC.

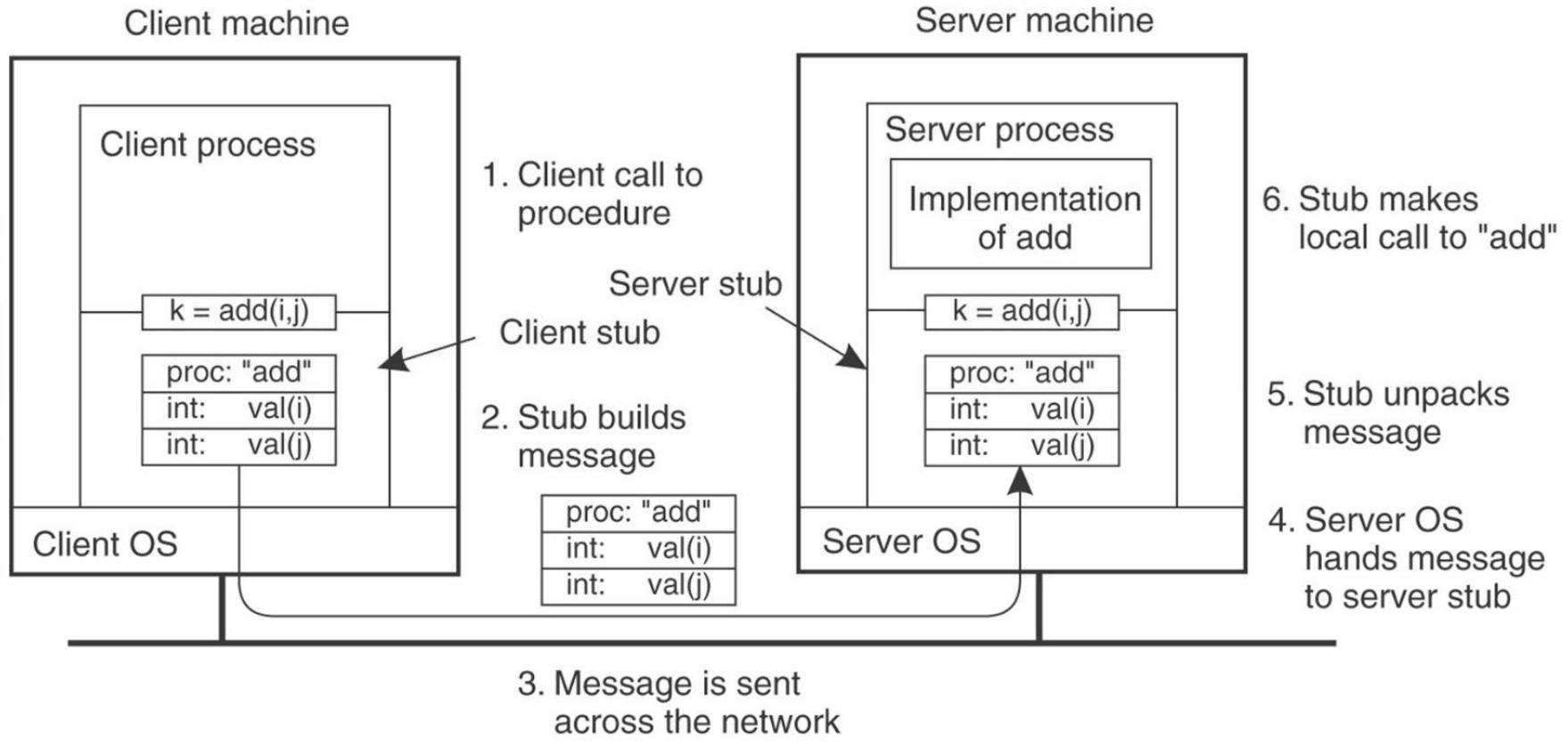
# Remote-Procedure-Call (RPC)

Ziel: Remote Funktionsaufruf analog zu lokalem Funktionsaufruf:



**Beispiel:** `result = doIt(a, b);`

# Remote-Procedure-Call: Funktionsprinzip

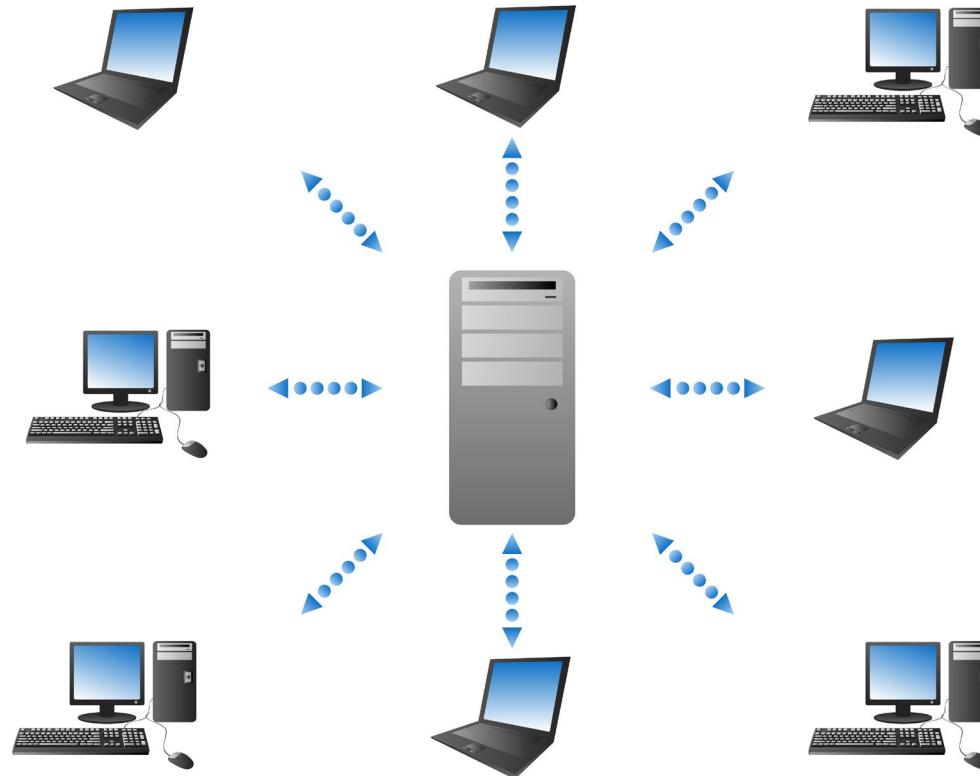


# Remote-Procedure-Call

- Vielfach verwendetes Konzept (gRPC, DCE, RMI, etc.).
- Was sind die Herausforderungen?

# Parallele (nebenläufige) Verarbeitung

- Serverprogramm muss ggf. mehr als einen Client **parallel** bedienen.



# Varianten der parallelen Verarbeitung

Variante	Einsatzgebiet
<b>Blocking I/O, Single-Threaded</b>	Eine langlebige Verbindung oder wenige kurzlebige Verbindungen.
<b>Blocking I/O, Mehrere Prozesse (Fork)</b>	Wenige langlebige Verbindungen.
<b>Blocking I/O, Mehrere Threads</b>	Wenige langlebige Verbindungen.
<b>Blocking I/O, Thread-Pools</b>	Viele kurzlebige Verbindungen.
<b>Non-Blocking I/O, Single-Threaded</b>	Viele Verbindungen (kurz- oder langlebig) mit wenig Bearbeitungszeit.
<b>Non-Blocking I/O, mehrere Threads</b>	Viele Verbindungen (kurz- oder langlebig) mit wenig bis viel Bearbeitungszeit.

# Parameterübergabe

**Beispiel:** Anhängen eines Datensatzes (data) an eine Liste (dbListe) auf einem entfernten System, Rückgabe als neue Liste (newlist).

```
newlist = append(data, dbList)
```

## Parameterübergabe:

### Lokal:

- By-value: Erstelle Kopie
- By-reference: Übergebe Referenz

### Remote:

- By-value: Erstelle Kopie
- By-reference: ???

## Fragen:

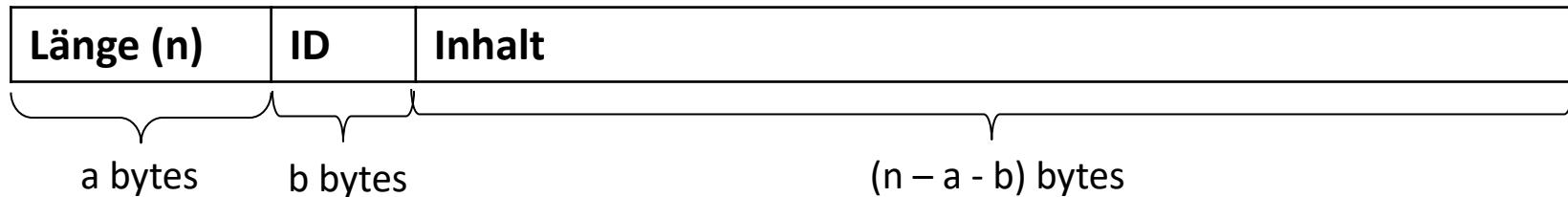
- Kosten?
- Wo sind Primitive oder Strukturen gespeichert?
- Wie identifizierte ich Primitive oder Strukturen?

# Kommunikationsprotokoll

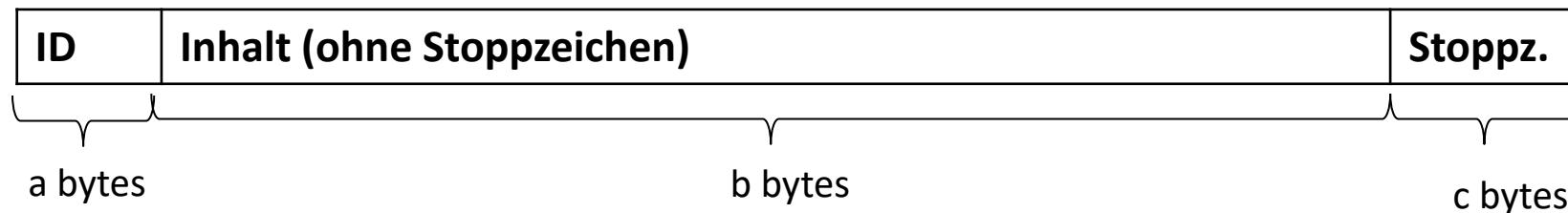
- Notwendig zur Verständigung zwischen zwei Maschinen (oft Client und Server).
- Hat Anforderungen an den Übertragungskanal (zuverlässig vs. unzuverlässig)
- Definiert generelle Nachrichtenstruktur.
  - Bspw.: Binär/Text, Länge, Stopnzeichen, Nachrichten ID, usw.
- Umfasst eine bestimmte Anzahl Nachrichten.

# Generelle Nachrichtenstruktur: Beispiele

Mit Längenangabe:



Mit Stopnzeichen:



# Nachrichten

- Teil eines Kommunikationsprotokolls.
- gesendet über Kommunikationskanal.
- enthalten Elemente bestimmter Datentypen.
  - ID:
    - identifiziert die Nachricht (z.B. GET, POST, ... bei HTTP)
    - Nicht immer benötigt: Bei strikten Interaktionen zwischen Kommunikationspartnern ist die ID unnötig. (z.B. Schach)
  - Argumente, oft abhängig von der ID:
    - Einfache Datentypen (Integer, Strings, Floating-point).
    - Strukturen oder Arrays
    - können zusätzliche Informationen enthalten, basierend auf der Art der Nachricht, z.B. eine Aktion oder Anfrage (Wandle X in Y und gib Resultat zurück oder Berechne X mit Hilfe von A,B,C).

# Aufbau einer HTTP-Nachricht

## ■ HTTP Request:

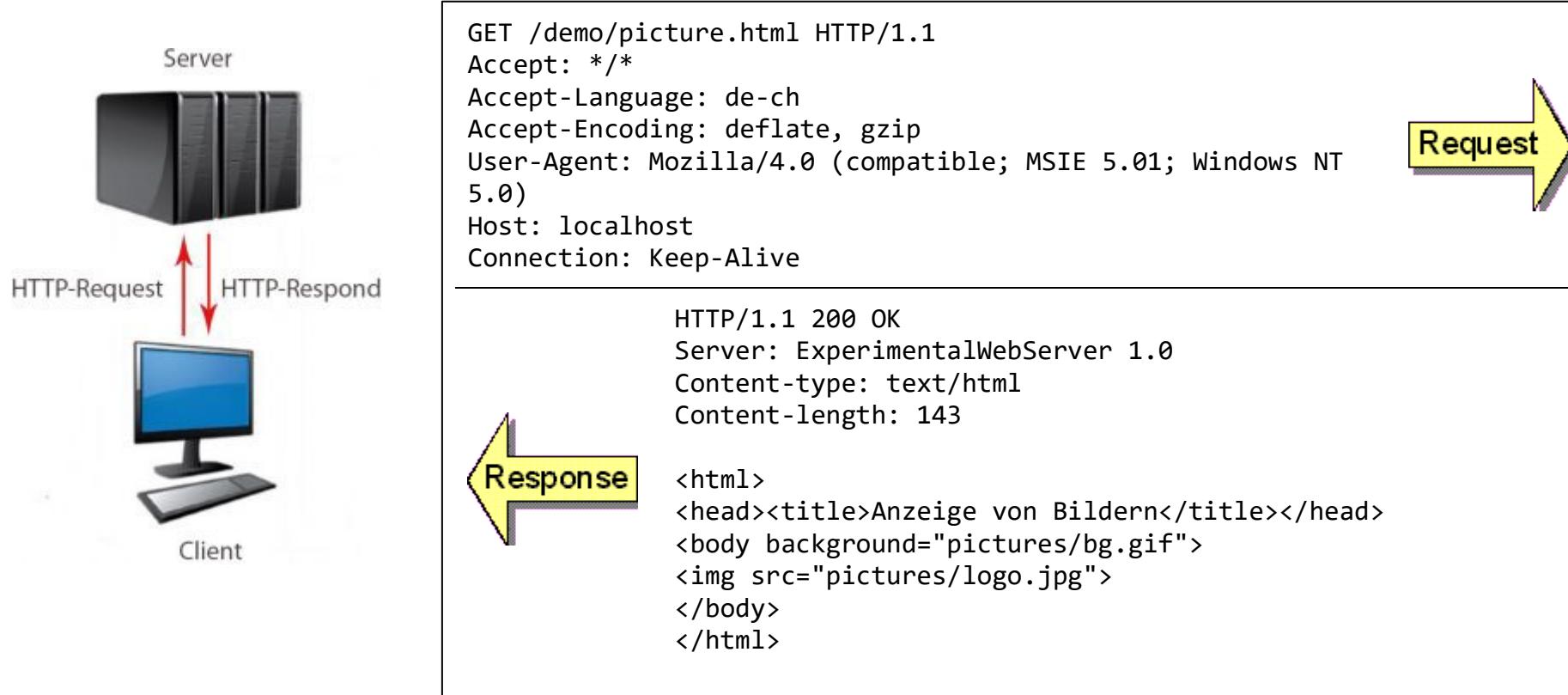
ID	METHOD
Argumente	URL
	HTTP/version
	General Header
	Request Headers
	Entity Header (optional)
	Leerzeile
	Request Entity (falls vorhanden)

## ■ HTTP Response:

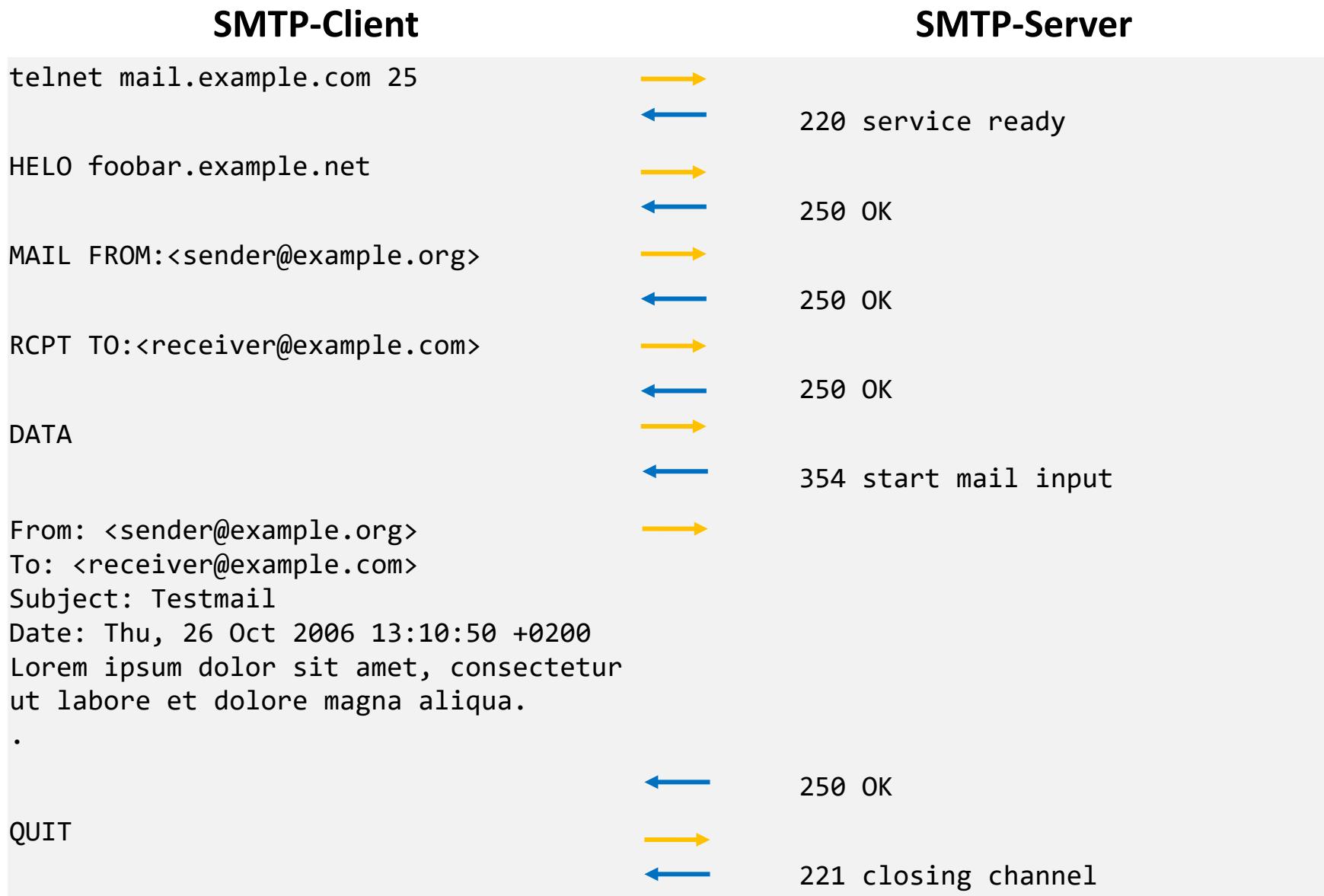
ID	HTTP/version
Argumente	Status Code
	Reason Phrase
	General Header
	Response Header
	Entity Header (optional)
	Leerzeile
	Resource Entity (falls vorhanden)

# Beispiel: HTTP-Kommunikation

- Anforderung des HTML-Dokuments demo/picture.html



# Beispiel: SMTP-Kommunikation

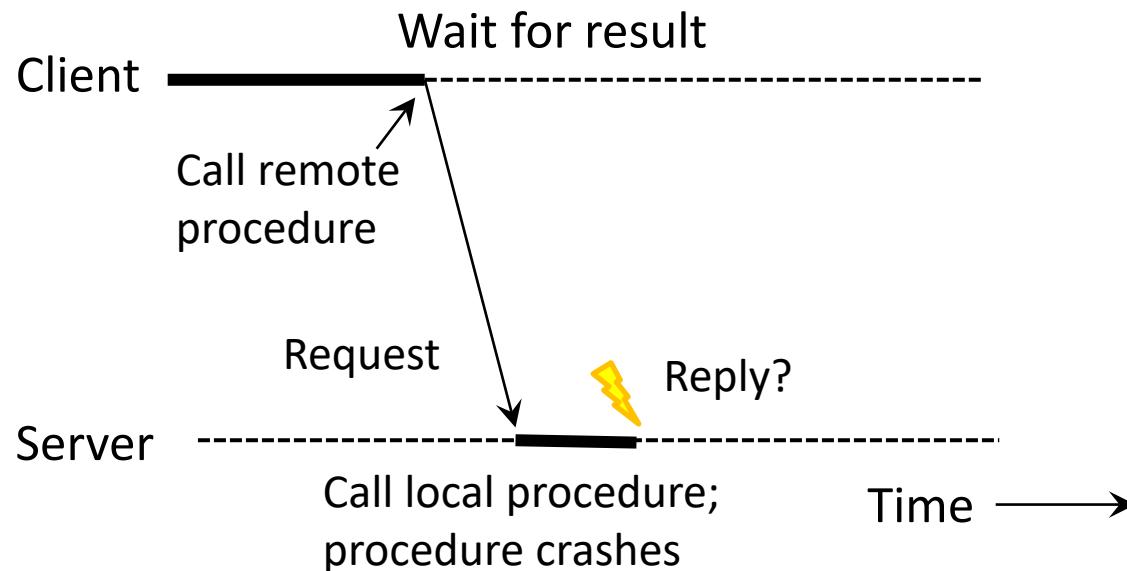


# Fehlerbehandlung

- Wie erfolgt die Fehlerbehandlung?

- Versuch wiederholen?
- Aktion abbrechen?
- Programm abbrechen?
- etc.

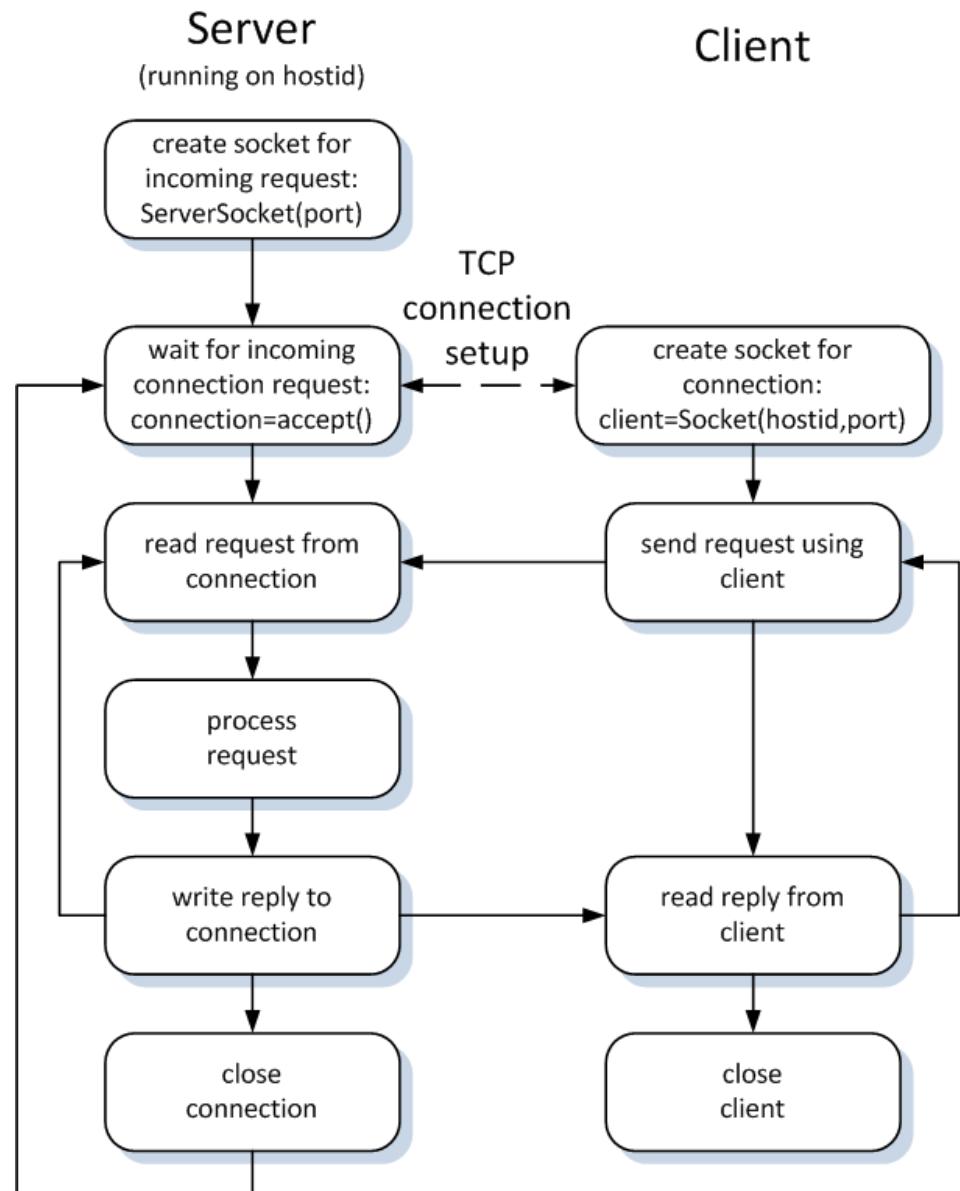
## Beispiel:



# **TCP Client- und Serverprogramme**

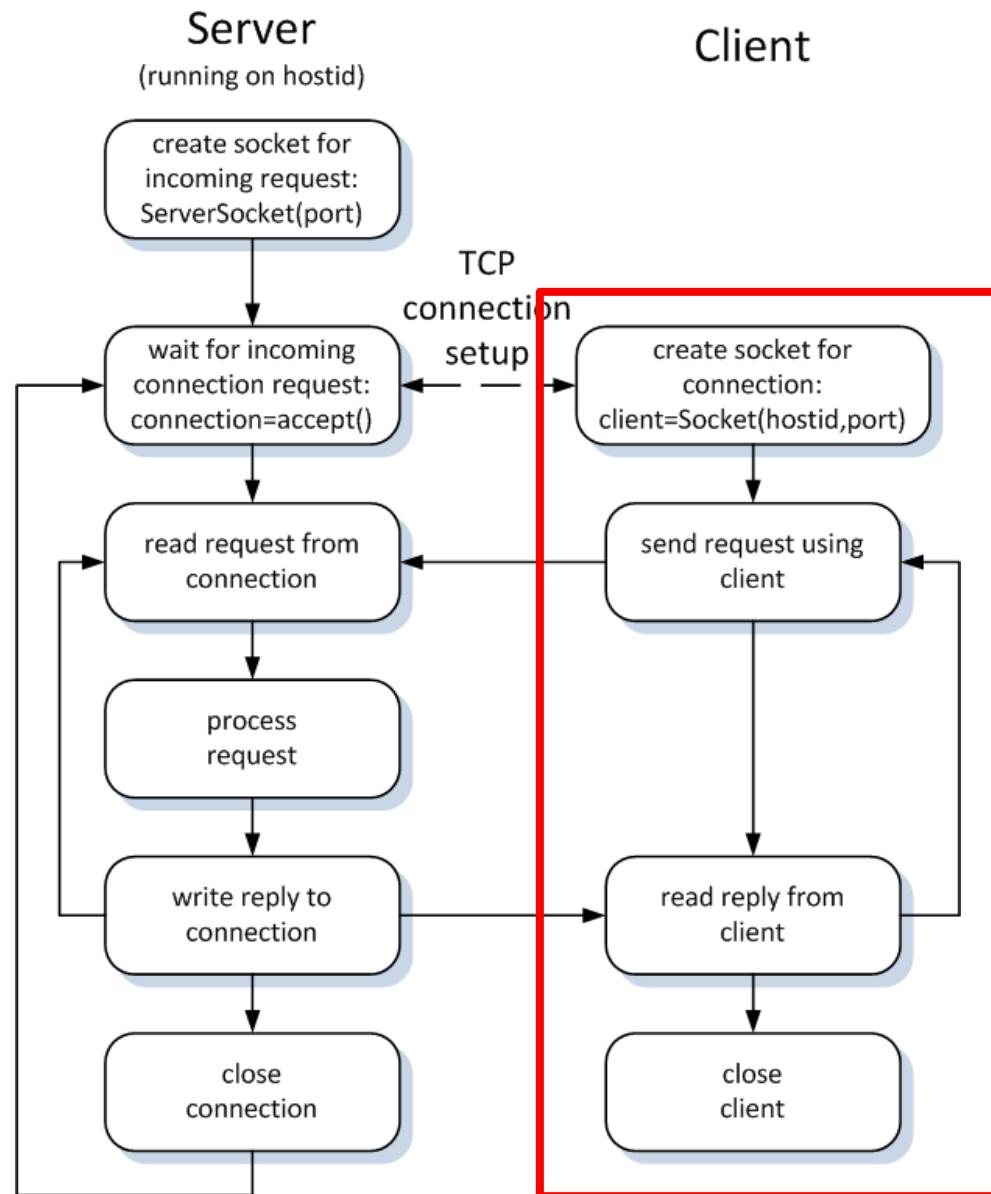
# Definitionen und Ablaufsübersicht

- **Socket:** Kommunikationsendpunkt in TCP/IP-Netzwerk (IP, Port).
- **ServerSocket:** Spezieller Socket um auf eingehende Verbindungen zu warten.



# Ablauf beim Client

- Um Daten verbindungsorientiert zu versenden, sind folgende Aktionen nötig:
  1. Socket erzeugen
  2. Socket an einen lokalen Port binden (Server-seitig)
  3. Verbindung mit Zieladresse herstellen
  4. Daten über Socket lesen/schreiben
  5. Socket schliessen

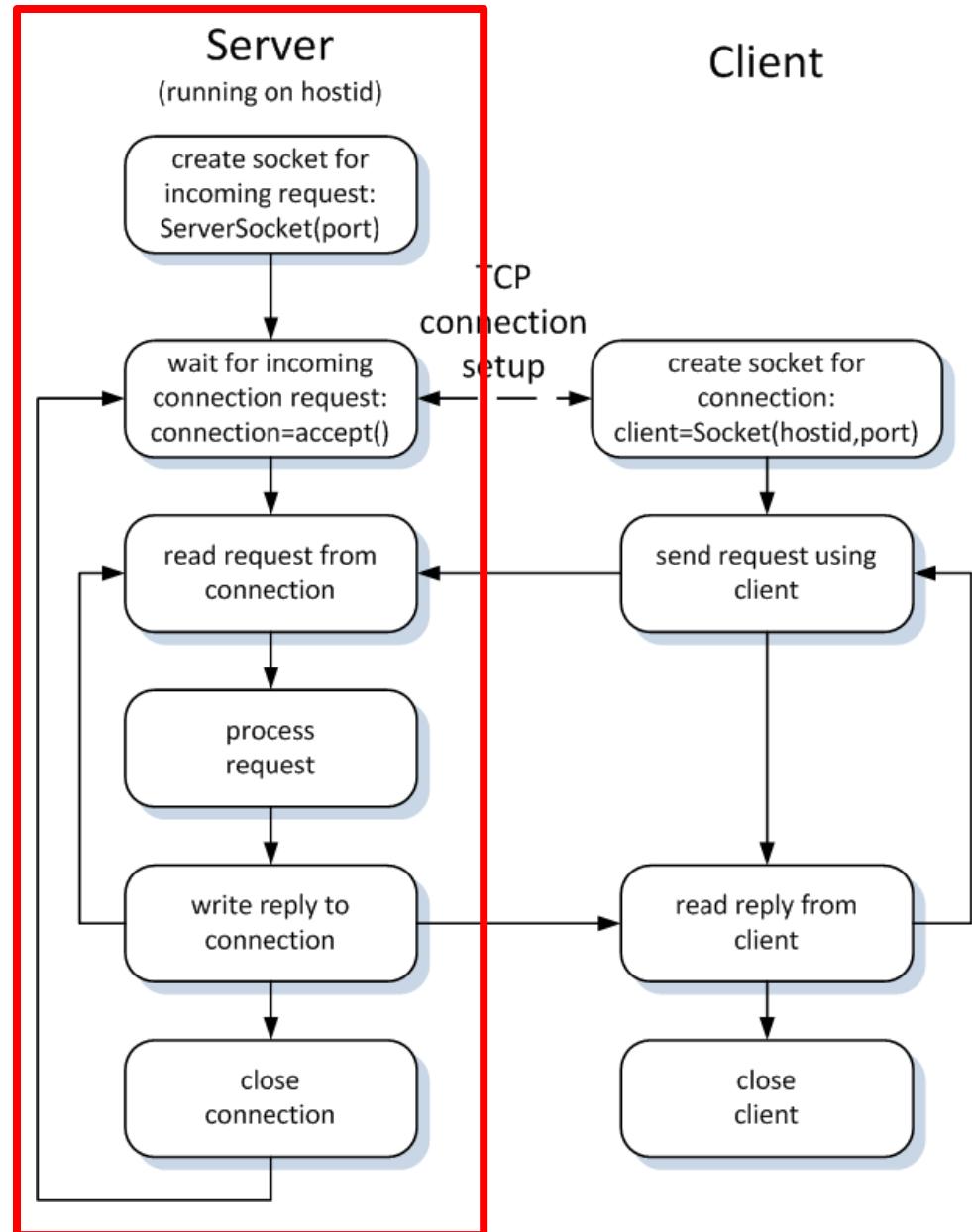


# Socket für den Server

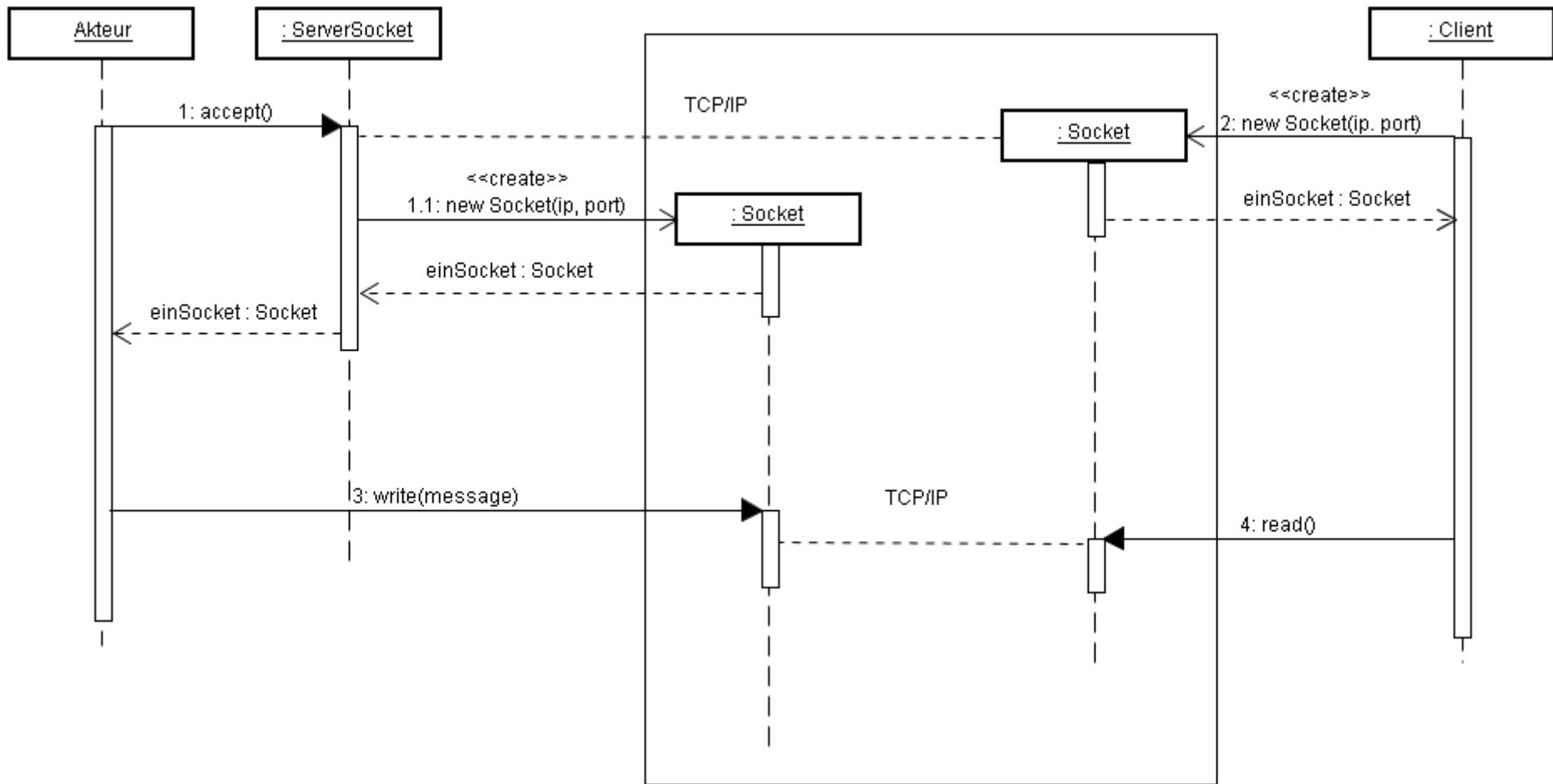
- Server hören an ihrem zugewiesenen Port auf Anfragen.
- Sockets bekommen als Argument Portnummer, zu der sich Clients verbinden können.
- Regeln für die Portnummer:
  - darf nicht in Benutzung sein,
  - kann nach Benutzung z.T. für einen bestimmten Zeitraum nicht verwendet werden (typisch: 4 Minuten)
  - falls < 1024 nur nutzbar durch Rootbenutzer bei Unix-Systemen (Linux, MacOS, etc.)

# Lebenszyklus eines TCP Servers

1. Server-Socket erzeugen.
2. Mit accept-Methode auf Verbindung warten.
3. Ein- und Ausgabestrom mit erhaltenem Socket verknüpfen.
4. Daten lesen und schreiben, entsprechend dem Kommunikationsprotokoll.
5. Stream von Client und Socket schliessen.
6. Bei Schritt 2 weitermachen oder Server-Socket schliessen.



# Sequenzdiagramm TCP mit Sockets



# Einfacher DayTime-Client mit Blocking I/O

```
static void getTime(String host, int port) throws IOException {  
    Socket socket = new Socket(host, port);  
  
    DataInputStream is;  
  
    is = new DataInputStream(socket.getInputStream());  
  
    byte[] bytes = is.readAllBytes();  
    socket.close();  
  
    String time = new String(bytes);  
    System.out.println(time);  
}
```

Socketverbindung starten

Eingabestrom für öffnen

Alle Bytes bis Ende der Verbindung einlesen

Socket schliessen

# Einfacher DayTime-Server (Blocking I/O, Single-Threaded)

```
public class SimpleDayTimeServer {  
    //...  
    public static void main(final String[] args) {  
        try {  
            final ServerSocket listen = new ServerSocket(1300);  
  
            while (true) {  
                try (final Socket client = listen.accept()) {  
                    final DataOutputStream dout =  
                        new DataOutputStream(client.getOutputStream());  
                    final Date date = new Date();  
                    dout.write((date.toString()).getBytes());  
                }  
            }  
        } catch (IOException ex) {  
            LOG.debug(ex.getMessage());  
        }  
    }  
}
```

Server Socket an Port 1300 binden

Warten auf Client

Zeit wird dem Client gesendet.

Try-with-resources wird geschlossen. Verbindung zum Client wird beendet.

# Warten auf Verbindungen

- Nur die accept-Methode der **ServerSocket** Klasse nimmt eine wartende Verbindung an und zwar genau eine Verbindung.
- Die accept-Methode blockiert den Programmablauf.
- Die Kommunikation läuft nicht über den Server-Socket, sondern über den von der accept-Methode zurückgegebenen Socket.
- Zur schnellen Wiederverfügbarkeit (neue Verbindungen) muss das Programm so schnell wie möglich zur accept-Methode zurückkehren.
- Warten auf Verbindungen und die Kommunikation mit dem Client sollte daher nebenläufig ausgeführt werden.

# Nicht-blockierender EchoServer

```
public class EchoServer {  
  
    private static final Logger LOG = LogManager.getLogger(EchoServer.class);  
  
    public static void main(final String[] args) throws IOException {  
        final ServerSocket listen = new ServerSocket(7777);  
        final ExecutorService executor = Executors.newFixedThreadPool(5);  
        while (true) {  
            try {  
                LOG.info("Waiting for connection...");  
                final Socket client = listen.accept();  
                final EchoHandler handler = new EchoHandler(client);  
                executor.execute(handler);  
            } catch (Exception ex) {  
                LOG.debug(ex.getMessage());  
            }  
        }  
    }  
}
```

Das Programm kann sofort auf den nächsten Verbindungsaufbau warten.

Executor für das parallele Handling.

Erstellung eines Echo Handlers, der die Kommunikation zum Client übernimmt.

Der EchoHandler wird in einem eigenen Thread ausgeführt.

# EchoHandler – Erzeugung

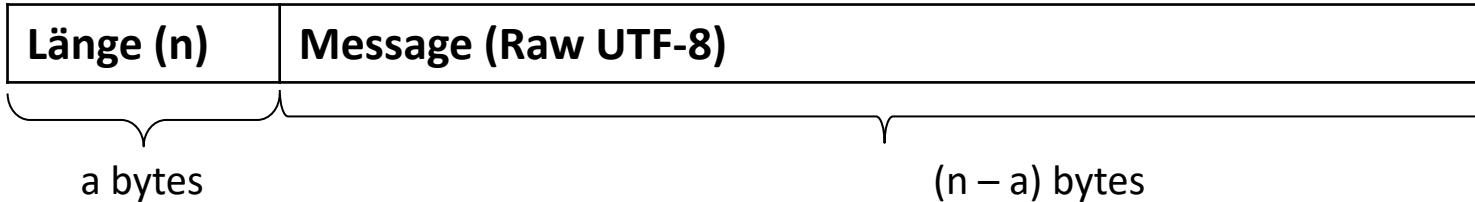
- Der EchoServer erstellt ein EchoHandler-Objekt und übergibt den Socket zur Clientverbindung.
- Sobald der Executor den EchoHandler mit einem Thread gestartet hat, läuft der Handler unabhängig von anderen laufenden Threads.
- Der Client bestimmt das Ende des Echo Handlings.

```
public class EchoHandler implements Runnable {  
  
    private static final Logger LOG =  
        LogManager.getLogger(EchoHandler.class);  
    private final Socket client;  
  
    public EchoHandler(final Socket client) {  
        this.client = client; ←  
    }  
}
```

Übergabe des Client Socket an  
den Echo Handler

# Kommunikationsprotokoll für den EchoServer

- Message mit Länge



```
private static void sendMessage(DataOutputStream os, String msg) throws IOException {
    byte[] bytesOut = msg.getBytes(StandardCharsets.UTF_8);
    os.writeInt(bytesOut.length); ← Schreibe Integer (plattformunabhängig)
    os.write(bytesOut); ← Schreibe Bytes
}

private static String getMessage(DataInputStream is) throws IOException {
    int length = is.readInt(); ← Lese Integer (plattformunabhängig)
    byte[] bytesIn = new byte[length];
    is.readFully(bytesIn); ← Lese Bytes

    return new String(bytesIn, StandardCharsets.UTF_8);
}
```

Annotations for the `sendMessage` method:

- An annotation "Schreibe Integer (plattformunabhängig)" with an arrow pointing to the line `os.writeInt(bytesOut.length);`.
- An annotation "Schreibe Bytes" with an arrow pointing to the line `os.write(bytesOut);`.

Annotations for the `getMessage` method:

- An annotation "Lese Integer (plattformunabhängig)" with an arrow pointing to the line `int length = is.readInt();`.
- An annotation "Lese Bytes" with an arrow pointing to the line `is.readFully(bytesIn);`.

# EchoHandler – Ausführung

```
@Override  
public void run() {  
    LOG.info("Connection to " + client);  
    try (OutputStream out = client.getOutputStream();  
         InputStream in = client.getInputStream()) {  
        DataInputStream dataIn = new DataInputStream(in);  
        DataOutputStream dataOut = new DataOutputStream(out);  
        while (true) {  
            String message = getMessage(dataIn);  
            sendMessage(dataOut, message);  
            dataOut.flush();  
        }  
    } catch (IOException ex) {  
        LOG.debug(ex.getMessage());  
    }  
}
```

Wartet auf Daten vom Client.

Das Programm wird nicht blockiert, da der EchoHandler in einem eigenen Thread läuft.

Wichtig: flush – damit die Daten den Prozess/Host verlassen.

# EchoClient

```
public static void main(final String[] args) {  
    BufferedReader userIn  
        = new BufferedReader(new InputStreamReader(System.in));  
    try (Socket socket = new Socket("localhost", PORT);  
        OutputStream out = socket.getOutputStream();  
        InputStream in = socket.getInputStream()) {  
  
        DataInputStream dataIn = new DataInputStream(in);  
        DataOutputStream dataOut = new DataOutputStream(out);  
  
        while (true) {  
            String messageOut = userIn.readLine();  
            sendMessage(dataOut, messageOut);  
            dataOut.flush();  
            String messageIn = getMessage(dataIn);  
            System.out.println(messageIn);  
        }  
    } catch (Exception ex) {  
        LOG.debug(ex.getMessage());  
    }  
}
```

Verwendet selbe  
Message-Parsing  
Methoden wie  
Server

# Klassenraumübung: EchoJava

Übung mittels Online-Programmierumgebung:

- <https://replit.com/@mbaettig/EchoJava>
  - **Kein HSLU-Dienst (benötigt separates Konto), erstellen Sie einen Fork.**
- Verwendung:
  - Server starten: Click auf RUN.
  - Client starten: Shell öffnen und eingeben «java EchoClient».

## Aufgabe:

- EchoMessages haben folgende Struktur und Inhalt:

Länge (n)	Message (Raw UTF-8)
	a bytes

- Message so erweitern, dass noch eine Integer (32-Bit) mitgesendet werden kann (nur Protokoll, API nicht anpassen. Sie können eine Konstante senden).
- Wie muss das Kommunikationsprotokoll angepasst werden?

# Exkurs: Non-Blocking I/O – Selector und Channel

## Ablauf:

- Erstelle einen Selector.
- Öffne einen Channel (ServerSocket oder Socket).
- Setze Sockets auf Non-Blocking und registriere gewünschte Events (z.B. accept).
- Warte auf Events und behandle diese.

## Initialisierung vom Server:

```
selector = Selector.open();
ServerSocketChannel socket = ServerSocketChannel.open();
ServerSocket serverSocket = socket.socket();
serverSocket.bind(new InetSocketAddress("localhost", 7777));
socket.configureBlocking(false);
socket.register(selector, SelectionKey.OP_ACCEPT);
```

# Exkurs: Non-Blocking I/O – Multiplexing

- Auf gleichem Selektor verschiedene Channels registrieren.

**Beispiel die Behandlung von Accept:**

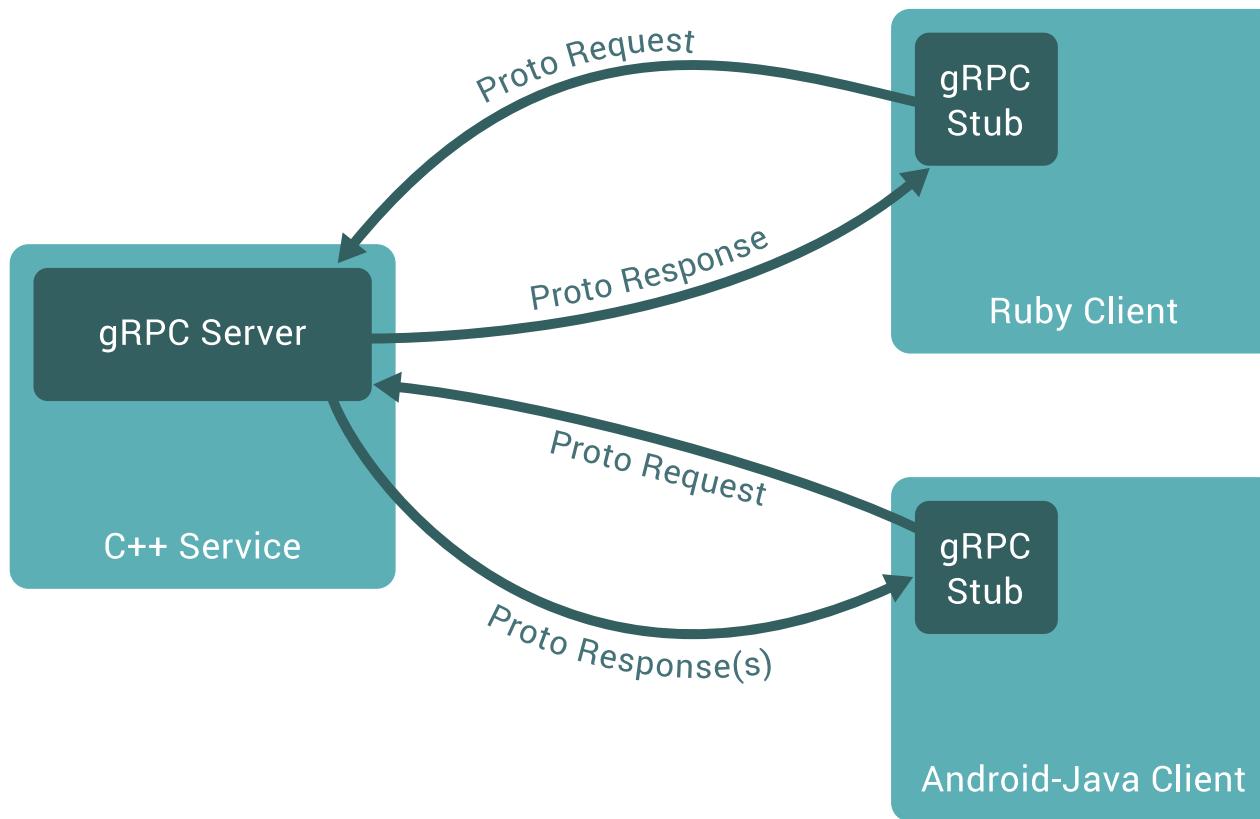
```
SocketChannel client = mySocket.accept();
client.configureBlocking(false);
client.register(selector, SelectionKey.OP_READ, new ClientState());
```

- Anschliessend auf Ereignisse warten («Event-Loop»):

```
while (true) {
    selector.select();
    Set<SelectionKey> selectedKeys = selector.selectedKeys();
    Iterator<SelectionKey> i = selectedKeys.iterator();
    while (i.hasNext()) {
        SelectionKey key = i.next();
        if (key.isAcceptable()) { handleAccept(socket); }
        else if (key.isReadable()) { handleRead(key); }
        else if (key.isWritable()) { handleWrite(key); }
        i.remove();
    }
}
```

# Remote-Procedure-Calls mit gRPC

- Plattform- und sprachübergreifendes RPC-Framework.
  - Java, C#, Python, Java, C++, Go, Node, ...
- Verwendet Google Protocol Buffers



# Definition von Messages mittels Protocol Buffers

- Protocol-Buffers sind Sprach- und Plattform neutral.
- Einsatzzweck: Serialisierung von strukturierten Daten.
- Dateiendung: .proto

```
syntax = "proto3";  
  
message SearchRequest {  
    string query = 1;  
    int32 page_number = 2;  
    int32 result_per_page = 3;  
}  
  
message SearchResult {  
    // weitere Message-Definitionen  
}
```

The diagram illustrates the structure of a Protocol Buffer definition file. It shows a snippet of .proto code with annotations explaining various parts:

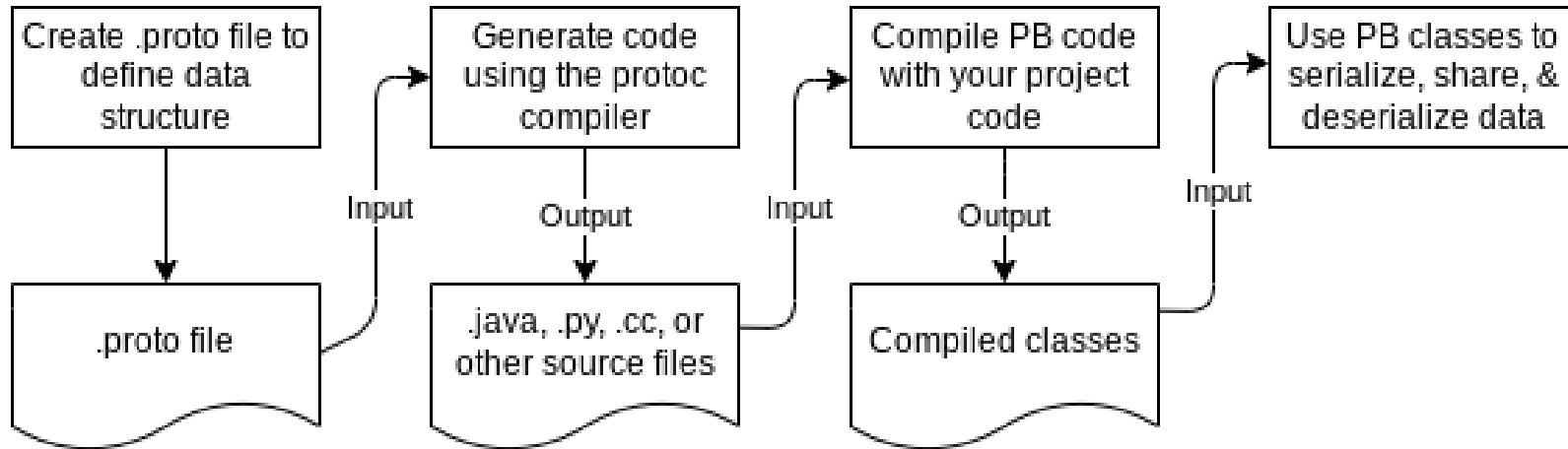
- An annotation for the `syntax = "proto3";` line points to the text: "Version: Falls nicht angegeben wird Version 2 verwendet."
- An annotation for the `message SearchRequest {` line points to the text: "Definition einer Messagestruktur."
- Annotations for the three field definitions (`string query = 1;`, `int32 page_number = 2;`, and `int32 result_per_page = 3;`) all point to the text: "Typisierte Felder."
- An annotation for the closing brace of the `SearchRequest` message points to the text: "Unique Ids für binäres Format."

# Type-Mapping: Protocol Buffer zu Zielsprache

.proto Type	C++ Type	Java/Kotlin Type	Python Type	Go Type	C# Type	PHP Type	Dart Type
double	double	double	float	float64	double	float	double
float	float	float	float	float32	float	float	double
int32	int32	int	int	int32	int	integer	int
int64	int64	long	int/long	int64	long	integer/string	Int64
uint32	uint32	int	int/long	uint32	uint	integer	int
uint64	uint64	long	int/long	uint64	ulong	integer/string	Int64
sint32	int32	int	int	int32	int	integer	int
sint64	int64	long	int/long	int64	long	integer/string	Int64
fixed32	uint32	int	int/long	uint32	uint	integer	int
fixed64	uint64	long	int/long	uint64	ulong	integer/string	Int64
sfixed32	int32	int	int	int32	int	integer	int
sfixed64	int64	long	int/long	int64	long	integer/string	Int64
bool	bool	boolean	bool	bool	bool	boolean	bool
string	string	String	str/unicode	string	string	string	String

# Arbeiten mit Protocol Buffers

- Verwendung von protoc um Messages in Zielsprache zu generieren:



```
protoc --java_out=DST_DIR path/to/file.proto
```

Für **Java** generiert der protoc-Compiler:

- ⇒ eine .java-Data mit einer Klasse pro Messagetyp.
- ⇒ sowie eine speziellen Builder-Klasse zur Erzeugung von Messageinstanzen.

# Service-Definitionen

```
syntax = "proto3";
```

Service: besteht aus einem oder mehreren RPC.

```
service SearchService {  
    rpc Search(SearchRequest) returns (SearchResponse);  
}
```

```
message SearchRequest {  
    ...  
}
```

RPC: Name mit Input-Parameter und Rückgabewert.

```
message SearchResponse {  
    ...  
}
```

# Echo mit gRPC: Service und Nachrichten

- Definition eines Kommunikationsprotokolls mit zwei Nachrichten

```
syntax = "proto3";

service EchoService {
    rpc echo(EchoRequest) returns (EchoResponse);
}

message EchoRequest {
    string message = 1;
}

message EchoResponse {
    string message = 1;
}
```

# Echo mit gRPC: Client

```
public class EchoClient {  
    private static final int PORT = 5001;  
  
    public static void main(String[] args) {  
        ManagedChannel channel =  
            ManagedChannelBuilder.forAddress("localhost", PORT)  
                .usePlaintext()  
                .build();  
  
        EchoServiceGrpc.EchoServiceBlockingStub stub =  
            EchoServiceGrpc.newBlockingStub(channel);  
  
        Echo.EchoResponse echo = stub.echo(Echo.EchoRequest.newBuilder()  
            .setMessage("test").build());  
        System.out.println(echo.getMessage());  
        channel.shutdown();  
    }  
}
```

# Echo mit gRPC: Server

```
public class EchoServer {  
    private static final int PORT = 5001;  
  
    public static class EchoService extends EchoServiceGrpc.EchoServiceImplBase {  
        public void echo(Echo.EchoRequest request,  
                         StreamObserver<Echo.EchoResponse> observer) {  
            Echo.EchoResponse response = Echo.EchoResponse.newBuilder()  
                .setMessage(request.getMessage()).build();  
            observer.onNext(response);  
            observer.onCompleted();  
        }  
    }  
  
    public static void main(String[] args) throws IOException, InterruptedException {  
        Server srv = ServerBuilder.forPort(PORT).addService(new EchoService()).build();  
        srv.start();  
        System.out.println("Server started, listening on " + PORT);  
        srv.awaitTermination();  
    }  
}
```

onNext: gibt Response zurück  
onComplete: RPC ist fertig

Starte Server

Füge Service dem Server hinzu

Warte auf Ende

# Exkurs: Customized-Echo mit gRPC

**Ziel:** Pro Verbindung soll der Client ein Prefix setzen können, welches der Server jeweils der Antwort voranstellt.

**Variante 1:** Streaming gRPC mit genereller Request-Struktur:

```
service CustomizedEcho {  
    rpc myCall(stream Request) returns (stream Response);  
}  
  
message Request {  
    int32 action = 1;    // 01: set prefix | 02:echo  
    string content = 2; // message or rprefix  
}
```

# Exkurs: Customized-Echo mit gRPC

**Ziel:** Pro Verbindung soll der Client ein Prefix setzen können, welches der Server jeweils der Antwort voranstellt.

**Variante 2:** Mittels einer «Session»

```
service CustomizedEcho {  
  
    // open connection  
    rpc open(SessionRequest) returns (SessionResponse);  
  
    // send new prefix  
    rpc setPrefix(SetPrefixRequest) returns (SetPrefixResponse);  
  
    // request a prefixed echo  
    rpc echo(EchoRequest) returns (EchoResponse);  
  
    // close connection  
    rpc close(CloseRequest) returns (CloseResponse);  
  
}
```

# Zusammenfassung

- Remote-Procedure-Calls sind synchrone entfernte Aufrufe.
- Umsetzung mittels Kommunikationsprotokoll.
  - Definiert Anforderungen an Kanal, generelle Struktur und Nachrichten.
- Viele verschiedene Möglichkeiten Verbindungen parallel zu verarbeiten.
- ServerSocket können Verbindungen akzeptieren (accept).
- Der Server selbst ist nicht aktiv, sondern horcht an seinem zugewiesenen Port und der IP-Adresse auf Client-Anfragen.
- gRPC ist ein modernes Framework für RPC.

# **Fragen?**

Verteilte Systeme und Komponenten

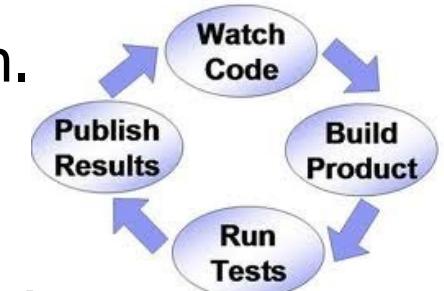
# **Professionelle Entwicklung mit Java**

**nach den Prinzipien der Continuous Integration (CI)**

Roland Gisler

# Entwicklungsprozess: Ziele im Modul VSK

- Entwicklung eines Java-Projektes aus mehreren Komponenten (bzw. Modulen), im Team, in einer vorbereiteten Projektstruktur.
  - Projekte sind im VCS für Sie bereits erstellt!
- Verwendung eines Versionskontrollsystems (VCS)
  - **git** und **GitLab**, auf Enterprise Lab zur Verfügung gestellt.
- Einsatz einer zeitgemäßen, integrierten Entwicklungsumgebung
  - NetBeans, Eclipse, IntelliJ, ... – Sie haben die Wahl!
- Continuous Integration (CI)
  - Automatisierter Buildprozess mit Apache Maven.
  - Automatisierte Unit-Tests mit JUnit/AssertJ.
  - Zentrale Codeverwaltung mit Git und GitLab.
  - Zentraler Buildserver mit Jenkins (und GitLab CI).



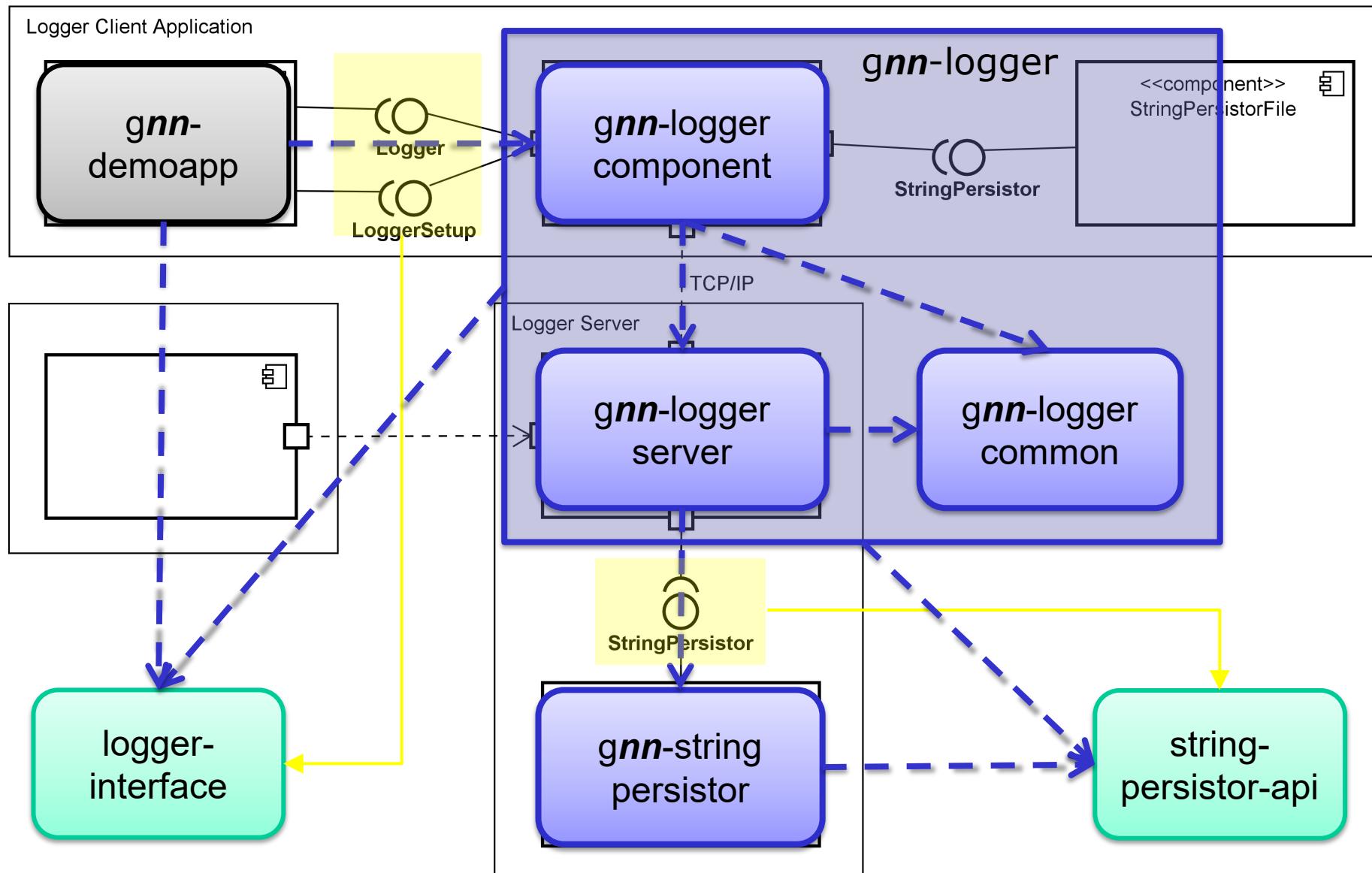
# Software Stack

- Oracle Java JDK (inkl. JRE), Version **17.0.4.1 LTS**
  - Wichtig: Wir verwenden alle diesen **einheitlichen** Stand!
  - Nur so ist die Austauschbarkeit zwischen den Teams gegeben.
  - Ältere Java-Versionen werden nicht mehr unterstützt!
- Apache Maven 3.8.6
  - In IDE's enthalten, aber freistehende Installation empfohlen!
  - Konfiguration (`settings.xml` im `$HOME/.m2`) wichtig!
- git 2.37.3
  - Empfehlung für zusätzlichen Client (IDE-unabhängig):  
SmartGit von <http://www.syntevo.com/smartgit/>
- Anleitung für SW-Migration- und Installation: siehe ILIAS!
- SW-Archiv auf SWITCHdrive: <http://bit.ly/2OH3Uhh>

# Java 17 LTS

- Im September 2021 wurde wie geplant der zweite LTS (nach neuem Releaseplan) wenige Tage vor Semesterstart veröffentlicht.
  - Dieser LTS löst offiziell die bisherige LTS-Version **11(.0.13)** ab.
  - Oracle hat versprochen, nun im Rhythmus von zwei (oder drei) Jahren (2018, 2021, 2023,...) jeweils wieder LTS Versionen (11, 17, 21, ...) zu veröffentlichen, welche für den produktiven Einsatz vorgesehen sind.
- Darum nutzen wir im HS22 alle einheitlich die Java Version **17.0.4.1 LTS** – was besonders im Hinblick auf die Austauschbarkeit von Modulen und Komponenten als Binaries wichtig ist.

# Projektstruktur – Logger System (mit DemoApp)



# **Projekte, Komponenten, Libraries, Applikationen...**

- ***gnn-demoapp*** und ***gnn-loggerserver*** sind Applikationen.
  - enthalten typisch eine `main()`-Methode.
- ***gnn-loggercomponent*** und ***gnn-stringpersistor*** sind Komponenten, welche je ein Interface implementieren.
- ***loggerinterface*** und ***stringpersistor-api*** sind Schnittstellen.
  - noch spezifischer: API – Application Programmers Interface
- ***gnn-loggercommon*** ist eine reine Library zur Modularisierung.
  
- Alles sind Projekte, wobei ***gnn-logger*** drei Submodule (**component**, **server** und **logger**) hierarchisch aggregiert.
  - Feature von Apache Maven, erlaubt es u.a. die Module logisch als **eine** einzige Releaseeinheit zu behandeln → Input folgt.

# VSK-Projekte auf HSLU-GitLab SCM

- Pro Gruppe vier git-Repositories, mit vorbereiteten Projekten
  - Bekannte Struktur und Build mit Apache Maven.
- Basierend auf dem von OOP und AD bekannten Java-Template.
- Gruppe für Moduldurchführung, Rechte pro Benutzer auf Gruppe.
  - **VSK-22HS01/gnn – VSK, HerbstSemester 2022, Kurs 01**
- Struktur/Namensgebung der Projekte (*nn*=Gruppennummer):
  - **gnn-demoapp** – Demo-Applikation für Integration und Test.
  - **gnn-stringpersistor** – Komponente für Dateispeicherung
  - **gnn-logger** – Logger Server, Multimodulprojekt, enthält:
    - **loggercomponent** – Logger-Komponente.
    - **loggerserver** – Logger Serveranwendung.
    - **loggercommon** – Gemeinsame Library (optionale Nutzung).
  - **gnn-documentation** – Für Dokumentation (kein Build/CI).

# Schedule der EP-Inputs

- Teil 1 (SW02):
  - [Versionskontrollsysteme mit git](#)
  - [Buildautomatisation mit Apache Maven](#)
- Teil 2 (SW03):
  - [Dependency Management mit Apache Maven](#)
  - [Buildserver Technologien mit GitLab CI und Jenkins](#)
- Teil 3 (SW07):
  - [Prinzipien der Continuous Integration \(CI\)](#)

# Demo

- Projekte auf GitLab:

<https://gitlab.enterpriselab.ch/vsk-22hs01>



# Wichtige Hinweise / Empfehlungen

- **git**-Client(s) bitte mit **Klarnamen** (Vorname Name) und korrekter EMail-Adresse konfigurieren, **bitte keine Synonyme** verwenden.
  - siehe Konfiguration in: `~/.gitconfig` (im User-\$HOME).
- Ziel: Im VCS befindet sich immer ein kompilierbarer, lauffähiger Stand aller Projekte.
  - Vor dem Commit mindestens kompilieren und testen!
  - Am einfachsten über `mvn package` oder `mvn verify`.
- Beim Commit schreiben wir **immer** einen aussagekräftigen Kommentar, wenn vorhanden **mit** einer **Issue-Number** (z.B. #23).
  - Commits werden dadurch in GitLab dem Issue zugeordnet, was die Nachvollziehbarkeit extrem erhöht!

# Aufträge

- Klonen sämtlicher Projekte (einzelne, jedes Teammitglied).
  - Integration in die IDE, Verständnis der Funktionsweise.
- Ziel für Heute: Alle sind bereit, um am Logger-Projekt aktiv entwickeln zu können.
- Hilfsmittel für Installation (Java, Maven etc.):  
Anleitung «OOP\_JavaDevelopmentManual\_jdk17.pdf»
  - Erweiterte und aktualisierte Fassung, bekannt aus OOP & AD.
  - Zu finden im ILIAS unter:

**Fragen?**

Verteilte Systeme und Komponenten

# **Versionskontrollsysteme**

**Source Code Management (SCM)**

**Version Control System (VCS)**

Roland Gisler

# Inhalt

- Grundlagen – was ist ein Versionskontrollsystem (VCS)?
- Wie arbeitet man mit einem VCS?
- Was gibt es für unterschiedliche Konzepte?
- Verschiedene Beispiele / Produkte
- Verschiedene Benutzerschnittstellen
- Konkret: Arbeit mit git und HSLU GitLab
- Zusammenfassung

# Lernziele

- Sie kennen die Aufgaben eines Versionskontrollsystems und können grundlegend damit arbeiten.
- Sie kennen die verschiedenen Konzepte und Arten von Versionskontrollsystemen.
- Sie können mit verschiedenen (Client-)Werkzeugen von Versionskontrollsystemen alleine und im Team arbeiten.

# **Grundlagen**

# Ziel und Zweck von VCS

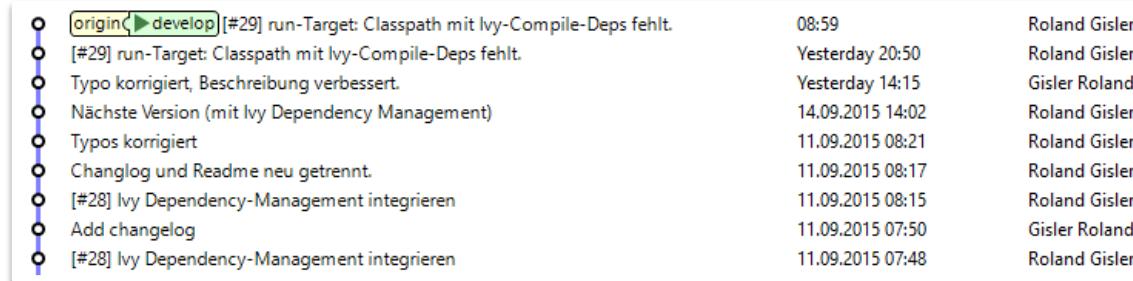
- Versionskontrollsysteme werden vorwiegend (aber nicht nur) in der Softwareentwicklung genutzt.
  - Alternativ als SCM (source control management) bezeichnet.
- Sie sind für eine eher «technisch» orientierte Nutzung konzipiert.
- Das Hauptziel ist:

**Die Abfolge aller Bearbeitungsschritte an  
Artefakten benutzerbezogen und detailliert zeitlich  
nachvollziehen zu können.**

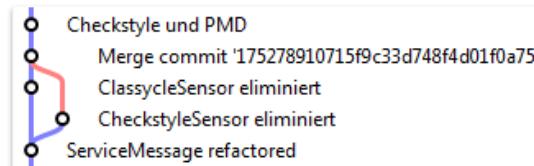
- Vereinfacht formuliert:  
**«Wer hat wann und aus welchem Grund in welcher Datei welche Änderungen vorgenommen?»**

# VCS - Grundlagen

- System, welches die zeitliche Entwicklung von Artefakten festhält und jederzeit einen Rückgriff auf alte Änderungsstände erlaubt.
  - so genannte Revisionen (Überarbeitungsstände).



- Ist für konkurrenzierende Zugriffe und Modifikationen ausgelegt.
  - Teamarbeit auf gemeinsamen Quellen.
  - Automatisches Merging bei Konflikten (soweit möglich).
  - Entweder zentrale Datenhaltung oder auch verteilt!
  - **Kein** Ersatz für fehlende Koordination!



# Abgrenzung zu Filesharing-Diensten

- Synchronisieren Dateien (inkl. Verzeichnisstrukturen) zwischen verschiedenen Rechnern und/oder der Cloud.
  - Typisch automatisch und im «Hintergrund».
- Artefakte werden bei **jeder** Veränderung bzw. bei jedem Speichern **sofort** übertragen.
  - Jedoch keine Garantie, kann auch verzögert erfolgen.
- Erzeugen teilweise auch Revisionen (pro Artefakte).
  - Meist stark beschränkte Anzahl an Revisionen.
- Beispiele: Dropbox, OneDrive, SWITCHdrive, Tresorit etc.

# VCS – Abgrenzung und Fokus

- Versionskontrollsysteme sind **weder** Backupsystem noch Filesharing-Dienst (OneDrive etc.) und auch kein DMS.
- Bewusster Umgang: **Sie** als Entwickler\*in **bestimmen, wann** eine **neue Version festgeschrieben** wird!
- Versionskontrollsysteem haben einen anderen Fokus:  
**Nachvollziehbarkeit von Änderungen**
  - Workflows und Koordination in (verteilten) Teams.
- Änderungen werden als sogenannte «Changesets» innerhalb einer Transaktion gespeichert.
  - **1..n** Dateiartefakte, die von einem konsistenten Zustand  **$z_1$**  zum nächsten konsistenten Zustand  **$z_2$**  führen.

# **Arbeit mit einem VCS**

# Grundlegende Arbeit mit einem VCS (Zentral)

Workspace User 1      Workspace User 2      Workspace User 3

Lokale Kopien

- **checkout**
- **update**
- **log**
- **diff**
- **commit**



Zentrales VCS

# Grundlegende Arbeit mit einem VCS

- **checkout** - von einem Projekt eine lokale Arbeitskopie erstellen.
    - Auf dieser Kopie wird dann gearbeitet.
  - **update** - Änderungen Dritter in lokaler Arbeitskopie aktualisieren.
    - Periodisch oder nach Bedarf, aber unbedingt vor einem Commit.
  - **log** - Bearbeitungsgeschichte der Artefakte ansehen.
    - Wer hat wann und warum welche Artefakte geändert?
  - **diff** - verschiedene Revisionen miteinander vergleichen.
    - Fremde (oder auch eigene) Änderungen nachvollziehen.
  - **commit** - Artefakte in das Repository schreiben (auch: **checkin**).
    - Lokale Veränderungen in das Repository zurückschreiben.
    - Veränderungen werden für Dritte bei einem Update sichtbar.
- Commit **immer mit Kommentar\*** und ggf. einer Issue-Number!

# **Simple lokale Versionsverwaltung mit git**

- Wenn Sie **git** installiert haben, funktioniert das auch rein **lokal!**

**git init**

Erzeugt ein neues Repo.

**git add .**

Bezieht (neue) Dateien mit ein.

**git commit -a -m "Message"**

Schreibt geänderte fest (Changeset).

**git log**

Zeigt die Geschichte des Repos an.

**git status**

Zeigt den lokalen Änderungsstatus an.

→ Natürlich ist das mit einem grafischen Client alles viel einfacher.

# Demo / Screencast's

- git mit einem lokalen Repository einsetzen:

**EP\_11\_SC01\_GitLokal.mp4**

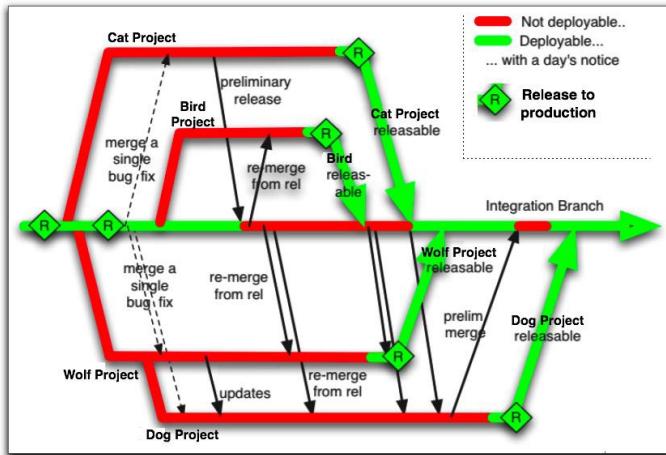


# Tagging – markieren von Revisionsständen

- Markieren eines bestimmten Revisionsstandes mit einem Namen.
  - Eine textuelle, einfach identifizierbare Marke oder Version.  
→ erleichtert die spätere Selektion dieses Standes.
  - Beispiele: **1.4.3**      **1.6.1**      **1.5.2beta**
- Sehr nützlich bei einem **Release** eines Produktes.
  - Als Markierung von Meilensteinen, Testversionen und/oder Auslieferungsständen, für einfachere Selektion.
- Tagging wird von den Systemen sehr unterschiedlich realisiert:
  - Nur eine Markierung in jeder Datei. (z.B. bei cvs)
  - Kopie aller Artefakte in ein anderes Verzeichnis. (z.B. bei Subversion)
  - Identifikation eines Änderungsstandes des gesamten Dateisystems. (vereinfacht formuliert, z.B. bei git, hg)

# Branching

- Voneinander getrennte Entwicklungszweige, lokal oder zentral, für:
  - Prototypen, Tests und/oder Experimente.
  - Bugfixing bereits geschlossener Versionen.
  - Professionelle (Änderungs-)Workflows (z.B. [GitFlow](#) etc.)



- Wenn es sich nicht um «Wegwerf»-Entwicklungen handelt ist später ein Merging möglich/notwendig.
  - Läuft idealerweise (halb-)automatisch ab.
  - Kann aber auch sehr aufwändig (manuell) werden.

# Was verwaltet man in einem VCS?

- Ausschliesslich **Quell**-Artefakte einchecken!
  - Sourcen, Konfigurationen, ggf. Dokumentation
  - Beispiele für Java: `*.java`, `*.properties` etc.
- Aber **NIE** Artefakte die generiert bzw. erzeugt werden können
  - Beispiele: `*.class`-Dateien, erzeugte HTML-Reports etc.
  - Konkret: `./target/**` **NIE** einchecken!
- In der Regel bieten die VCS-Systeme Hilfen an, um ausgewählte Verzeichnisse/Dateien automatisch zu ignorieren.
  - Beispiel für git: `.gitignore`-Datei mit Filtereinträgen.
  - In unseren HSLU-Projekten bereits vorbereitet.



# **Konzeptionelle Unterschiede**

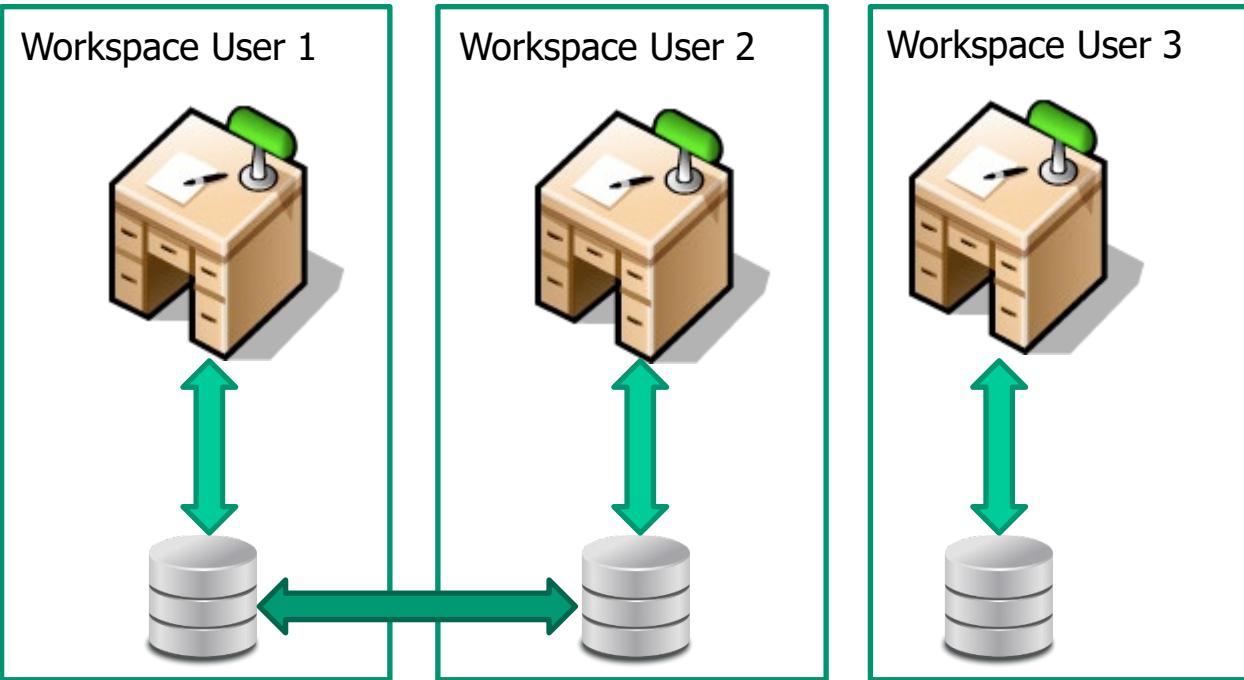
# Was gibt es für Unterschiede?

- VCSs werden in banalen Vergleichen häufig «in einen Topf» geworfen, zumal die Befehle zum Teil identisch lauten.
  - Gefährliche Vereinfachung, weil zwar die Ziele identisch, aber die Lösungswege grundverschieden sind.
- Beispiele für Konzeptunterschiede:
  - **Zentrale** oder **verteilte** Systeme.
  - **Optimistische** und **pessimistische** Lockverfahren.
  - Versionierung auf der Basis einer **Datei**, der **Verzeichnisstruktur** (FS) oder der **Änderung** (changeset).
  - **Transaktionsunterstützung** (vorhanden oder nicht).
  - Verschiedene **Zugriffsprotokolle** und Sicherheitsmechanismen.
  - **Integration** in Webserver (vorhanden oder nicht).

# Verteiltes VCS - Konzept

Die «klassischen» Befehle wirken sich **nur** auf das lokale Repository aus!

**Lokale** Repos  
(`.git`-Verzeichnis)



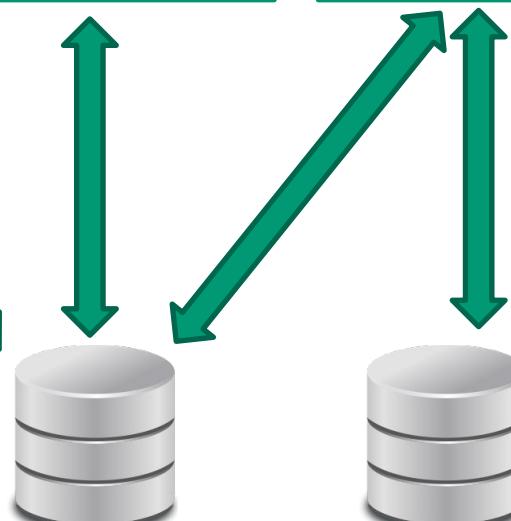
Um die lokal erzeugten Revisionen in entfernten Repositories zu Verfügung zu stellen gibt es **zusätzliche** Befehle.

Entfernte Repos, z.B.

`git@gitlab.enterpriselab.ch:vsk-22hs01/g01-demoapp.git`

- `checkout`
- `update`
- `log`
- `diff`
- `commit`

- `clone`
- `fetch`
- `pull`
- `push`



# Verteilte Versionsverwaltung mit git

- Wenn wir verteilt bzw. mit einem zentralen, gemeinsamen Repository arbeiten, müssen wir dieses **zusätzlich** Synchronisieren!

**git clone <url>**

Lokales klonen (kopieren) einen entfernten Repos und Workspace einrichten (einmalig, Initialisierung).

**git pull**

Änderungen vom entfernten Repo lokal nachführen und in Workspace mergen (fetch/merge).

**git push**

Lokale Änderungen (im Repo) in entferntes Repo übertragen.

# Demo / Screencast's

- git mit einem zentralen Repository einsetzen:

**EP\_11\_SC02\_GitRemote.mp4**



# **Drei konkrete Produkte**

# CVS – nicht mehr zeitgemäss



- (Ur-)altes Ur-Versionskontrollsystem
  - Sehr robust, Verbreitung stark abgenommen, gilt als «veraltet».
  - Clients und Integrationen nehmen stark ab.
  - Basiert auf einem zentralen Server.
- Vorteile
  - Sehr stabil, nur noch sehr wenig Fehler.
  - Einfache Anwendung, gut überschaubar.
  - Repository-Konzept strukturell vorgegeben.
- Nachteile
  - Nur dateibasierend (Verzeichnisstruktur **nicht** versioniert).
  - Unterscheidung zwischen Text- und Binärdateien.
  - Ablage von Binärdateien sehr ineffizient (Platzintensiv).
  - **Keine Transaktionen!**

- Wurde offiziell als CVS-Nachfolger eingeführt.
  - Popularität sinkt derzeit sehr stark.
  - Aber noch immer gute und breite Unterstützung.
  - Basiert auf einem zentralen Server.
- Vorteile
  - **Transaktionsorientiert** (Commit in Transaktion, LUW).
  - Versioniert die ganze **Verzeichnisstruktur**.
  - Optimierte/effiziente Speicherung und Übertragung.
  - Repositorystruktur frei wählbar (für Experten flexibler).
  - Integration in / mit Webserver möglich.
- Nachteile
  - Repositorystruktur frei wählbar (für Anfänger schwieriger).
  - Branching und Tagging technisch eigentlich Kopien/Links.

# git – sehr Populäres, zeitgemäßes System



- Verteiltes System, wird u.a. für Linux-Codeverwaltung verwendet.
  - Entwickelt von Linus Torvalds, sehr flexibles Konzept.
  - Ausgelegt für massives, billiges Branching.
  - Operationen möglichst 'billig' und schnell, weil primär **lokal**.
- Vorteile
  - Verteiltes System, beliebig viele Server / Repos möglich.
  - Sehr flexibel, auch rein **lokal** (genial!) einsetzbar.
  - Skaliert (Funktionsumfang, Verteilung, Grösse).
  - Meist mit Integration in/mit zusätzlichen Web-Applikationen.
- Nachteile
  - Erfordert bei verteiltem Einsatz ein solides Konzept, das organisiert und verstanden werden muss.
  - Für Einsteiger schwieriger, weil sehr mächtig / viele Funktionen.

# **Benutzerschnittstellen (Clients)**

# Verschiedene Benutzerschnittstellen

Im wesentlichen gibt es drei verschiedene Varianten:

- Kommandozeile in der Shell.
    - Für einfache Befehle (z.B. `git clone`) sehr effizient.
    - Sehr effizient (wenn man es beherrscht), aber auch kompliziert.
  - Spezialisierte VCS-GUI-Clients.
    - In der Regel etwas einfacher zu Bedienen.
    - Unterstützen das jeweilige VCS optimal.
    - Qual der (Aus-)Wahl! (siehe Quellen und Links)
  - Integrierte VCS-Clients, z.B. in der Entwicklungsumgebung.
    - Eigentlich sehr bequem (alles in einer Applikation).
    - Aber je nach Integration etwas verwirrend.
- Nutzen Sie individuell die Ihnen zusagende Schnittstelle!

# **HSLU - GitLab**

# GitLab Enterprise auf EnterpriseLab – 15.3.x

The screenshot shows the GitLab Enterprise Edition 15.3.1-ee interface on the left and a browser window displaying a project details page on the right.

**GitLab Enterprise Edition 15.3.1-ee (Left):**

- Header: Help > Help
- Title: GitLab Enterprise Edition 15.3.1-ee
- Text: GitLab is open source software to collaborate on code. Manage git repositories with fine-grained access controls that keep your code secure. Perform code reviews and enhance collaboration with merge requests. Each project can also have an issue tracker and a wiki. Used by more than 100,000 organizations, GitLab is the most popular solution to manage git repositories on-premises. Read more about GitLab at [about.gitlab.com](#).
- Link: Check the current instance configuration
- Text: Visit [docs.gitlab.com](#) for the latest version of this help information with enhanced navigation, discoverability, and readability.
- Section: GitLab Docs
- Text: Welcome to GitLab documentation. Here you can access the complete documentation for GitLab, the single application for the [entire DevOps lifecycle](#).
- Section: Overview
- Text: No matter how you use GitLab, we have documentation for you.

**Browser Window (Right):**

URL: [https://gitlab.enterpriselab.ch/oop/oop\\_maven\\_template](https://gitlab.enterpriselab.ch/oop/oop_maven_template)

Project Details:

- Project Name: oop\_maven\_template
- Description: Template für einfache Java-Projekte mit Maven-Build.
- Owner: Hochschule Luzern Modul OOP
- Statistics: Unstarred (1), Forked (0), SSH URL: git@gitlab.enterpriselab.ch:oop/oop\_maven\_template
- File List: Files (236 KB), Commits (48), Branches (2), Tags (8), Readme, LICENSE, CI configuration, Add Changelog, Add Contribution guide
- Commit Log:

  - #9 Hamcrest-Exclusion bei JUnit entfernt. (Roland Gisler, a week ago)
  - ... (other commits listed)

- File List (Detailed):

Name	Last commit	Last update
config	Vorbereitung für Release 1.5 (18FS)	a week ago
src	Formatierung	a week ago
.classpath	Eclipse-Projektkonfig ergänzt	a year ago
.gitignore	Initial Commit	a year ago
.gitlab-ci.yml	CI auf GitLab aktiviert	a year ago
.project	Eclipse-Projektkonfig ergänzt	9 months ago
LICENSE.txt	Vorbereitung für Release 1.5 (18FS)	a year ago
README.md	Vorbereitung für Release 1.5 (18FS)	a week ago

- <https://gitlab.enterpriselab.ch/vsk-22hs01>

# GitLab – Funktionalität

- GitLab stellt auf der Basis von git-Repositories eine komplette Codehosting-Plattform zur Verfügung.
  - Codeverwaltung in Projekten und Gruppen.
  - Planung von Milestones, Issue-Tracking, Taskboard.
  - Einfache Website (Markup-Readme) und Wiki.
  - Direktes bearbeiten von Artefakten über Web-GUI.
  - Merge-Requests, Pull-Requests, CI-Dienste etc.
- Vergleichbar mit bekannten Diensten wie GitHub, BitBucket, Sourceforge etc.
- Hinweis zum Einsatz in VSK: Wir nutzen GitLab für die **Planung, Controlling und die Codeverwaltung**. Die CI-Dienste werden durch eine zusätzliche Infrastruktur angeboten.

# GitLab – Zugriff, Protokolle und Clients

- Zentrale Git-Repositories mit Web-GUI, hosted im EnterpriseLab
  - Läuft in der DMZ, somit direkt (ohne VPN) erreichbar.
- URL: <https://gitlab.enterpriselab.ch/>
  - Login mit ELAB-Account! → <https://eportal.enterpriselab.ch/>
- Zugriff auf das Repository
  - **https**: Einfach, Passwort wird ggf. in jedem Client hinterlegt.
  - **ssh**: Elegant, Public-/Private-Key Infrastruktur notwendig.
- Direktzugriff auf die Git-Repositories mittels:
  - NetBeans / Eclipse - EGit-Client (vorinstalliertes Plugin).
  - SourceTree - Für Windows und Mac.
  - Git in Konsole/Shell - Für Puristen, sehr effizient.
  - **SmartGit** - Sehr guter Client (Multiplattform), **Empfehlung!**

# **Wichtige Hinweise**

# Wichtige Empfehlungen für die Arbeit mit VCS/git



- Vor Arbeitsbeginn und vor jedem **commit/push** ein **pull** machen
  - Aktuellsten Stand für Weiterarbeit übernehmen.
  - evt. parallele Änderungen prüfen und einarbeiten (merge).
- **commit immer** mit einem **aussagekräftigen** Kommentar!
  - Woran bzw. warum hat man etwas geändert?
  - Wenn vorhanden Link auf Issue!! (#nn, z.B.: #18)
  - Zeitnah (sofort!) ein **push** in das zentrale Repo!
- Generierte, automatisch erstellbare Artefakte **NIE** einchecken
  - **.gitignore**-Datei (pro Verzeichnis) mit Einträgen was ignoriert werden soll, ist bereits vorkonfiguriert, ggf. ergänzen!
- Wer noch keine Erfahrung hat: *Üben, üben, üben!*
  - Am Besten zuerst Lokal, dann gemeinsam in einem Projekt.



# Für Selbststudium: git-Kurs auf ILIAS

- Im Sommer wurde ein ILIAS-Kurs zum Thema **git** erstellt:

[https://elearning.hslu.ch/ilias/goto.php?target=crs\\_5207233&client\\_id=hslu](https://elearning.hslu.ch/ilias/goto.php?target=crs_5207233&client_id=hslu)

The screenshot shows the ILIAS course management system interface. At the top, there's a navigation bar with links like 'Zu Favoriten', 'Inhalt' (which is highlighted in blue), 'Info', 'Einstellungen', 'Mitglieder', 'Badges', 'Lernfortschritt', 'Metadaten', 'Export', 'Rechte', and 'Voransicht als Mitglied aktivieren'. Below the navigation is a toolbar with 'Zeigen', 'Verwalten', and 'Sortieren'. A button for 'Neues Objekt hinzufügen' and a link to 'Seite gestalten' are also visible. The main content area features the HSLU Hochschule Luzern logo on the left and a profile picture of a man on the right. A dark green horizontal bar spans across the top of the content area. The main heading 'Herzlich willkommen im HSLU-I Git Kurs' is displayed in white text. Below it, a welcome message from Silvan Wegmann is shown, followed by a brief description of the course goals and a note about help availability. A section titled 'Themen' lists five topics: 'Einführung', '1. Git Basics', '2. Branching & Merging', '3. Arbeiten mit Remotes', and '4. GitLab Issue Tracking'. At the bottom, a 'Hilfe' section includes a 'Forum' link.

- Geben Sie uns gerne Feedback zum Kurs! Danke.

# Zusammenfassung

- Versionskontrollsysteme bewusst einsetzen und nutzen.
  - Erfahrungen sammeln in der Anwendung.
- Minimale Basis:
  - Lokal: **checkout** und **commit**
  - Verteilt/Zentral: **clone**, **pull** (**fetch/merge**) und **push**
- Fortgeschrittene Nutzung:
  - Taggen – markieren von Revisionsständen (Baseline).
  - Branchen – getrennte Entwicklungszweige.
- **Koordination** zwischen den Teammitgliedern **sinnvoll!**
  - Konflikte können passieren, sind aber **nicht** das Ziel!
- Commits **immer** mit sinnvollem, aussagekräftigem Kommentar, welcher auf eine Issue-Nummer verweist (wenn vorhanden)!

# Quellen und Links

- git - <http://git-scm.com/> - für alle Plattformen.
- Git Clients
  - SmartGit - <http://www.syntevo.com/smartgit/> (Empfehlung!)
  - SourceTree - <https://www.sourcetreeapp.com/> (Atlassian)
  - TortoiseGit - <https://tortoisegit.org>
- IDE's
  - Alle relevanten IDE's verfügen über eine git-Integration.
  - Aber: Sehr unterschiedliche Umsetzungen...
- Git Hosting
  - HSLU GitLab – <https://gitlab.enterpriselab.ch/>
  - GitHub - <https://github.com/> (nur öffentliche Repos, Gratis)
  - BitBucket - <https://bitbucket.org/> (auch private Repos, Gratis)

**Fragen?**

Verteilte Systeme und Komponenten

# **Buildautomatisation**

## **Reproduzierbare Buildprozesse**

Roland Gisler

# Inhalt

- Wie wird Software entwickelt?
- Automatisierung des Buildprozesses
- Buildwerkzeuge
- Apache Maven
- Demo

# Lernziele

- Sie kennen die Vorteile eines automatisierten Buildprozesses.
- Sie können verschiedene Beispiele von Buildwerkzeugen benennen.
- Sie beherrschen die Anwendung eines ausgewählten Buildwerkzeuges (Apache Maven).
- Sie sind mit den wesentlichen Konzepten von Apache Maven vertraut.

# **Einleitung**

# Wie haben Sie bisher Software entwickelt?

- In den Modulen OOP/PLAB und AD haben Sie
  - typisch «alleine»,
  - in einem einzigen (kleinen) Projekt,
  - und meist vollständig in der Entwicklungsumgebung (IDE) gearbeitet.
- Wesentliche Schritte waren meist nur die Kompilation (automatisch durch die IDE), das Ausführen von Unit-Tests sowie, das Starten des «Programmes» selber.
  - Das lässt sich alles sehr einfach und direkt in der jeweiligen Entwicklungsumgebung erledigen.
- Die Arbeitsweise ist in einem realen Projekt aber etwas anders...

# Wie wird Software typisch entwickelt?

- Typische Situation
  - Arbeit im Team → gemeinsame Codebasis → VCS.
  - Code aufgeteilt in mehrere → Projekte / Module / Komponenten.
  - Verteilung des Endproduktes (Releases) in Form von binären Artefakten (typisch JAR-Dateien).
- Das heisst: Software entwickeln beinhaltet deutlich mehr Aufgaben:
  - Binaries erstellen für Deployment.
  - Testfälle ausführen, Test-Reports erstellen.
  - Qualitätssicherung (Codeanalyse, Metriken etc.).
  - Distributionen zusammenstellen und Deployment etc.
- Die Gesamtheit all dieser Tätigkeiten um aus Quellartefakten ein fertiges Produkt zu erstellen bezeichnet man als →Buildprozess.

# Der Buildprozess

- Generieren, Kompilieren, Testen, Packen, JavaDoc erzeugen...  
Viele dieser Tätigkeiten kann man problemlos mit einer modernen IDE erledigen!
- Aber: Es sind zum Teil aufwändige und manuelle Schritte!
  - Sie sind somit mühsam und fehleranfällig.
  - Bei grossen Projekten langsam und langweilig (Wartezeiten).
  - Bei langweiligen Task wird der Mensch unzuverlässig.
- Was, wenn Sie das **n**-mal pro Tag machen müssen?
- Bob the Builder?



# **Automatisation des Buildprozesses**

# Automatisation von Builds mit Script/Batches

- Idee: Man könnte ein Script schreiben, dass die notwendigen Tools (Kompiler, Packer etc.) sequenziell mit allen notwendigen Parametern ausführt.
-  Vorteile:
  - Vollautomatisierter Ablauf, keinerlei Interaktion mehr.
  - Reproduzierbare Ergebnisse.
  - Lange Builds können über bzw. in der Nacht laufen.
  - Unabhängig von Entwicklungsumgebung (IDE).
-  Nachteile:
  - Eher sturer, unflexibler Ablauf (oder komplizierte Scripte).
  - Abhängigkeit von Shell und Plattform (OS).
  - Aufwändige Wartung und Erweiterung.

# Alternative: Ein spezialisiertes Build-Werkzeug!

- Eigenständiges, spezialisiertes Werkzeug mit eigener (Script- oder Definitions-)Sprache.
  - Optimiert für die typischen Build-**Aufgaben**.
    - Aufruf von Kompiler, Packer, Tests, Deploying.
    - vereinfachter Umgang mit Ressourcen (Dateimengen).
    - Abhängigkeiten zwischen Dateien überprüfen und steuern.
  - Optimiert für die typischen Build-**Abläufe**.
    - Logische Abfolge und Abhängigkeiten zwischen Aufgaben.
  - Plattformübergreifend lauffähig.
    - Abstraktion der Plattformspezifika.
- ➔ Das gibt es natürlich schon lange! Und Sie kennen es schon! ☺

# **Build-Werkzeuge**

# Build-Werkzeug: make

- Die Mutter aller Build-Tools: `make`
  - Hauptsächlich für C/C++ - Projekte verwendet.
  - Existiert seit vielen Jahren, und wird noch immer genutzt.
  - Bietet eine sehr hohe Flexibilität.
- Sehr einfaches Beispiel:

```
program.o: program.c
    gcc -c program.c -o program.o
```

```
program: program.o
    gcc program.o -o
```

- Vielleicht eine etwas gewöhnungsbedürftige Syntax?

# Spezialisierte Build-Werkzeuge für Java

- Im Java-Ökosystem existiert mittlerweile eine ganze Palette von Werkzeugen!
  - Viele davon basieren selber auf Java (mindestens der JRE).
- Ein paar Beispiele:
  - **Ant** – «altes» und bewährtes Werkzeug, Java mit XML.
  - **Maven** – populäres, etabliertes Werkzeug, Java mit XML.
  - **Buildr** – junges Werkzeug, Ruby-Script.
  - **Gradle** – populäres, junges Werkzeug, Groovy-Script mit DSL.
  - **Bazel** – Buildwerkzeug von Google, Java mit Python-like Scripts.
- Die Qual der Wahl:
  - Implizite Regeln vs. explizite Aufgaben (Imperativ/Deklarativ).
  - Es gibt sehr viele unfaire Vergleiche (wie bei den VCS)!
  - Vergleich zu Datenbanken: Pragmatisch, stabil, langfristig...

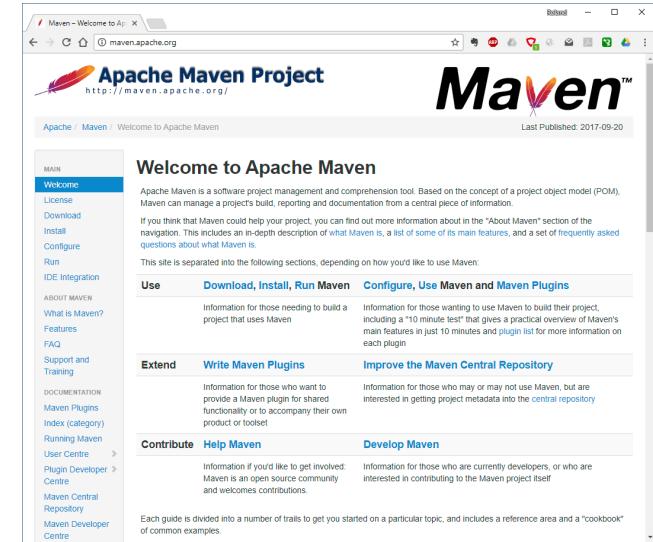
# **Verschiedene Produkte – gemeinsame Ziele und Vorteile!**

- Einheitliche, einfache Definition eines Build-Ablaufes.
  - Einfache und doch flexible Syntax/Sprache.
- Anwendung über relativ eingängige Build-Ziele/Goals/Targets.
  - Einfache Anwendung für Entwickler\*in.
- Optimierte Abläufe (nur machen, was nötig ist)
  - z.B. optimierte Kompilation: Nur modifizierte Quelldateien kompilieren (wenn Quellen neuer als Kompilate).
- Erweiterbarkeit
  - Flexibel erweiterbar mit neuen Fähigkeiten / Funktionen.
- Möglichst geringer Ressourcenverbrauch (shell, headless etc.).
- Reproduzierbarer Ablauf, technologische Konstanz.

# **Apache Maven**

# Buildautomatisation mit Apache Maven

- Apache Maven - <http://maven.apache.org/>
- Ist selber in Java implementiert und somit plattformunabhängig.
- Deklaration des Projektes in XML.
  - Deklarativer Ansatz (nicht imperativ)!
  - Das zentrale Element pro Projekt stellt das **P**roject **O**bject **M**odel (POM) dar: **pom.xml**
  - Definiert alle Metainformationen, Plugins und Dependencies.
- Bewährtes und robustes Buildwerkzeug für Java!
  - Integration in praktisch jede(r) IDE vorhanden.
  - Minimale Ressourcen notwendig (nur JDK).
  - Basiert auf einem globalen, **binären** Repository!

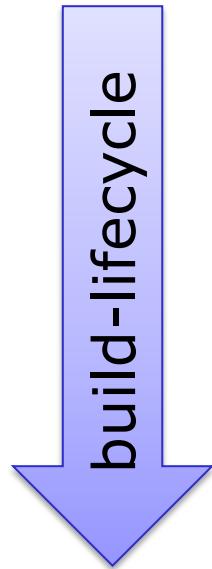


# Apache Maven - Konzept

- Maven selber kann eigentlich sehr wenig! Es definiert im wesentlichen nur die POM-Struktur und die →Lifecycle-Phasen.
- Alles wird dann von dynamisch geladenen Plugins (mit eigenen Releases) erledigt! Extreme grosse Vielfalt an Funktionen!
  - Entfernt vergleichbar mit der Eclipse Platform und (JDT-)Plugins.
- Somit ist Maven extrem **flexibel** und nahezu beliebig **erweiterbar**.
- Vorsicht: Neben den Core-Plugins (die von Apache Maven selber entwickelt werden), existieren sehr viele weitere (Dritt-)Plugins, deren Qualität sehr unterschiedlich ist.
  - Vorsicht: Maven-Plugins sind auch Abhängigkeiten!
- Man kann natürlich auch eigene Plugins schreiben.
  - Aber bitte nur wohlüberlegt: Lohnt es sich wirklich?

# Apache Maven – «lifecycle phases» und «goals»

- Maven definiert einen generalisierten Ablauf eines Buildprozesses – den so genannten «lifecycle» (Auszug, Vereinfacht):
  - **validate** – Validiert die Projektdefinition.
  - **compile** – Kompliation der Quellen.
  - **test** – Ausführen der Unit-Tests.
  - **package** – Packen der Distribution (JAR, EAR etc.)
  - **verify** – Ausführen der Integrations-Tests.
  - **install** – Deployment im lokalen Repository.
  - **deploy** – Deployment im zentralen Repository.
- Die per Deklaration aktivierte Plugins registrieren sich und ihre Aufgaben (häufig) automatisch in den jeweiligen Phasen!
  - Buildprozess passt sich quasi dynamisch der Projektart an.



# Apache Maven – Binäre Repositories

- Maven integriert auch das → Dependency Management.
- Im POM deklarierte Abhängigkeiten sowohl von Plugins als auch von Libraries werden aus einem zentralen Repository geladen.
  - The Central Repository: <https://search.maven.org/>
- Alle heruntergeladenen Artefakte werden in einem lokalen Repository auf dem Rechner gespeichert (gecached).
  - Lokales Repository: **\$HOME/.m2/repository**
  - Nebenbei: Im **\$HOME/.m2** befindet sich auch die Konfigurationsdatei **settings.xml** - haben sie sie kopiert?
- Firmen und Organisationen nutzen typisch ein eigenes Repository als lokaler Speicher und Mirror von öffentlichen Repositories.
  - HSLU RepoHub Nexus: <https://repohub.enterpriselab.ch/>

# Apache Maven – Single- und Multimodulprojekte

- In den Modulen OOP/PLAB und AD haben wir bisher nur einfache «Single-Modul»-Projekte genutzt: Ein Verzeichnis enthielt genau ein einziges Projekt.
  - Einfache Struktur für kleine in sich abgeschlossene Projekte.
  - Kriterium: **Releaseeinheit**, d.h. was wollen wir einzeln unter einer Version «releasen» (veröffentlichen) können.
- Maven unterstützt aber auch Multimodulprojekte, d.h. ein Projekt kann beliebig viele Submodule enthalten → Modularisierung!
  - Übergeordnete Konfiguration wird an Submodule «vererbt».
  - Abhängigkeiten zwischen Submodulen können definiert werden.
- Diese Technik setzen wir im Modul VSK ein!

# Demo / Screencast's

- Maven – Einsatz in der Shell:  
**EP\_12\_SC01\_MavenConsole.mp4**
- Maven – Integration in IDE (NetBeans, Eclipse, IntelliJ):  
**EP\_12\_SC02\_MavenIDE.mp4**



# Apache Maven Dokumentation

- Ausführliche Dokumentation auf der Projektseite:  
<http://maven.apache.org/>
- Wichtige Konzepte (zur Vertiefung für Interessierte):
  - Lifecycles, Phases und Goals
  - Dependencies / Dependency-Resolution
  - Repositories
  - Deployment
  - Plugins
  - Site

# Installation von Maven – Einsatzszenarien

- **Variante 1**, minimalistisch: Maven ist in vielen IDEs bereits enthalten!
  - z.B. Eclipse, Netbeans, IntelliJ - nicht immer aktuellste Version!
  - Vorteil: sehr einfache Anwendung ohne Aufwand.
  - Nachteil: **unnötige** Abhängigkeit zur IDE.
- **Variante 2, empfohlen**: Installation von Maven auf das System
  - Distribution als ZIP, einfach entpacken!
  - `$MAVEN_HOME/bin` in Suchpfad aufnehmen, damit `mvn`-Kommando gefunden wird
  - Vorteil: **Unabhängig** von IDE, Build in einer Shell möglich
  - Nachteil: Installation (nicht wirklich ein Nachteil, oder?)
- **Wichtig**:  
`settings.xml` in jedem Fall lokal in `$HOME/.m2` kopieren.

# **Quellen und Links**

- Apache Maven – <http://maven.apache.org/>
- Gradle – <https://gradle.org/>
- Apache Ant – <http://ant.apache.org/>

**Fragen?**

Verteilte Systeme und Komponenten

# **Komponentenbegriff und Schnittstellendesign**

Martin Bättig



# Inhalt

- Komponenten
- Nutzen von Komponenten
- Entwurf mittels Komponenten
- Exkurs: Rolle von Komponenten in Softwarearchitekturen
- Konzept der Schnittstelle
- Dienstleistungsperspektive
- Spezifikation von Schnittstellen

# Lernziele

- Sie kennen das Konzept der Software-Komponenten.
- Sie können Komponenten gemäss Spezifikation erstellen, dokumentieren, testen und überarbeiten.
- Sie kennen die Kriterien für gute Schnittstellen im Software-Entwurf und können solche Schnittstellen entwerfen.
- Sie können verschiedene Arten von Schnittstellen angemessen dokumentieren.

# **Komponenten**

# Definition des Komponentenbegriffs

(es gibt mehrere Definitionen, nachfolgend zwei gebräuchliche)

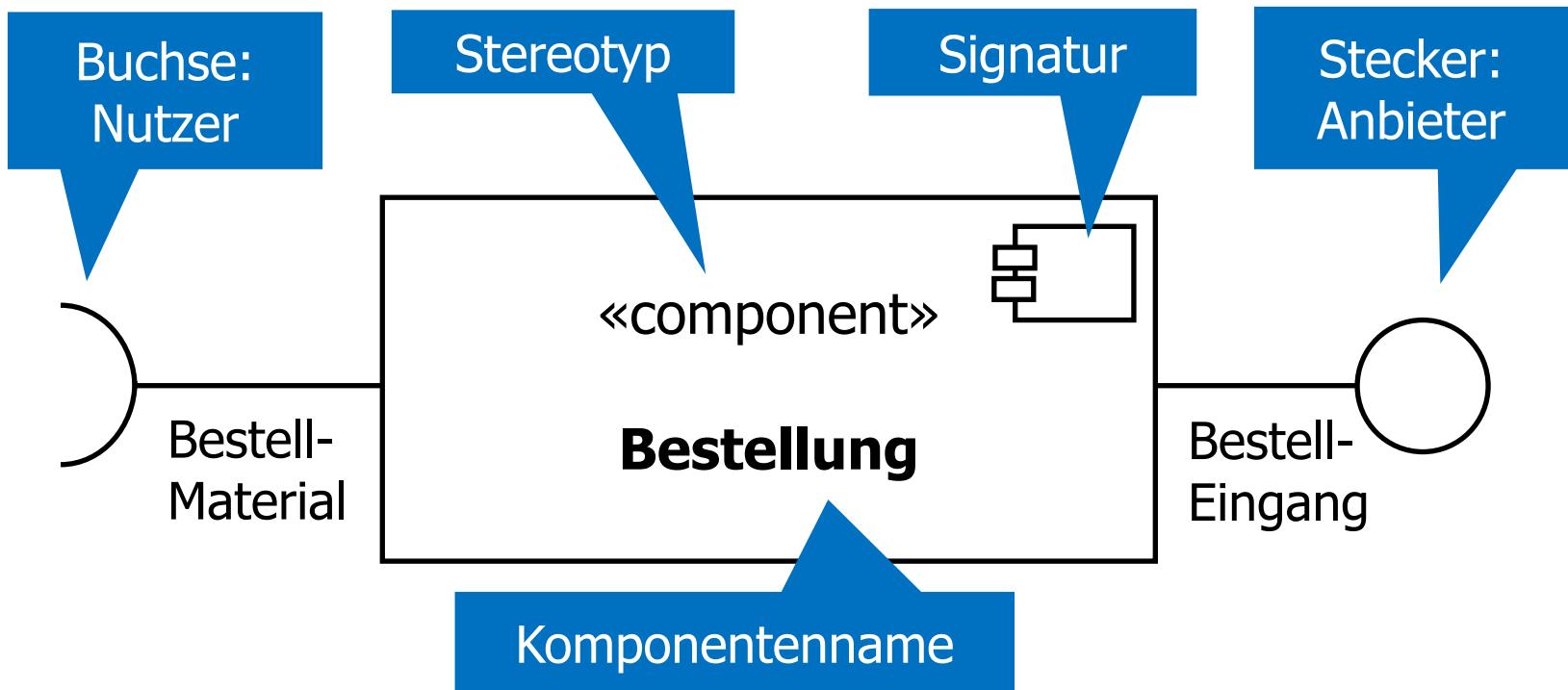
- "A software component is a unit of composition with contractually specified interfaces and explicit context dependencies only. A software component can be deployed independently and is subject to composition by third parties." (*European Conference on Object-Oriented Programming (ECOOP), 1996*)
- "Eine Software-Komponente ist ein Software-Element, das zu einem bestimmten Komponentenmodell passt und entsprechend einem Composition-Standard ohne Änderungen mit anderen Komponenten verknüpft und ausgeführt werden kann." (*Council, Heineman: Component-Based Software Engineering, Addison-Wesley, 2001*)

# Eigenschaften von Komponenten

- Eigenständige, ausführbare Softwareeinheiten (Laufzeit-Sicht) (\*).
- Über ihre Schnittstellen in Ihrer Umgebung austauschbar definiert.
- Lassen sich unabhängig voneinander entwickeln.
- Kunden- / anwendungsspezifische, bzw. wiederverwendbare Software sowie Components-off-the-shelf (COTS).
- Können installiert bzw. deployed werden.

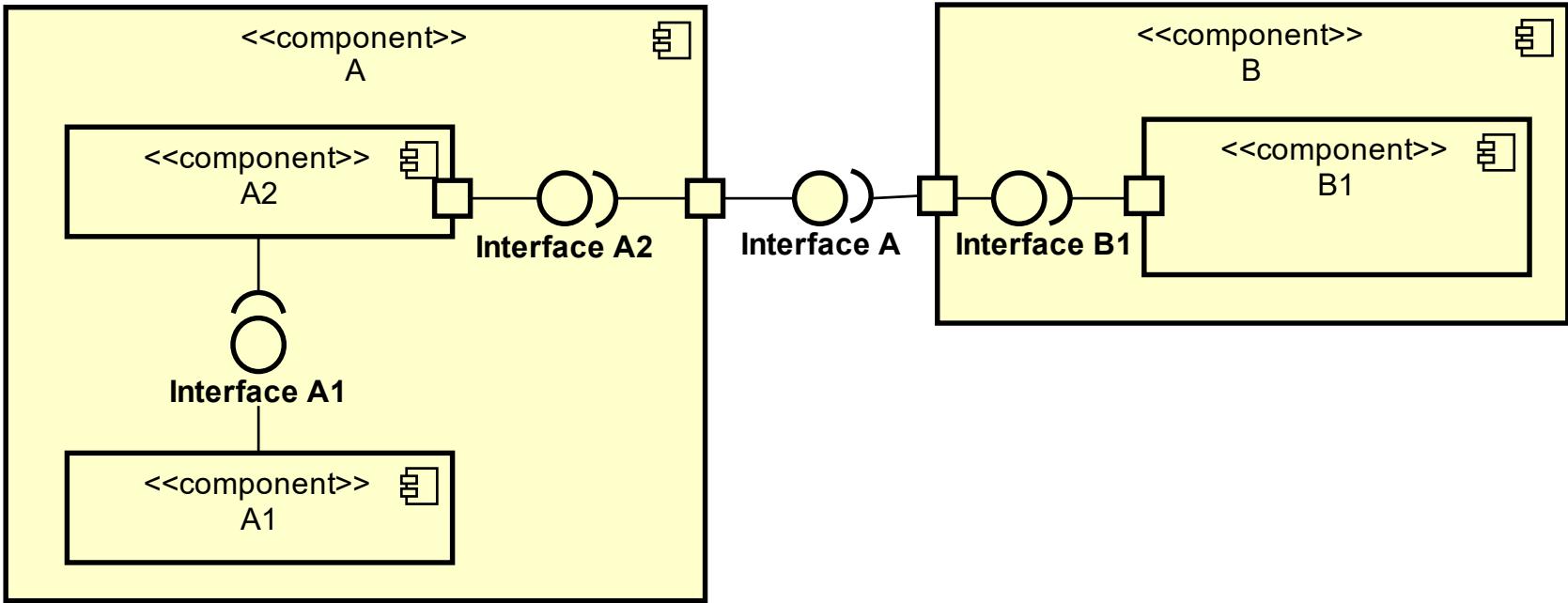
(\*) Nicht zwingend in einem eigenen Prozess / Thread.

# Modellierung von Software Komponenten in UML



# Komponentenbegriff ist hierarchisch

- Modellierung vom "Subsystemen" in UML mittels Verschachtelung:



- UML kennt auch den Stereotyp <<subsystem>>, welcher aus Sicht Modellierung identisch zu <<component>> ist.
- Applikation kann wiederum Komponente eines grösseren Systems sein (ggf. mit anderem Komponentenmodell).

# Komponentenmodelle

- Komponentenmodelle sind konkrete Ausprägungen des Paradigmas der Komponentenbasierten Entwicklung.
- Neben der genauen Form und den Eigenschaften einer Komponente muss das Komponentenmodell einen **Interaction-Standard** und einen **Composition-Standard** festlegen.
- Komponentenmodelle können ausserdem Implementierungen verschiedener Hersteller besitzen.

## Beispiele:

- Enterprise Java Beans (Java).
- (Distributed) Component Object Model (Microsoft, C++).
- CORBA Component Model (Sprachunabhängig).
- OSGi (Java).
- Viele proprietäre Komponentenmodelle (Eigenentwicklungen).

# Interaction-Standard

- Beschreibt den Schnittstellenstandard, also wie Komponenten innerhalb eines Komponentenmodells miteinander kommunizieren.

## Beispiele:

- verteilt oder lokal.
  - verteilte Objekte, Remote-Procedure-Calls, Unix-Pipes (Streams).
  - konkrete Technologien wie SOAP / REST (HTTP).
- 
- Legt fest wie innerhalb einer Komponente die Schnittstelle festgelegt wird.

## Beispiele:

- Interface Definition Language (CORBA).
- WSDL (SOAP).

# Composition-Standard

- Beschreibt wie der Entwickler Komponenten zusammensetzt um grössere Einheiten zu bilden.
- Beschreibt wie Komponenten ausgeliefert werden.

## Beispiele:

- Unix-Prozesse: Zusammenfügen via Pipes, Auslieferung als Binaries.
- Webservices: Zusammenfügen via Domainname / IP-Adresse, Auslieferung als WAR.
- Microservices: Zusammenfügen via Domainname / IP-Adresse, Auslieferung als Service (Package / ZIP / Docker / etc.).

# **Nutzen von Komponenten**

# Wiederverwendung

- Verpackung versteckt Komplexität (divide and conquer).
- Reduzierte Auslieferzeit (des eigenen Produkts):
  - Wiederverwenden ist schneller als selbst bauen.
  - Weniger Tests notwendig.
- Grössere Konsistenz: Verwendung von "Standard"-Komponenten.
- Möglichkeit die Beste von verschiedenen Komponenten zu verwenden: Wettbewerb und Markt.

## **Erbringt vereinbarte Dienstleistung**

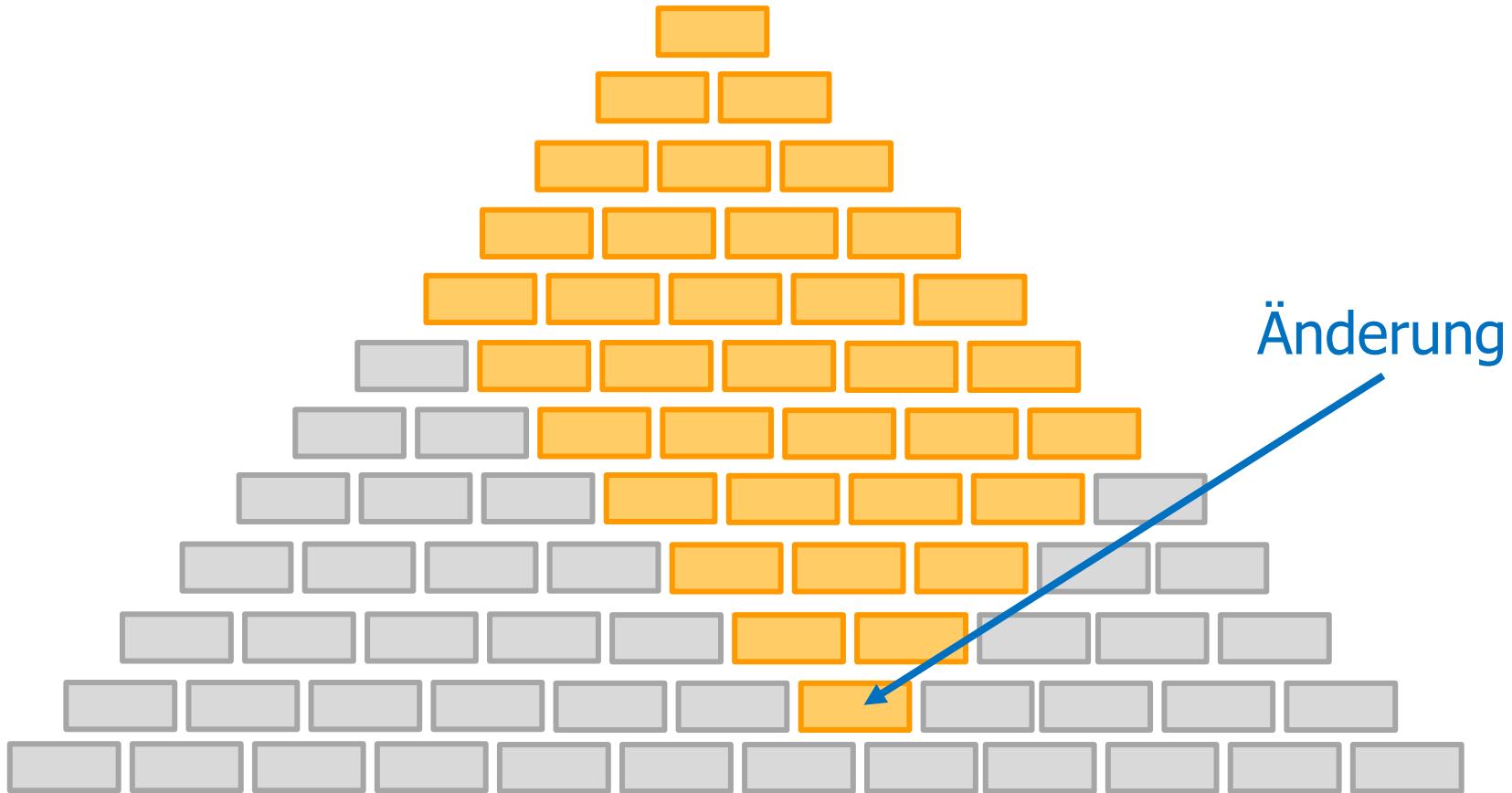
- Erhöhte Produktivität: Existierende Komponenten zusammenfügen.
- Höhere Qualität: Vorgetestet.

# Vollständigkeit

- Komponente als ganzes ersetzbar.
- Parallele und verteilte Entwicklung.
  - Präzise Spezifikationen.
  - Verwaltete Abhängigkeiten.
- Verbesserte Wartung.
  - Kapselung limitiert den **Auswirkung von Veränderung.**

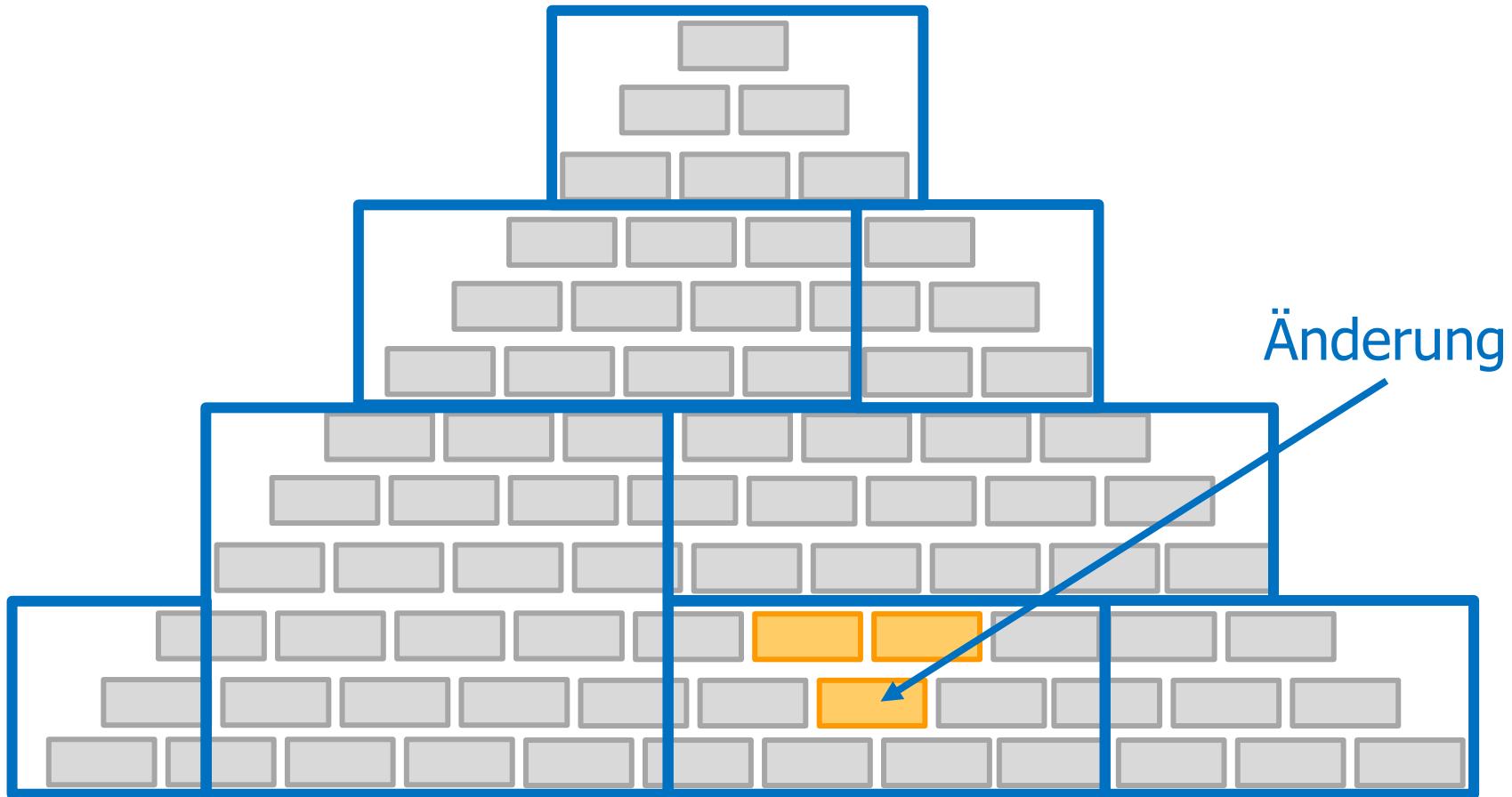
# Auswirkung von Änderungen

- Monolithisches System (keine Komponenten):



## Auswirkung von Änderungen (forts.)

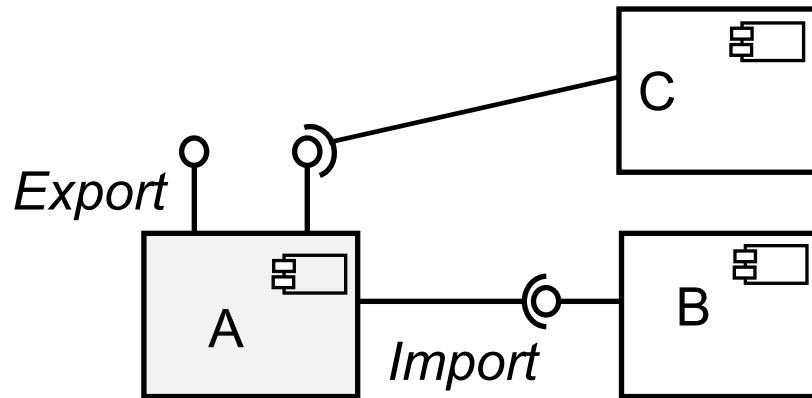
- Komponentenbasiertes System:



# **Entwurf mittels Komponenten**

# Spezifikation von Komponenten

- **Export:** unterstützte Interfaces, die andere Komponenten nutzen können.
- **Import:** benötigte / benutzte Interfaces von anderen Komponenten.
- **Verhalten:** Verhalten der Komponente.
- **Kontext:** Rahmenbedingungen im Betrieb der Komponente.



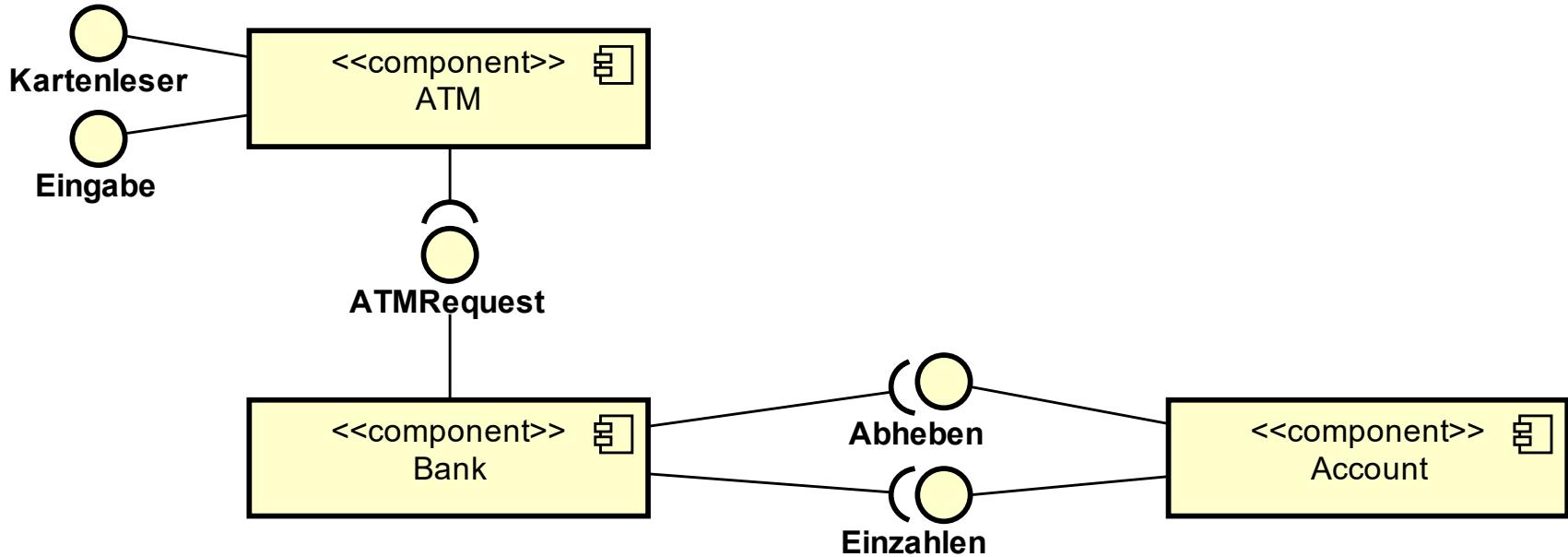
# Verhaltenssicht

Weil Komponenten ausführbare SW-Einheiten sind, lässt sich mit ihrer Hilfe das Systemverhalten auf höherer Flughöhe gut darstellen:

- **Components & Connectors:** ausführbare Einheiten und gemeinsame Daten
- **Datenfluss:** Datenfluss zwischen Komponenten.
- **Kontrollfluss:** Wird angestoßen von ...
- **Prozess:** Welche Komponenten laufen parallel?
- **Verteilung:** Zuordnung der Komponenten zur HW.

# UML-Komponentendiagramm: "Verdrahtung"

Beispiel: Bankomat (ATM)



- Sichtbar sind Komponenten und Konnektoren
- Wie sieht es mit Daten- und Kontrollfluss sowie Verteilung aus?

# **Exkurs: Rolle von Komponenten in Architekturen**

# Softwarearchitektur

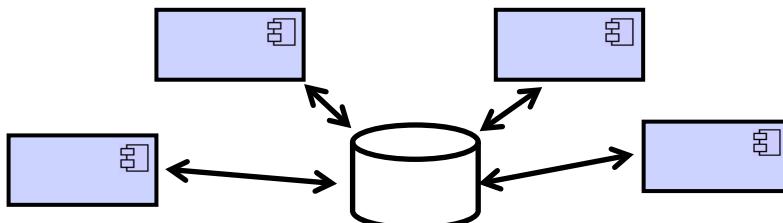
- Softwarearchitektur enthält Informationen über die Struktur eines Software-Systems:
  - Aus welchen Komponenten besteht ein System?
  - Wie kommunizieren die einzelnen Komponenten?
- Muss relativ früh zu Beginn der Softwareentwicklung stattfinden
  - Spätere Änderung u.U. sehr teuer.
- Architekturmuster für häufig wiederkehrende Architekturen.

# Typische Architekturmuster

- Hinweis: Pfeile zeigen den Datenfluss



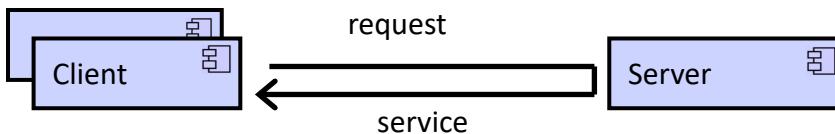
**Pipe and Filter**



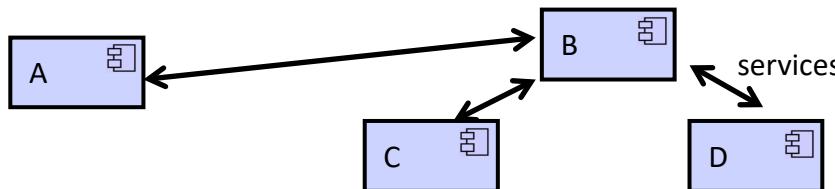
**Shared Data**



**Publish-Subscribe**



**Client-Server**



**Peer-to-Peer**

# **Begriff und Konzept der Schnittstelle**

# Schnittstellen: Begriff und Konzept

- Wo Komponenten kooperieren oder zusammengefügt werden sollen, müssen sie zueinander passen.
- Wir konstruieren Verbindungsstellen, Festlegungen, welche die Kombinierbarkeit sicherstellen.
- Eine Schnittstelle tut nichts und kann nichts.
- Schnittstellen verbinden:
  - Komponenten untereinander: Programmschnittstellen oder kurz Schnittstelle.
  - Komponenten mit dem Benutzer: Benutzerschnittstellen.

# Bedeutung des Schnittstellenkonzepts

- **Verständlichkeit:** Schnittstellen machen Software leichter verständlich, denn es genügt, die Schnittstelle (\*) zu betrachten.
- **Reduktion von Abhängigkeiten:** Schnittstellen gestatten es, die Abhängigkeiten in der Schnittstelle zu konzentrieren und jede Abhängigkeit von der Implementierung zu vermeiden.
- **Wiederverwendung:** Schnittstellen erleichtern die Wiederverwendung von bewährten Implementierungen und sparen damit Arbeit.

(\*) Damit sind nicht nur die Signaturen der Methoden gemeint, sondern auch deren Verhalten und Zusammenspiel.

## Bedeutung des Schnittstellenkonzepts (forts.)

- Die Java-Implementierungen der gängigen Behälter (HashSet, TreeSet, HashMap, TreeMap, usw.) sind mit Sicherheit leistungsfähiger als alles, was der beste Programmierer in seinem Projekt unter Zeitdruck fertig bringt.
- Achtung: Schnittstellen entfalten ihren Nutzen nur dann, wenn wirklich ohne Bezug auf die Implementierung nur gegen Schnittstellen programmiert wird.

# Breite einer Schnittstelle

Bestimmt über:

- Anzahl Operationen (mehr ist breiter).
- Anzahl Funktionsüberschneidungen (mehr ist breiter).
- Anzahl von Parametern (mehr ist breiter).
- Anzahl globaler Daten (mehr ist breiter).
- Typ der Parameter und Rückgabewerte (generisch ist breiter).

**Schmälere Schnittstellen haben weniger Abhängigkeiten!**

# Kriterien für gute Schnittstellen

- Schnittstellen sollen **schmal** sein.
- Schnittstellen sollen **einfach zu verstehen** sein.
- Schnittstellen **sollen gut dokumentiert** sein.

# Beispiel: Modem-Schnittstelle

Reduktion von Abhängigkeiten erzielt durch Aufteilung:

## Variante 1 (schmal):

```
interface Modem {  
    void dial(String number);  
    void hangup();  
    void send(char data);  
    char receive();  
}
```

## Variante 2 (noch schmäler):

```
interface Connection {  
    void dial(String number);  
    void hangup();  
}  
  
interface Transmit {  
    void send(char character);  
    char receive();  
}
```

# Weitere Beispiele

- Breit:

<https://docs.oracle.com/javase/8/docs/api/java/sql/ResultSet.html>

- Schmal:

<https://docs.oracle.com/javase/8/docs/api/java/lang/Comparable.html>

# **Schnittstellen: Design by Contract**

# Dienstleistungsperspektive

**Konsument**

stimmt zu



arbeitet gemäss

**Anbieter**



**Vertrag**

**Dienstleistung**

A large yellow arrow pointing downwards, indicating the final step in the process is the delivery of the service.

# Serviceperspektive

**Konsument**

stimmt zu



arbeitet gemäss

**Anbieter**



**Schnittstelle**



**Dienstleistung**

# Design by Contract

Das Zusammenspiel der Komponenten wird durch einen „Vertrag“ erreicht, dieser besteht aus:

- **Preconditions:** Zusicherungen, die der Aufrufer einzuhalten hat.
- **Postconditions:** Nachbedingungen, die der Aufgerufene garantiert.
- **Invarianten:** Bedingung, die Instanzen einer Klasse ab der Erzeugung erfüllen müssen (kann während der Ausführung einer Funktion/Methode verletzt sein).

Der Vertrag kann sich auf Variablen, Parameter und Objektzustände beziehen.

# Verantwortlichkeiten bei "Design by Contract"

---

	Nutzer	Anbieter
<b>Precondition</b>	Nutzer muss sicher stellen, dass Vorbedingungen vor Ausführung einer Methode gelten	Prüfen. Aussagekräftige Fehlermeldung, falls inkorrekt.
<b>Postcondition</b>	<a href="#">Während Entwicklung: Doppelcheck mit assert (Defensives Programmieren)</a>	Anbieter muss sicher stellen, dass Nachbedingungen nach Ausführung einer Methode gelten.
<b>Invariante</b>		Anbieter muss sicher stellen, dass Invarianten vor- und nach Ausführung jeder Methode gelten. <a href="#">Während Entwicklung: Doppelcheck mit assert (Defensives Programmieren)</a>

# Beispiel mit Pre- und Postconditions

```
public class LinkedList {  
    private static class Node {  
        int value;  
        Node prev, next;  
        LinkedList list;  
    }  
  
    private Node first, last; // INV: either both null or both not null.  
    private int size; // INV: size >= 0  
}  
  
// Inserts a new element with content value after node.  
// PRE: Node is element of list (and not null).  
// POST: Returns node containing new element,  
//        new element is part of list, size of list increased by 1.  
public Node insert(Node node, int value) {  
    if (node.list != this) throw new IllegalArgumentException(...);  
    ...  
}
```

Klasseninvarianten

Pre- und  
Postconditions

# Klassenraumübung: Pre- and Postconditions

Gegeben sei folgende Methode:

```
static int sum(int[] arr, int fromIndex, int toIndex) {  
    int sum = 0;  
    for (int i = fromIndex; i < toIndex; ++i) {  
        sum += arr[i];  
    }  
    return sum;  
}
```

## Aufgabe:

- Bestimmen Sie Pre- and Postconditions.
- Würde Sie die Eingabeparameter prüfen? Falls ja, wie?

# **Spezifikation von Schnittstellen**

# Dokumentation von Schnittstellen

- Zur Programmschnittstelle gehört alles,
  - was für die Benutzung der Komponente wichtig ist und
  - was der Programmierer verstehen und beachten muss.
- Jede Programmschnittstelle definiert eine Menge von Methoden mit den folgenden Eigenschaften:
  - **Syntax** (Rückgabewerte, Argumente, in/out, Typen).
  - **Semantik** (Was bewirkt die Methode?).
  - **Protokoll** (z.B. synchron, asynchron).
  - **Nichtfunktionale Eigenschaften** (Performance, Robustheit, Verfügbarkeit, bei Web-Anwendungen möglicherweise auch Kosten).

# Dokumentation von Schnittstellen (forts.)

- Die Syntax einer Programmschnittstelle notieren wir in der verwendeten Programmiersprache.
- Die Semantik lässt sich weniger leicht darstellen. Obwohl Schnittstellen so wichtig sind, gibt es bis heute keine allgemein akzeptierte Art der Dokumentation für deren Semantik.
- Beispiel für die Dokumentation einer Schnittstelle ist die Spezifikation des StringPersistor:

[ILIAS: I.BA\\_VSK\\_MM.H2201 » Projekt » VSK\\_stringpersistor-api-6.0.2](#)

# Angaben aus der Sicht des Schnittstellenanbieters

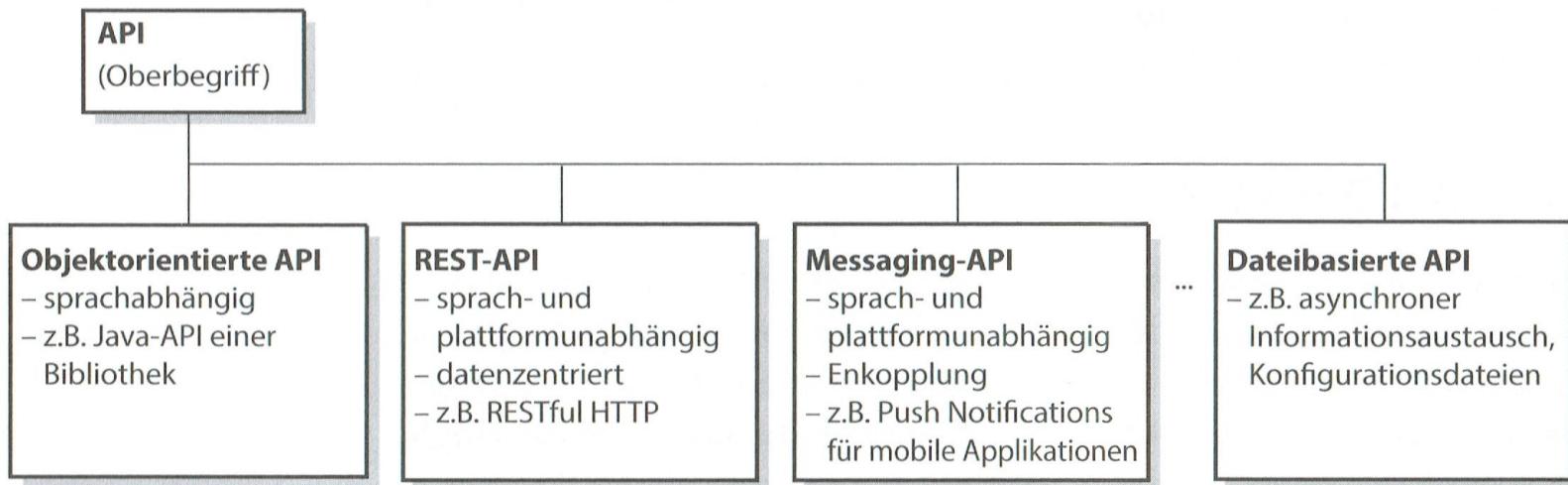
- Name der Schnittstelle.
- Eingabeparameter: Welche Informationen werden an die Komponente weitergeleitet.
- Ausgabeparameter: Welche Informationen die Komponenten zurückliefert.
- Zustandsänderung der Komponenten.
- Spezifikation inwiefern sich Eingabeparameter auf die Zustandsänderung und die Ausgabeparameter auswirken.

# Schnittstellen in Java

- In Java werden Schnittstellen mit Java-Interfaces deklariert.
- Schnittstellen werden (wie Klassen) zu class-Dateien kompiliert.
- Schnittstellen können somit (wie Klassen) in JAR-Dateien verpackt und verteilt werden.
- Sinnvollerweise dokumentiert man Schnittstellen besonders gut mit JavaDoc (woraus eine Dokumentation im HTML-Format erzeugt werden kann).
- **Schnittstellen an der Systemgrenze und zwischen Subsystemen sind architekturrelevant und werden in der Architekturbeschreibung dokumentiert.**
- öffentliche Schnittstellen bezeichnet man häufig als API: Application Programmer Interface (können auch mehrere Java-Interfaces sein).

# Application Programmer Interface (API)

- Eine API spezifiziert die Operationen sowie die Ein- und Ausgaben einer Softwarekomponente.
- Hauptzweck besteht darin, eine Menge an Funktionen unabhängig von ihrer Implementierung zu definieren.
- Die Implementierung kann variieren ohne die Benutzer der Softwarekomponente zu beeinträchtigen.



# Zusammenfassung

- Komponente: Softwareelement passend zu einem bestimmten Komponentenmodell, eigenständig ausführbar und über Schnittstellen austauschbar definiert.
- Nutzen erzielt durch einfache Wiederverwendung, Erbringung der vereinbarten Dienstleistung sowie vollständiger Ersetzbarkeit.
- Komponenten limitieren Auswirkung von Änderungen auf das Gesamtsystem.
- Entwurf mittels Komponenten durch Betrachtung von importierten und exportierten Schnittstellen, sowie durch die Beschreibung des Verhaltens und des Ausführungskontextes.
- Darstellung in UML mittels Komponentendiagramm.

# Zusammenfassung (forts.)

- Begriff und Konzept der Schnittstelle:
  - Schnittstellen machen Software leichter verständlich.
  - Schnittstellen helfen Abhängigkeiten zu reduzieren.
  - Schnittstellen erleichtern die Wiederverwendung.
  - Kriterien: schmal / einfach zu verstehen / gut dokumentiert.
- Dienstleistungsperspektive / Design by Contract.
  - Service Provider / Service Consumer.
  - Preconditions / Postconditions / Invarianten.
- Spezifikation von Schnittstellen
  - Dokumentation: Syntax / Semantik / Protokoll / nicht-funktionale Eigenschaften.
  - UML-Notationsmöglichkeiten.
  - API dokumentiert öffentliche Schnittstellen.

# Literatur und Quellen

- Die UML-Kurzreferenz für die Praxis von B. Oestereich, Oldenbourg Wissenschaftsverlag, 2014.
- Moderne Software-Architektur von Johannes Siedersleben, Dpunkt Verlag, 2004.
- Component Software: Beyond Object-Oriented Programming von Clemens Szyperski, Addison-Wesley Professional, 2002.
- Effektive Software-Architekturen von Gernot Starke, Hanser Fachbuch, 2002.
- Software Engineering von Jochen Lodewig & Horst Richter, Dpunkt Verlag, 2013.

# Fragen?

Verteilte Systeme und Komponenten

# **Messageorientierte Kommunikation**

Martin Bättig



# Inhalt

- Messageorientierte Kommunikation
- Transiente Kommunikation am Beispiel von ZeroMQ und WebSockets.
- Persistente Kommunikation am Beispiel von ActiveMQ Artemis.
- Zusammenfassung

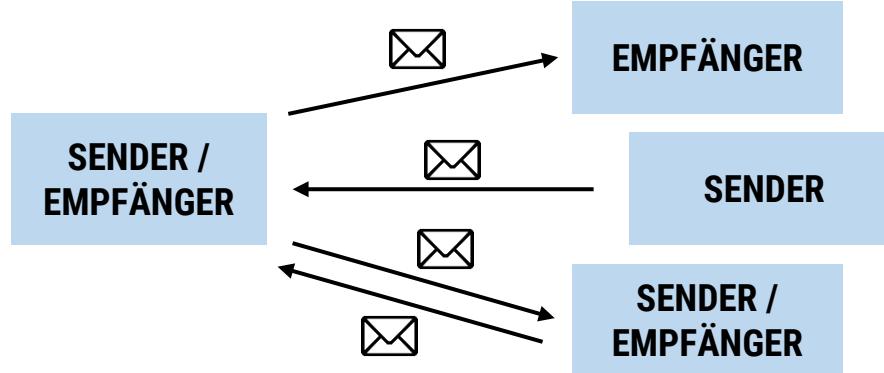
# Lernziele

- Sie wissen was Persistenz und Synchronität in der Kommunikation bedeutet und können daraus die möglichen Kommunikationsformen ableiten.
- Sie kennen die verschiedenen Kommunikationsmuster und können einschätzen, wann welches anzuwenden ist.
- Sie wissen welche Rolle Message-Queues und Message-Broker im Rahmen der persistenten Kommunikation einnehmen.

# **Messageorientierte Kommunikation**

# Kommunikation mittels Messages (Nachrichten)

**Prinzip:** Kommunikation mittels **expliziten Messages** gesendet von einem **Sender** zu einem oder mehreren **Empfängern**.



## Verwendung:

- Nebenläufige und parallele Programmierung (z.B. Actor-Model [1]).
- Interprozesskommunikation (z.B. Unix-Sockets).
- Kommunikation zwischen verteilten Systemen.

## Abgrenzung zu:

- Synchroner Remote-Procedure-Call (Transparenz)
- Streaming (z.B. reines TCP, Unix-Pipes).

[1] Carl Hewitt, Peter Bishop, and Richard Steiger. 1973. A universal modular ACTOR formalism for artificial intelligence. In Proceedings of the 3rd international joint conference on Artificial intelligence (IJCAI'73).

# Fragestellungen der messageorientierten Kommunikation

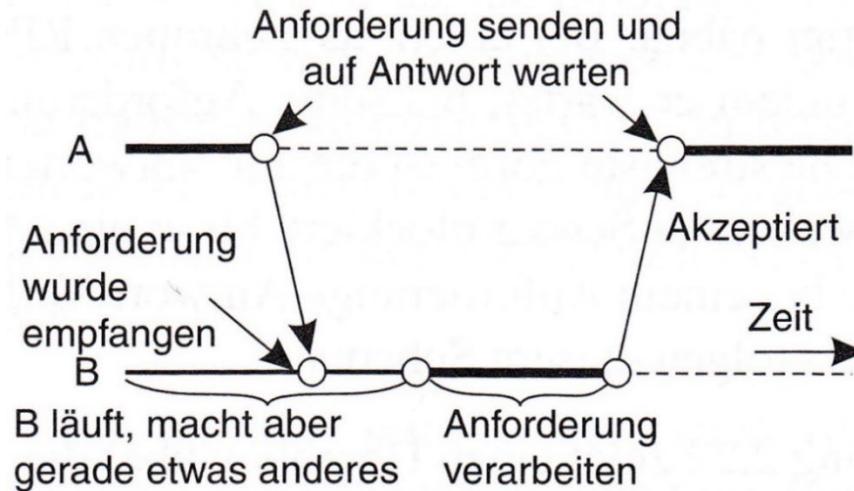
- **Zuverlässigkeit:** Können Messages verloren gehen?
- **Reihenfolge:** Kommen die Messages in der vorgesehenen Reihenfolge an?
- **Anzahl Empfänger:**
  - Unicast / Anycast (1:1)
  - Multicast / Broadcast (1:n)
  - Client-Server (n:1)
  - Peer-to-Peer (m:n)
- **Aufführungszeitpunkt:** Synchron vs. asynchron.
- **Sicherheit:** Offen vs. Zugangsgeschützt, Verschlüsselung.  
=> Ggf. Bereits auf Stufe Kommunikationskanal klären.

# Persistente vs. transiente Kommunikation

- **Persistente** Kommunikation: Nachricht wird solange gespeichert bis Empfänger bereit ist (z.B. E-Mail).
- **Transiente** Kommunikation: Nachricht wird nur gespeichert, solange sendende und empfangende Applikation ausgeführt werden (z.B. Router, Socket).

# **Transiente messageorientierte Kommunikation**

# Synchrone Kommunikation

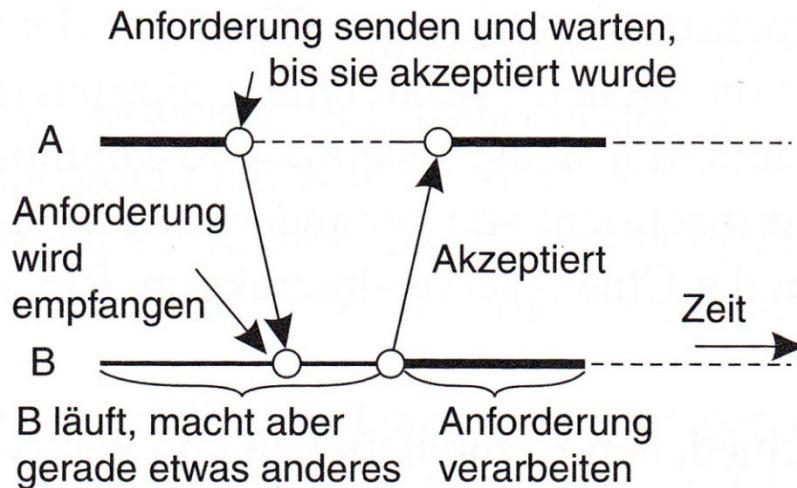


## Beispiele:

- Remote-Procedure-Call.
- Anfrage einer Ressource mittels HTTP-Protokoll und synchroner Auslieferung.

Zeit bis zur Verarbeitung der Anforderung kurz halten, z.B. mit Threads oder Non-Blocking I/O.

# Asynchrone Kommunikation



## Beispiele:

- Asynchroner Remote-Procedure-Call.
  - **Aber wie kommt das Resultat zurück?**
- Anfrage einer Ressource mittels HTTP-Protokoll mit Quittierung (ACK)
  - **Auch hier: Wie kommt der Client an das Resultat der Anfrage?**

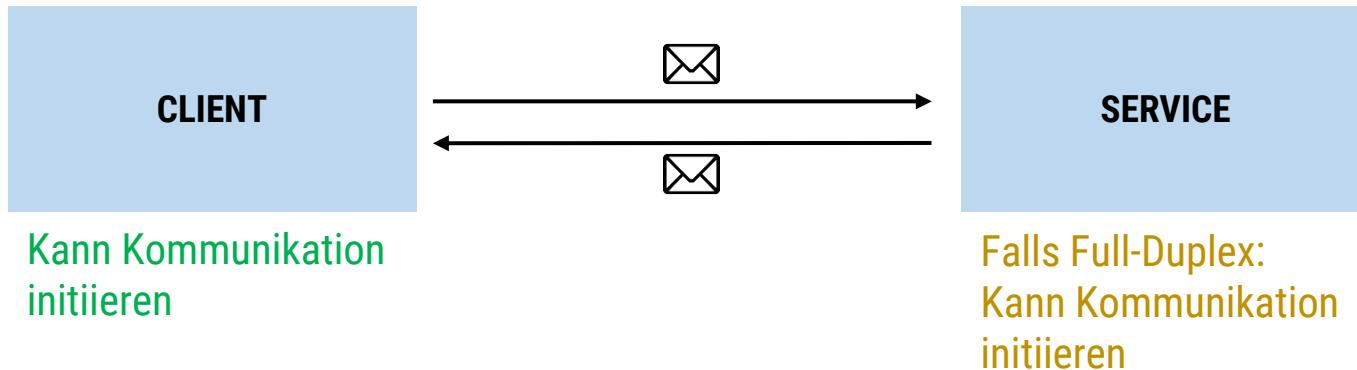
# Kommunikationsmuster (Auswahl)

- Kommunikation lässt sich in Muster einteilen.
- Bekannte Muster sind:
  - **Request-Reply:** Client-Server (One-To-One).
  - **Publish-Subscribe:** Datenverteilung (One-To-Many).
  - **Pipeline:** Verarbeitung in mehreren Schritten (One-To-Many).

# Muster: Request-Reply

**Ziel:** Verbinden **einer Menge von Clients** mit **einer Menge von Services**.

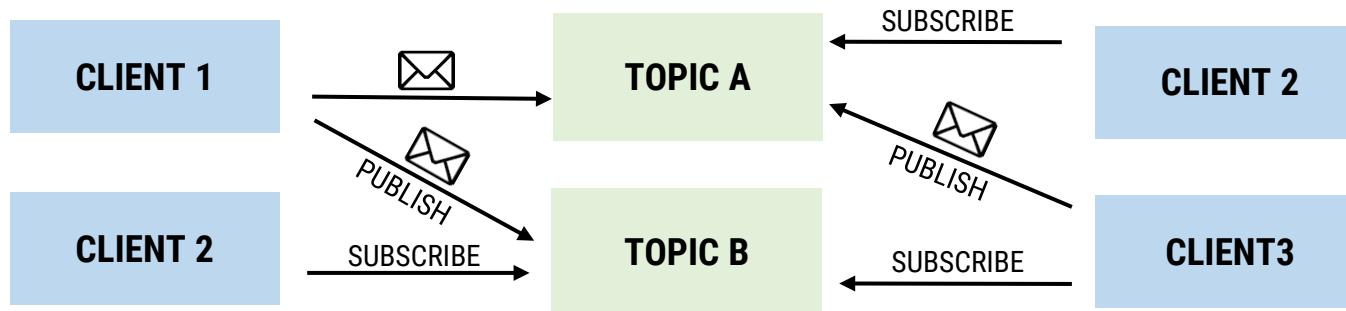
- Half-Duplex: Nur Client kann Nachrichten initiieren.
- Full-Duplex: Sowohl Client als auch Service können Nachrichten initiieren.



# Publish-Subscribe Muster

Ziel: Datenverteilung.

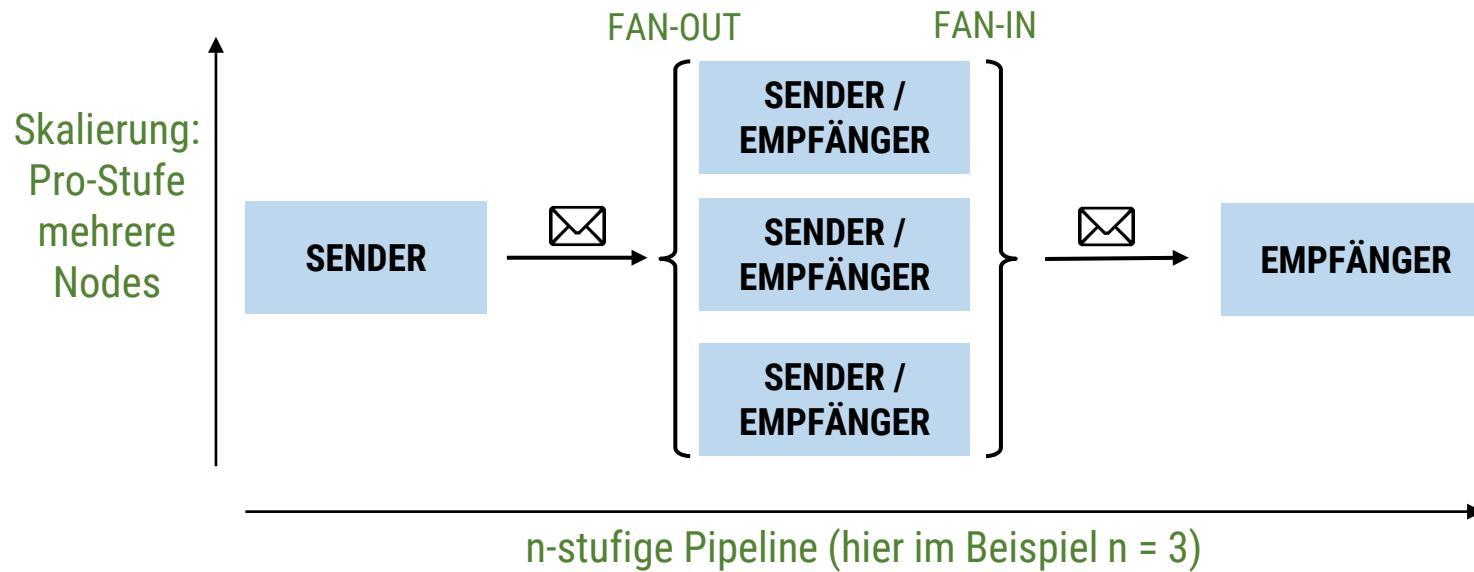
- Verbindet eine **Menge von Publishers** mit einer **Menge von Subscribers**.
- Verbreitet in Enterprise- (ActiveMQ/Websphere) und IoT-Umfeld (-> MTTQ).



# Pipeline-Muster

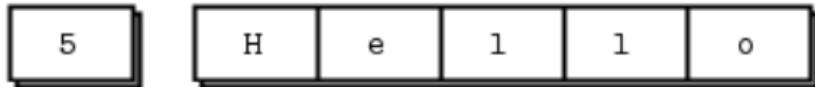
## Ziel: Aufgabenverteilung

- Verbindet mehrere Nodes in einem Fan-out / Fan-in Pattern.
- Das Pattern kann **mehrere Stufen** und **sogar Schleifen** haben.
- Auch bekannt als parallele Aufgabenverteilung und -zusammenführung.



# Technologie: ZeroMQ

- Library für messageorientierte Kommunikation.
- Messages werden i.d.R. asynchron gesendet.
- Bekanntes Socket-Prinzip auf höherem Abstraktionsniveau.
  - Spezifische Sockets für bestimmte Kommunikationsmuster.
- Messageinhalt beliebig (bytes oder Text).
- Framing basierend auf Länge



# ZeroMQ: Sockets passend zu Kommunikationsmustern.

Pattern	Socket	Einsatzgebiet
Request-Reply	REQ	Sendet Anfrage (an RES-Socket). Half-Duplex.
	RES	Beantwortet Anfragen (von REQ-Sockets). Half-Duplex.
PubSub	PUB	Publiziert Message unter bestimmtem Topic.
	SUB	Empfängt Messages für abonnierte Topics.
Pipeline	PUSH	Sendet Message an Verarbeiter (PULL-Socket).
	PULL	Empfängt Message zur Verarbeitung (von PUSH-Socket).

Auswahl: ZeroMQ kennt weitere Sockets.

# ZeroMQ Beispiel 1: Half-Duplex Echo (Client)

```
public static final String ADDRESS = "tcp://localhost:5555";

public static void main(String[] args) {
    try (ZContext context = new ZContext()) {
        LOG.info("Echo client connecting to " + ADDRESS);
        ZMQ.Socket socket = context.createSocket(SocketType.REQ);
```

REQ: Socket für Requests

```
        socket.connect(ADDRESS);
```

Clients benutzen connect

```
        BufferedReader in = new BufferedReader(new InputStreamReader(System.in));
        while(true) {
            String input = in.readLine();
            socket.send(input.getBytes(ZMQ.CHARSET));
            byte[] reply = socket.recv();
            LOG.info("Received " + new String(reply, ZMQ.CHARSET));
        }
    } catch (IOException ex) {
        LOG.error(ex.getMessage());
    }
}
```

REQ/REP: Auf jedes send muss ein recv folgen

# ZeroMQ Beispiel 1: Half-Duplex Echo (Server)

```
public static final String ADDRESS = "tcp://localhost:5555";  
  
public static void main(String[] args) {  
    try (ZContext context = new ZContext()) {  
        ZMQ.Socket socket = context.createSocket(SocketType.REP);  
  
        socket.bind(ADDRESS);  
  
        while (true) {  
            byte[] reply = socket.recv();  
            LOG.info("Received message " + new String(reply, ZMQ.CHARSET));  
            socket.send(reply);  
        }  
    }  
}
```

REP: Socket für Replies

Server benutzen bind

REQ/REP: Auf jedes recv muss ein send folgen

## ZeroMQ Beispiel 2: Data Distribution (Publisher)

```
public static final String ADDRESS = "tcp://localhost:5555";  
  
public static void main(String[] args) {  
    try (ZContext context = new ZContext()) {  
        ZMQ.Socket socket = context.createSocket(SocketType.PUB);  
  
        socket.bind(ADDRESS);  
  
        while (true) {  
            long station = (long) Math.floor(Math.random() * 3.0);  
            long temp = 15 + (long) Math.floor(Math.random() * 10.0);  
  
            String message = "Temp/" + station + "/" + temp;  
  
            socket.send(message.getBytes(ZMQ.CHARSET));  
            LOG.info("Published '" + message + "'");  
  
            Thread.sleep(1000);  
        }  
    } catch (InterruptedException e) {  
        LOG.info("interrupted"); // cannot occur  
    }  
}
```

PUB: Socket für Publisher

Publisher benutzen bind

Topic implizit definiert durch Message-Prefix

## ZeroMQ Beispiel 2: Data Distribution (Subscriber)

```
public static final String ADDRESS = "tcp://localhost:5555";

public static void main(String[] args) {
    String topic = args[0];

    try (ZContext context = new ZContext()) {
        ZMQ.Socket socket = context.createSocket(SocketType.SUB);

        socket.connect(ADDRESS);
        socket.subscribe(topic);

        while (true) {
            byte[] message = socket.recv();

            LOG.info("Received: " + new String(message, ZMQ.CHARSET));
        }
    }
}
```

SUB: Socket für Subscriber

Subscriber benutzen bind

Bestimmten Topic abonnieren (Message-Prefix)

Nur Messages der abonnierten Topics

# ZeroMQ Beispiel 3: Taskverarbeitung (Producer)

```
public static final String ADDRESS = "tcp://localhost:5555";

public static void main(String[] args) {
    try (ZContext context = new ZContext()) {
        LOG.info("Providing tasks on " + ADDRESS);
        // Socket to talk to server
        ZMQ.Socket socket = context.createSocket(SocketType.PUSH);
```

PUSH: Socket für Producer

```
socket.bind(ADDRESS);
```

bind für Fan-out  
connect für Fan-in

```
int i = 0;
while(true) {
    String workPackage = "Work Package #" + ++i;
    socket.send(workPackage.getBytes(ZMQ.CHARSET));
    LOG.info("Send task '" + workPackage + "'");
    Thread.sleep(1000);
}
```

Blockiert bis Consumer verbunden sind

```
} catch (InterruptedException e) {
    LOG.info("interrupted"); // cannot occur
}
}
```

# ZeroMQ Beispiel 3: Taskverarbeitung (Consumer)

```
public static final String ADDRESS = "tcp://localhost:5555";

public static void main(String[] args) {
    try (ZContext context = new ZContext()) {
        LOG.info("Listening for tasks on " + ADDRESS);
```

PULL: Socket für Consumer

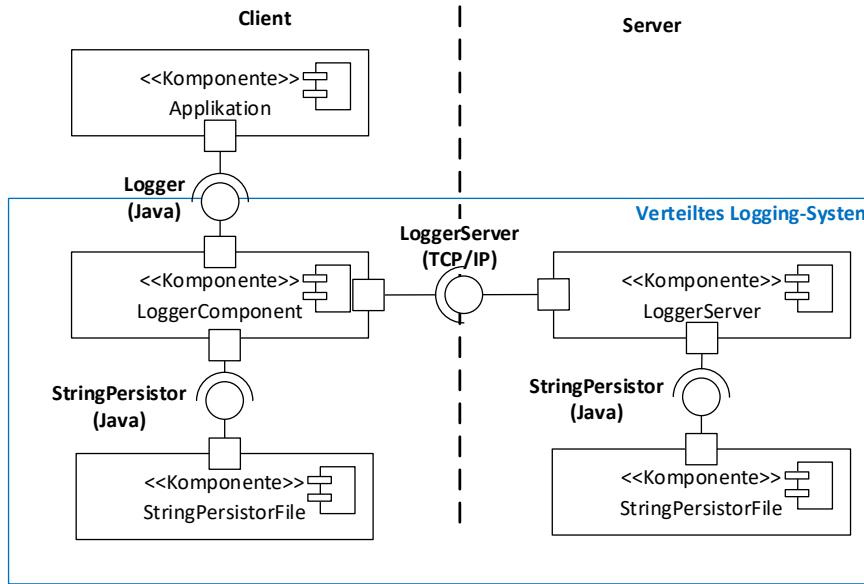
```
        // Socket to talk to server
        ZMQ.Socket socket = context.createSocket(SocketType.PULL);
        socket.connect(ADDRESS);
```

Connect für Fan-out  
Bind für Fan-in

```
        while(true) {
            byte[] bytes = socket.recv();
            LOG.info("Received task '" + new String(bytes, ZMQ.CHARSET) + "'");
            Thread.sleep(3000);
            LOG.info("Processed task '" + new String(bytes, ZMQ.CHARSET) + "'");
        }
    } catch (InterruptedException e) {
        LOG.error("interrupted"); // cannot occur
    }
}
```

# Diskussion: Kommunikationsmuster

*Diskutieren Sie zu zweit, ob und wo Sie Kommunikationsmuster im Logger verwenden (können). Sehen Sie Vorteile/Nachteile?*



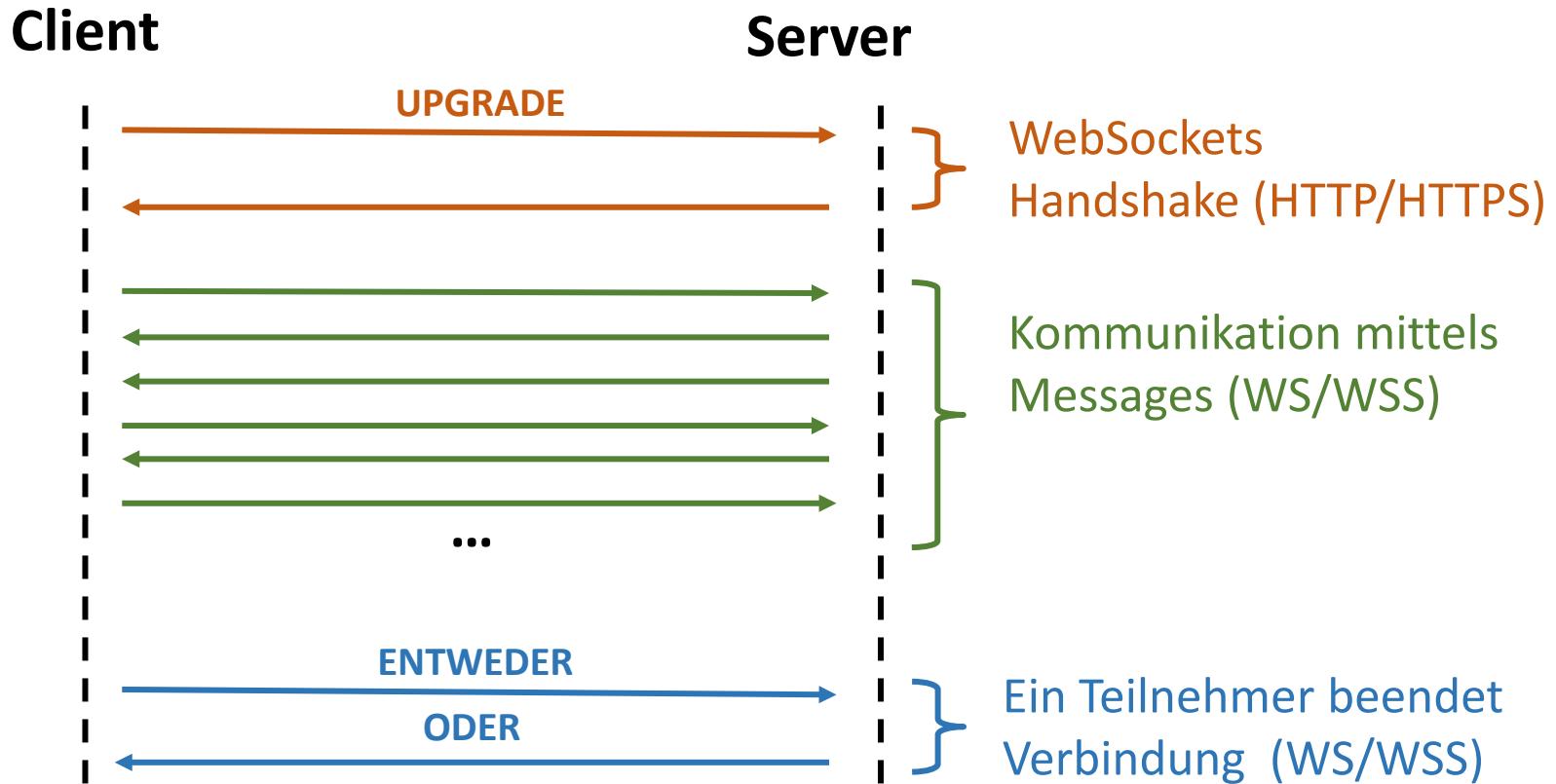
*Angenommen ein oder mehrere Logger-Viewer, welche Log-Messages in Real-Time anzeigen sollen, sollen an das verteilte Logger-System via TCP/IP angebunden werden. Wie sieht die Situation dann aus?*

*Wir besprechen die Resultate im Plenum.*

# Protokoll: WebSockets

- Full-Duplex-Kommunikation.
- Protokoll auf Anwendungslevel (KEIN Transportprotokoll wie TCP/UDP).
- Basiert auf HTTP -> Protokollupgrade.
- RFC 6455 <https://tools.ietf.org/html/rfc6455>
- Viele Implementationen: Tomcat, Jetty, Nginx, Apache, usw.

# WebSockets: Kommunikationsverlauf



# WebSockets: Handshake

## Client

```
GET /chat HTTP/1.1
Host: server.example.com
Upgrade: websocket
Connection: Upgrade
Sec-WebSocket-Key: x3JJHMbDL1EzLkh9GBhXDw==
Sec-WebSocket-Protocol: chat, superchat
Sec-WebSocket-Version: 13
Origin: http://example.com
```

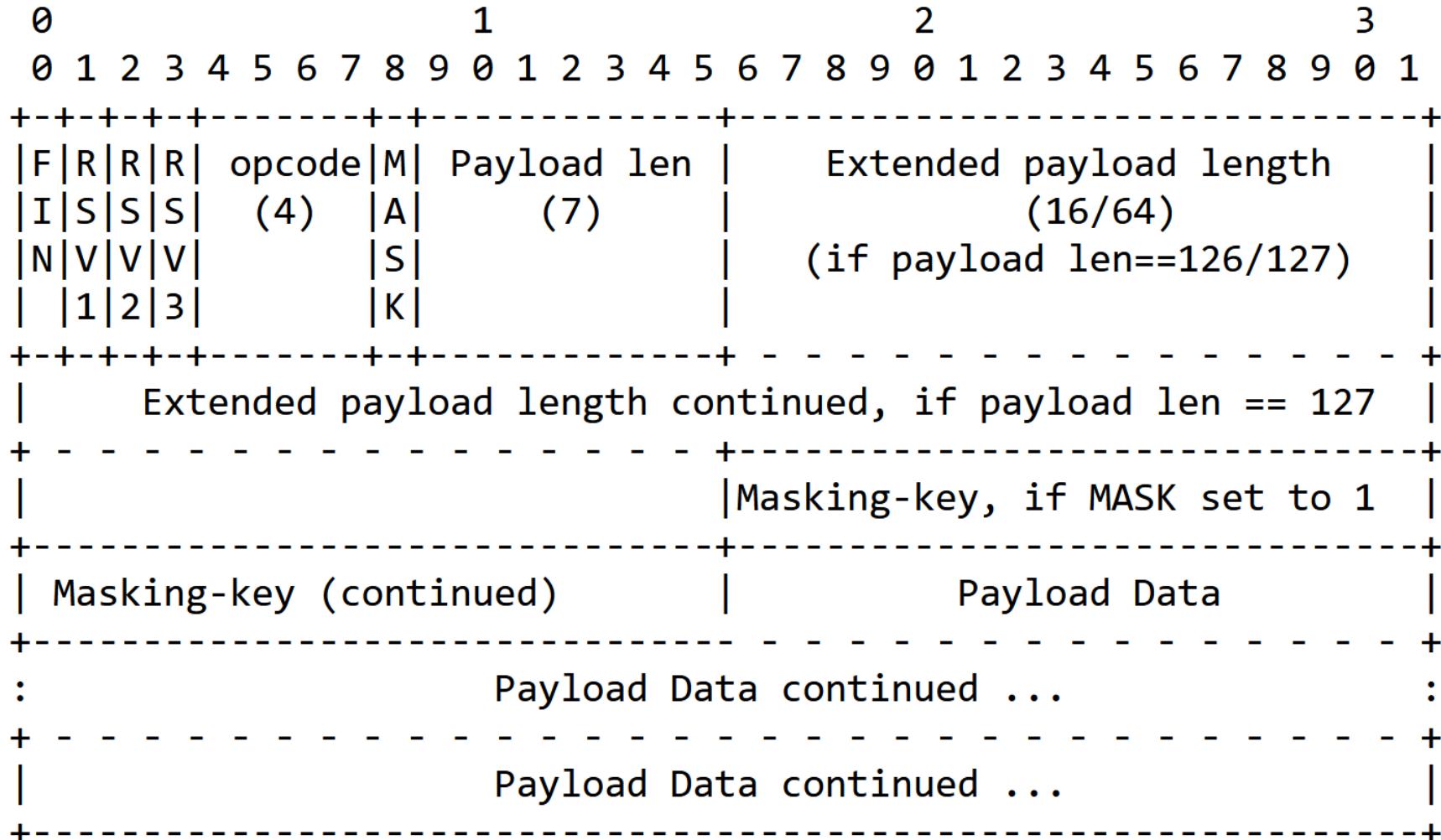
## Server

```
HTTP/1.1 101 Switching Protocols
Upgrade: websocket
Connection: Upgrade
Sec-WebSocket-Accept: HSmrC0sM1YUkAGmm5OPpG2HaGwk=
Sec-WebSocket-Protocol: chat
```

Quelle: <https://www.rfc-editor.org/rfc/rfc6455>

# WebSocket Transport

**Binärprotokoll:** Messages übermittelt via TCP mittels Frames.



Quelle: <https://www.rfc-editor.org/rfc/rfc6455>

# WebSockets Server Endpoint API (Beispiel Jetty Server API)

org.eclipse.jetty.websocket.api.WebSocketListener

Methode	Beschreibung
<b>void onWebSocketConnect( Session session)</b>	Client hat sich verbunden.
<b>void onWebSocketClose( int statusCode, String reason)</b>	Session wurde clientseitig geschlossen.
<b>void onWebSocketBinary(<b>byte[]</b> payload, int offset, int length)</b>	Binärdaten empfangen.
<b>void onWebSocketText( String message)</b>	Textnachricht empfangen (vollständige Message in String enthalten).
<b>void onWebSocketError( Throwable cause)</b>	Fehler aufgetreten.

# Beispiel: Chat-Server mit WebSockets (Session Manager)

```
public class ChatSessions {  
    private final List<Session> sessions = new LinkedList<>();  
  
    public synchronized void add(Session session) {  
        sessions.add(session);  
    }  
    public synchronized void remove(Session session) {  
        sessions.remove(session);  
    }  
  
    public synchronized void broadcast(String message) {  
        Iterator<Session> it = sessions.iterator();  
        while (it.hasNext()) {  
            Session session = it.next();  
            try {  
                session.getRemote().sendString(message);  
            } catch (IOException e) {  
                it.remove();  
                LOG.error("removed session from broadcast list"  
                        + session.getRemoteAddress() + " due to IOException");  
            }  
        }  
    }  
}
```

Alle Operationen auf Sessionliste sind Thread-safe

Sende Nachricht an Session

# Beispiel: Chat-Server mit WebSockets (Endpoint)

```
public class ChatEndPoint implements WebSocketListener {  
    private static final ChatSessions CHAT_SESSIONS = new ChatSessions();  
    private Session session;  
    private String username;  
  
    public void onWebSocketConnect(Session session) {  
        this.session = session;  
        CHAT_SESSIONS.add(session);  
    }  
    public void onWebSocketClose(int statusCode, String reason) {  
        CHAT_SESSIONS.remove(session);  
    }  
    public void onWebSocketError(Throwable cause) {  
        CHAT_SESSIONS.remove(session);  
        LOG.error("web socket error occurred" , cause);  
    }  
    public void onWebSocketText(String message) {  
        if (username == null) {  
            username = message;  
        } else {  
            CHAT_SESSIONS.broadcast(username + ":" + message);  
        }  
    }  
}
```

Zustandsbasiertes-Protokoll

# Beispiel: Chat-Server mit WebSockets (Einbettung in Jetty)

```
private static final int PORT = 8080;
```

Beispiel zur Illustration

```
public static void main(String[] args) throws Exception {
    Server server = new Server(PORT);
    ServletContextHandler servletContextHandler = new ServletContextHandler();
    server.setHandler(servletContextHandler);

    Servlet websocketServlet = new JettyWebSocketServlet() {
        @Override protected void configure(JettyWebSocketServletFactory factory) {
            factory.setIdleTimeout(Duration.ofMinutes(30));
            factory.addMapping("/", (req, res) -> new ChatEndPoint());
        }
    };
    servletContextHandler.addServlet(new ServletHolder(websocketServlet), "/chat");
    JettyWebSocketServletContainerInitializer.configure(servletContextHandler, null);

    server.start();
    LOG.info("Chat server listening on port " + PORT);
}
```

# Übung: Chatserver mit WebSockets

*Der Chatserver interpretiert die erste Message als Benutzernamen und alle nachfolgenden Messages als Chatnachrichten.*

## Fragen:

- Ist dieses Protokoll robust? Warum?
- Wie müsste man das Protokoll ändern um explizite IDs hinzufügen?
- Vorteile/Nachteile?

*Der Chatserver sendet beim Eintreffen einer Nachricht eines Benutzers eine Push-Nachricht an alle Benutzer.*

## Fragen:

- In welchen Szenarios ist dieses Push-Verfahren besser als ein Pull-Verfahren (Client fragt nach, ob neue Nachrichten da sind)?
- Könnte es mit dem Push-Verfahren Probleme geben, welche?

**Besprechen Sie diese Fragen zu zweit (oder dritt) während 10 Minuten. Wir diskutieren anschliessend im Plenum.**

# **Persistente Messageorientierte Kommunikation**

# Zuverlässige Messagetransfer

**Zuverlässige Auslieferung:** Eine Message wird ausgeliefert und zwar exakt einmal.

**Unzuverlässige Auslieferung:** Eine Message wird entweder gar nicht oder mehr als einmal ausgeliefert.

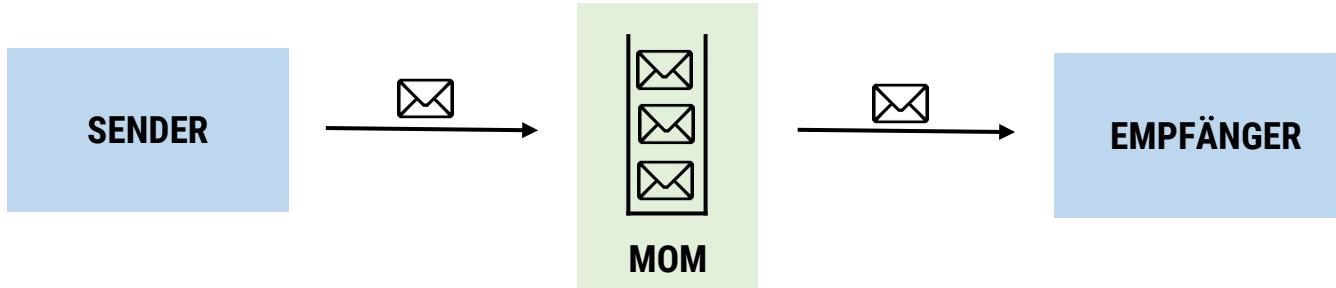
## Ursachen:

- Messages über unzuverlässigen Kanal gesendet.
- Messages verworfen (Queue voll).
- Systeme laufen nicht immer.
- Prozesse oder Systeme können während Verarbeitung ausfallen.

# Persistente messageorientierte Kommunikation

Message-Oriented-Middleware (MOM):

- Zwischenspeicherung von Messages.
- Zeitversetzte Kommunikation.



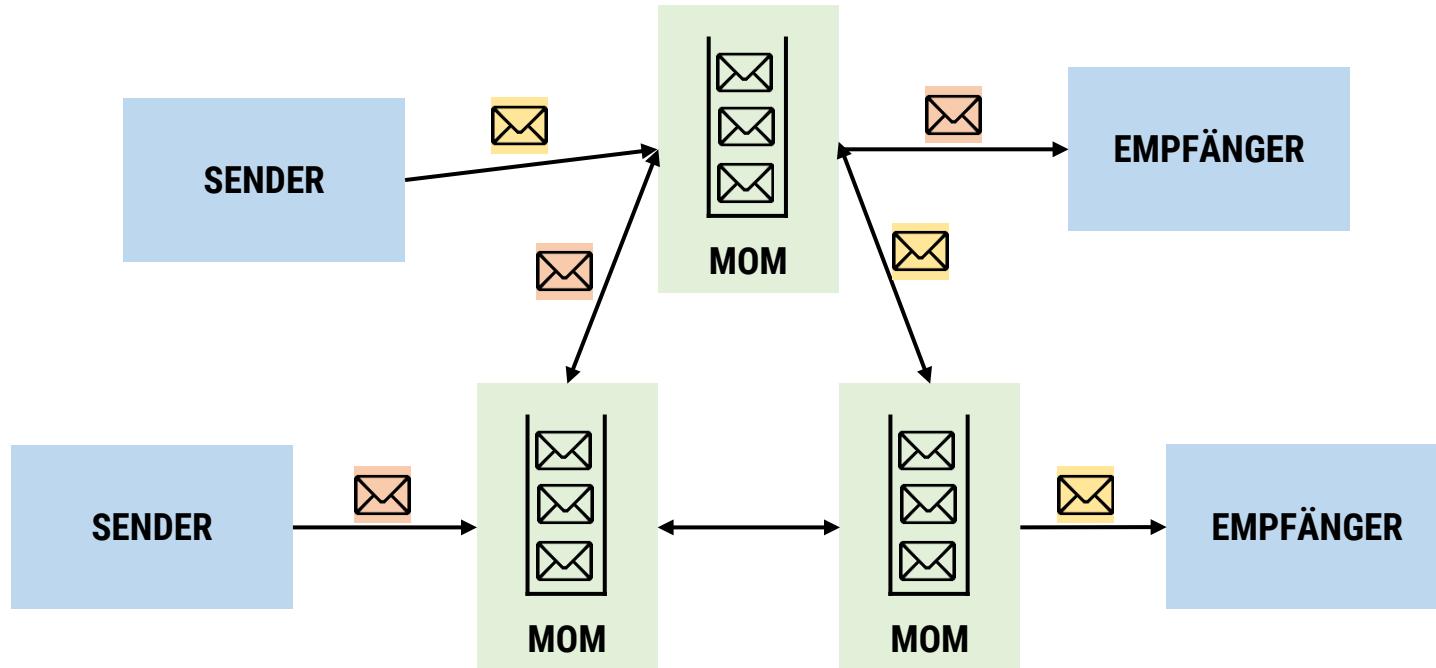
- Sender oder Empfänger müssen nicht aktiv sein während Übertragung.
- Unterstützt Transfers, welche Minuten dauern (anstelle von ms).

**Populäre MOMs:**

- Apache ActiveMQ/ ActiveMQ Artemis
- RabbitMQ
- Websphere MQ

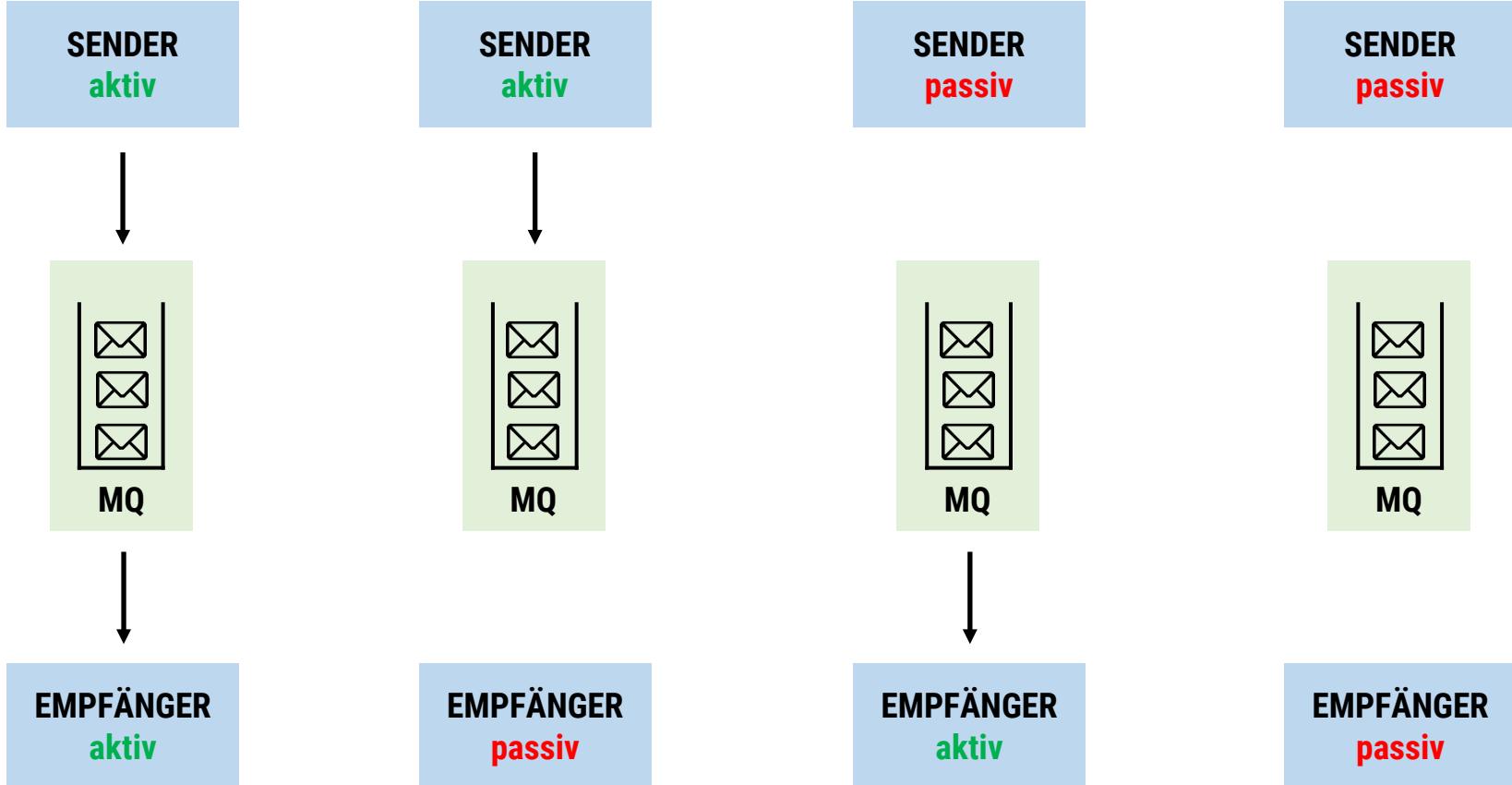
# Message-Queuing-Model

- Kommunikation mittels Einfügen von Messages in spezifische Warteschlange.
- Keine Garantien, **wann** eine Message gelesen wird.
- Message werden von Kommunikationsservern weitergeleitet, bis zum Ziel.
  - Oft sind die Kommunikationsserver direkt miteinander verbunden.
- Typischerweise eine Queue pro Kommunikationsteilnehmer.
  - Queue kann aber geteilt werden.



# Zeitliche Entkopplung mittels Message-Queues

Vier Kombinationen:



# Typisches Message-Queue-API

Operation	Beschreibung
put()	Füge eine Message einer bestimmten Queue hinzu.
get()	Warte bis eine Queue nicht mehr leer ist und retourniere die erste Message in der Queue. <b>=&gt; blockierend</b>
poll()	Überprüfe ob eine Queue mindestens eine Message enthält. Ist dies der Fall ist, retourniere die erste Message in der Queue. <b>=&gt; nicht blockierend</b>
notify(callback)	Registriere eine Callback-Funktion, welche aufgerufen wird, sobald eine Message in dieser Queue eintrifft.

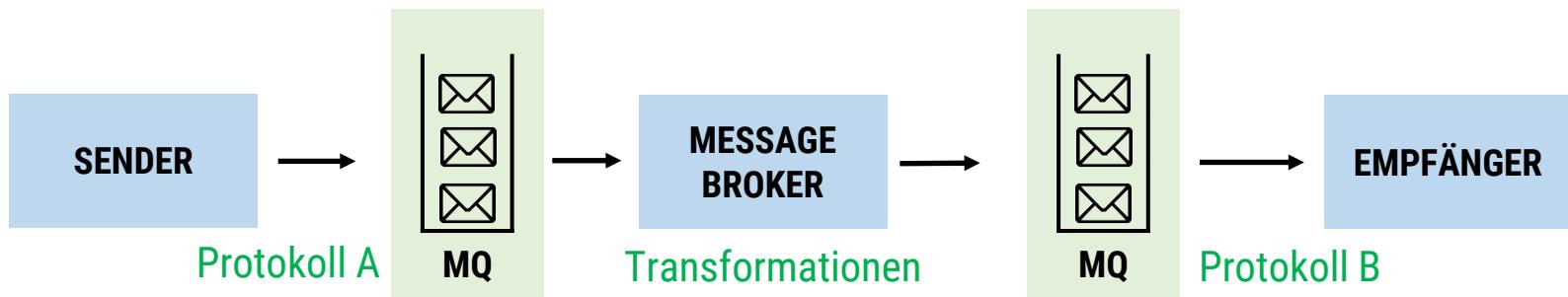
# Populäre Message-Queuing-Protokolle

- AMQP: Ubiquitous, secure, reliable and open internet protocol for handling business messaging.
- MQTT: light weight, client to server, publish / subscribe messaging protocol.  
*Oft im IoT-Bereich verwendet.*
- STOMP: text-orientated wire protocol.
- XMPP: eXtensible Messaging and Presence Protocol.
- [OpenWire: Proprietäres Protokoll von ActiveMQ.]

# Message-Broker

**Ziel:** Integration von heterogenen Applikationen.

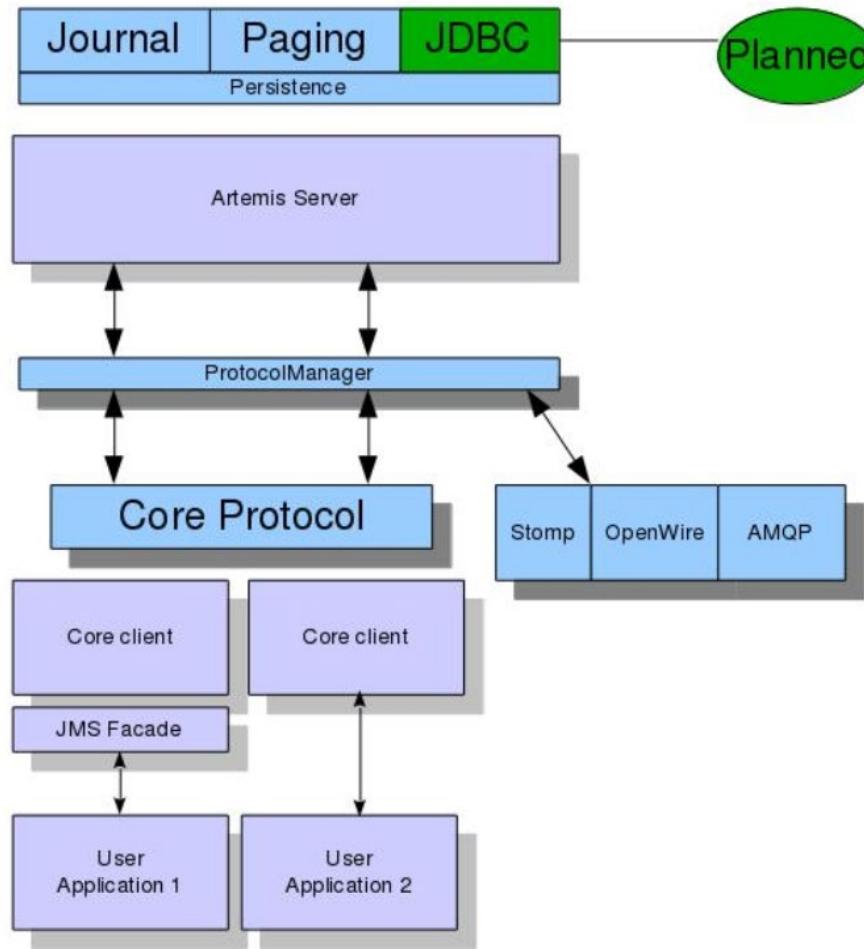
- System mit verschieden Protokollen und Nachrichtenformaten verbinden.
- Alles auf den gemeinsamen Nenner bringen hätte Komplexität  $N \times N$ .
- Broker wandeln zwischen Protokollen und Nachrichtenformaten und routen Messages zu verschiedenen Ziel (ggf. mit Publish-Subscribe).



*Konzeptionell auf gleicher Stufe wie Sender oder Empfänger, oft aber direkt mit einem MQ-System verknüpft.*

# Fallbeispiel: Apache ActiveMQ Artemis

- open source, multi-protocol, Java-based message broker
- <https://activemq.apache.org/>
- Architektur:



Quelle: <https://activemq.apache.org/>

# Fallbeispiel: Broker und Queue erstellen, Admin-Interface

```
## Neuen Broker erstellen
artemis create mybroker --user admin --password admin --require-login

## Broker starten
mybroker\bin\artemis run

## Neue Queue erstellen
mybroker\bin\artemis queue create --user admin --password admin --address
demoQueue --anycast --name demoQueue --auto-create-address --durable --
preserve-on-no-consumers

## Webinterface
http://localhost:8161
```

# Fallbeispiel: ActiveMQ Artemis Producer

```
public static void main(String args[]) throws Exception {
    ServerLocator locator =
        ActiveMQClient.createServerLocator(ARTEMIS_LOCATION);
    ClientSessionFactory factory = locator.createSessionFactory();
    ClientSession session = factory.createSession(
        "admin", "admin", true, false, false, false, 1);

    ClientProducer producer = session.createProducer("demoQueue");

    BufferedReader userIn = new BufferedReader(new InputStreamReader(System.in));
    while (true) {
        session.start();
        ClientMessage message = session.createMessage(true);
        message.getBodyBuffer().writeString(userIn.readLine());
        producer.send(message);
        LOG.info("send message: " + message.getBodyBuffer().readString());
        session.commit();
    }
}
```

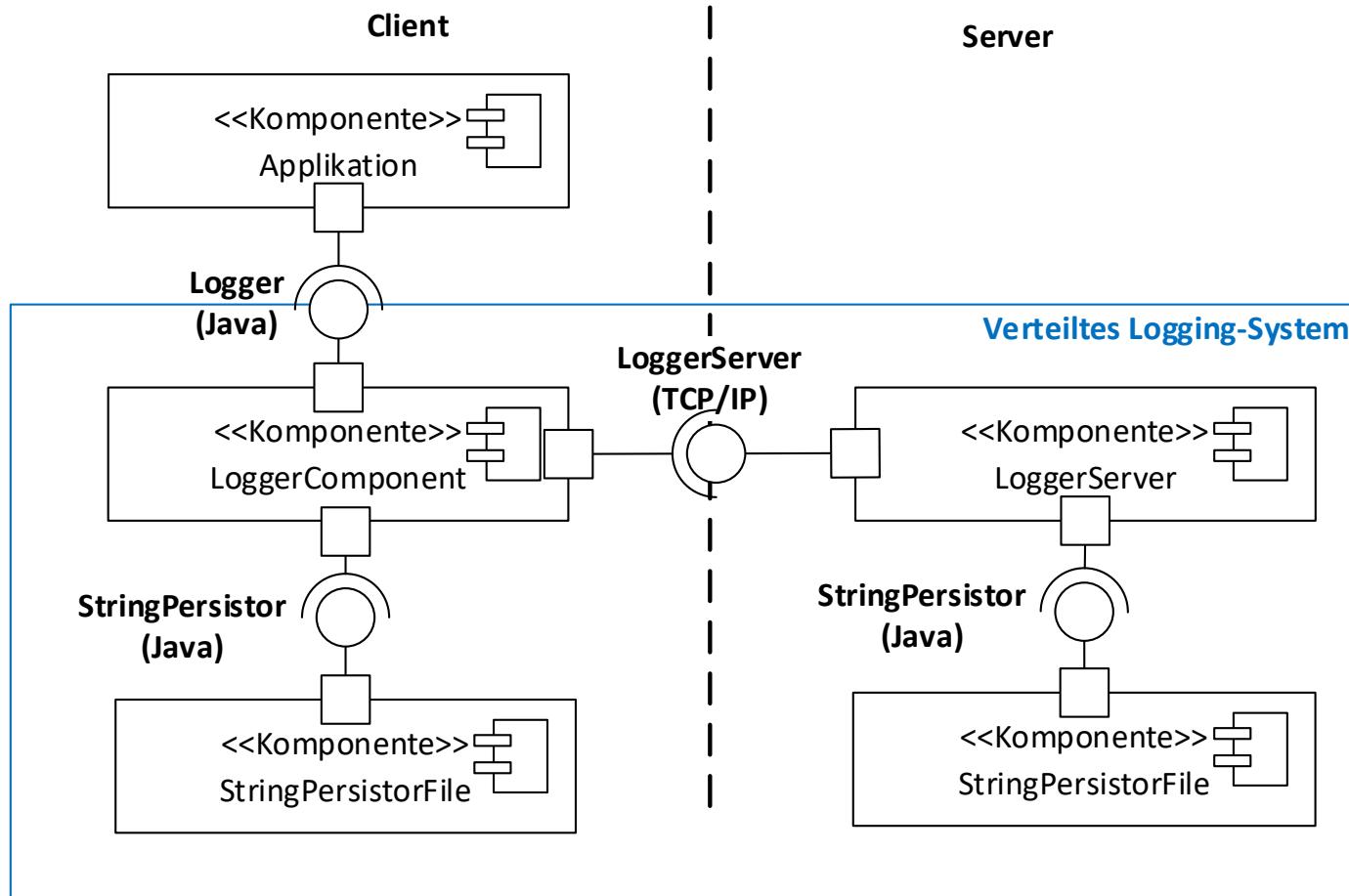
# Fallbeispiel: ActiveMQ Artemis Consumer

```
public static void main(String args[]) throws Exception {
    ServerLocator locator =
        ActiveMQClient.createServerLocator(ARTEMIS_LOCATION);
    ClientSessionFactory factory = locator.createSessionFactory();
    ClientSession session = factory.createSession(
        "admin", "admin", true, false, false, false, 1);

    ClientConsumer consumer = session.createConsumer("demoQueue");

    while(true) {
        session.start();
        ClientMessage message = consumer.receive();
        LOG.info("received msg: " + message.getBodyBuffer().readString());
        session.commit();
    }
}
```

# Diskussion: Wann persistente Kommunikation einsetzen?



# Zusammenfassung

- Für das Design einer messageorientierten Kommunikation sind verschiedene Anforderungen festzulegen: Form der Nachrichten, Art der Kommunikation (transient, persistent, asynchron, synchron).
- Verschiedene Kommunikationsmuster (Request-Reply, Publish-Subscribe, Pipeline) bestimmen die Architektur einer Applikation.
- Middleware (Beispiel: ZeroMQ, WebSockets) unterstützen beim Implementieren dieser Kommunikationsmuster.
- Persistente Message-Queues ermöglichen zeitliche Entkopplung von Komponenten oder Systemen.
- Message-Broker ermöglichen Integration heterogener Systeme.

# Literatur

- Distributed Systems (3rd Edition), Maarten van Steen, Andrew S. Tanenbaum, Verleger: Maarten van Steen (ehemals Pearson Education Inc.), 2017.
- ZeroMQ, by Pieter Hintjens, O'Reilly Media, Inc., 2013.
- Apache ActiveMQ Artemis 2.26.0 User Manual,  
<https://activemq.apache.org/components/artemis/documentation/>

# **Fragen?**

Verteilte Systeme und Komponenten

# **Dependency Management**

Roland Gisler

# Inhalt

- Einleitung
- Maven Repositories.
- Dependency Management bei Java.
- Dependency Management mit Apache Maven.
- Dependency Scopes.
- Transitive Dependencies und Konflikte.
- Versionierung von Dependencies / Snapshots.
- «Managed Dependencies» in Multimodul-Projekten.
- Deployment von Dependencies.

# Lernziele

- Sie haben ein grundsätzliches Verständnis von Dependency Management.
- Sie wissen wie am Beispiel von Java und Apache Maven das Dependency Management funktioniert.
- Sie sind mit den Begriffen «dependency scopes» und «transitiv dependencies» vertraut und können diese erklären.
- Sie kennen das Versionskonzept und die Funktionsweise von Snapshots.
- Sie wissen auf welche Art Dependencies deployed werden.

# **Einleitung**

# Einleitung - Dependency Management (DM)

- Das Dependency Management beschreibt die Organisation und die Techniken für den Umgang mit Abhängigkeiten zu anderen Modulen.
  - Modul wird in diesem Kontext vereinfacht als Überbegriff für Package, Library, Bundle oder Komponente verwendet.
  - Häufig wird auch der Begriff «Package Management» verwendet.
- Abhängigkeiten können sowohl auf interne und externe Modulen bestehen.
  - Intern: Modul im selben Projekt.
  - Extern: Dritt-Modul, aus anderem Projekt oder Organisation.
- Abhängigkeiten werden typisch in binärer (kompilierter) Form aufgelöst. Deshalb kommen dafür so genannte Binär-Repositories und Packagemanager(-Tools) zum Einsatz.

# Beispiele für Dependency Management (DM)

- Einige populäre Systeme / Repositories für DM und PM:
  - **NuGet** – Package Manager für .NET-Plattform.
  - **apt** – Advanced Packaging Tool – Packetverwaltung für Linux.
  - **Yum** – Yellowdog Updater, Modified – Packetverwaltung Linux.
  - **P2** – OSGi-basiertes Komponentensystem, Eclipse Equinox.
  - **npm** – Node Package Manager für JavaScript / node.js.
  - **Gems** – Packetmanager für Ruby.
- Allen gemeinsam ist: Zentrale Ablage auf Server, ein standardisiertes Format, zusätzliche Metainformationen, typisch mit Abhängigkeiten (Dependencies) versehen, Sicherung der Konsistenz (z.B. über hash-Mechanismen), geregelte Zugriffsprotokolle, Suchmöglichkeiten etc.

# **Maven Repository**

# Maven Repository

- Es gibt verschiedene öffentliche Repos (OSS):
  - Maven Central: <https://search.maven.org/> (das Original!)
  - Maven Central: <https://central.sonatype.dev/> - neues UI
  - Google Maven Repo: <https://maven.google.com/>
- Natürlich hat man auf öffentliche Repo's keine Schreibrechte!
  - Bzw. nur ausgewählte Personen über definierte Prozesse.
- Organisationen betreiben typisch **interne** Repositories.
  - Ursprünglich realisiert als einfach Dateisysteme / Webserver.
- Professionelle Produkte mit diversen Zusatzfunktionen:
  - Apache Archiva - <https://github.com/apache/archiva>
  - JFrog/Artifactory - <https://jfrog.com/community/open-source/>
  - Sonatype/Nexus - <https://de.sonatype.com/products/nexus-repository/>

# Maven Repository an der HSLU

- Firmen und Organisationen nutzen ein eigenes Repository oft auch als Mirror von öffentlichen Repositories.
- HSLU RepoHub Nexus: <https://repohub.enterprise.hslu.ch>
  - Dient auch als Mirror zu öffentlichen Repositories.
- Alle heruntergeladenen Pakete werden zudem in einem lokalen Cache gespeichert (caching).  
Dieses Repository: **\$HOME/.m2/repository**

**Jetzt aber wirklich ALLE! ;)**

- Damit dieses HSLU-Repository verwendet wird, müssen Sie in das Verzeichnis **\$HOME/.m2** die Konfigurationsdatei **settings.xml** (auf ILIAS zur Verfügung gestellt) kopieren!
  - Haben Sie das gemacht? Kontrolle: Achten Sie während des Builds auf die URL's beim Download der Dependencies.

# **Dependency Management für Java**

# Dependency Management für Java

- Binäre Module (kompilierte Projekte) werden bei Java typisch als JAR-Dateien (oder EAR, WAR, RAR, JMOD etc.) ausgetauscht.
- Java selber kennt zwar seit Version 9 neue Mechanismen zur Modularisierung und Definition von Abhängigkeiten, das umfasst aber **nicht** die Versionierung und die Ablage.
- Ursprünglich wurden deshalb die JAR-Dateien einfach von Hand z.B. in **/lib**-Verzeichnisse direkt in die Projekte kopiert.
  - Fehleranfällig, hohe Redundanz, grosser Platzbedarf, ...
  - Ergebnis: «JAR Hell», wenn sich Entwickler\*innen nicht aktiv um die Abhängigkeiten (im Classpath) gekümmert haben.
- Im Jahr 2001 führte Apache Maven mit seinem Buildsystem auch ein einfaches Dependency Management ein, das zum Quasi-Standard wurde und sehr populär ist.

# Dependency Management (DM) mit Maven Repositories

- Grundsätzlich sollte man Unterscheiden zwischen:
  - Das **Format** für die zentralen Ablage der meist binären Artefakten mit zusätzlichen Metainformation im → Repository.
  - Das **Werkzeug** welches es ermöglicht, Artefakte von Repositories zu suchen, zu beziehen, zu deployen und ggf. auch zu verwalten.
- Während beim Format der Repositories Maven zum Quasi-Standard geworden ist, gibt es bei den Werkzeugen eine grosse Vielfalt:
  - Apache Ivy – einziges «reines» Dependency-Management Tool.
  - Apache Maven – in Buildtool integriert, das «Original».
  - Das DM zahlreicher weiterer Buildtools basiert auch auf Maven-Repos: Buildr, Groovy Grape, Gradle/Grails, SBT etc.

# **Dependency Management mit Apache Maven**

# Maven - Identifikation

Eine Maven Projekt identifiziert sich über drei Attribute, die als «maven coordinates» bezeichnet werden:

- **GroupId:** Meistens zusammengesetzt aus dem «reverse domain name» der Organisation und einem Zusatz für eine OE, eine Projektgruppe etc. (→ grosse Ähnlichkeit zu Java Package Name)  
Beispiel: **ch.hslu.vsk**
- **ArtifactId:** Entspricht häufig dem Namen des Projektes bzw. den darin enthaltenen Modulen.  
Beispiel: **stringpersistor-api**
- **Version:** Empfohlen wir eine dreistellige Versionsnummer (Semantic Versioning, <http://semver.org>).  
Beispiel: **6.0.2**

# Maven Coordinates

- Die Identifikation eines Projektes ist somit eines der wichtigsten Pflichtelemente in einem POM (`pom.xml`):

```
<groupId>ch.hslu.vsk</groupId>
<artifactId>stringpersistor-api</artifactId>
<version>6.0.2</version>
```

- Mittels diesen Koordinaten wird/soll eine Dependency weltweit absolut eindeutig identifiziert werden können!
- Weitere Beispiele in Kurzform:
  - `org.apache.logging.log4j:log4j-api:2.19.1` – ok.
  - `nl.jqno.equalsverifier>equalsverifier:3.10.1` – ok.
  - `junit:junit:4.12` – eine alte «Sünde».
  - `org.junit.jupiter:junit-jupiter-api:5.9.1` – besser!
  - `ch.hslu.vsk:stringpersistor-api:6.0.2` – perfekt! ☺

# Deklaration von Dependencies im POM

- Benötigte Dependencies können im POM (`pom.xml`) eines Projektes im Element `<dependencies>` eingetragen werden:

```
<dependency>
    <groupId>ch.hslu.vsk</groupId>
    <artifactId>stringpersistor-api</artifactId>
    <version>6.0.2</version>
    <scope>compile</scope>
</dependency>
```

- Diese werden beim Build automatisch vom Repository (default: Maven Central) herunter geladen, und im lokalen Repository (`$HOME/.m2/repository`) gespeichert.
- Der Buildprozess referenziert die Artefakte (typisch: JAR-Dateien) dort mit einem entsprechenden Classpath.

# **Dependency Scopes**

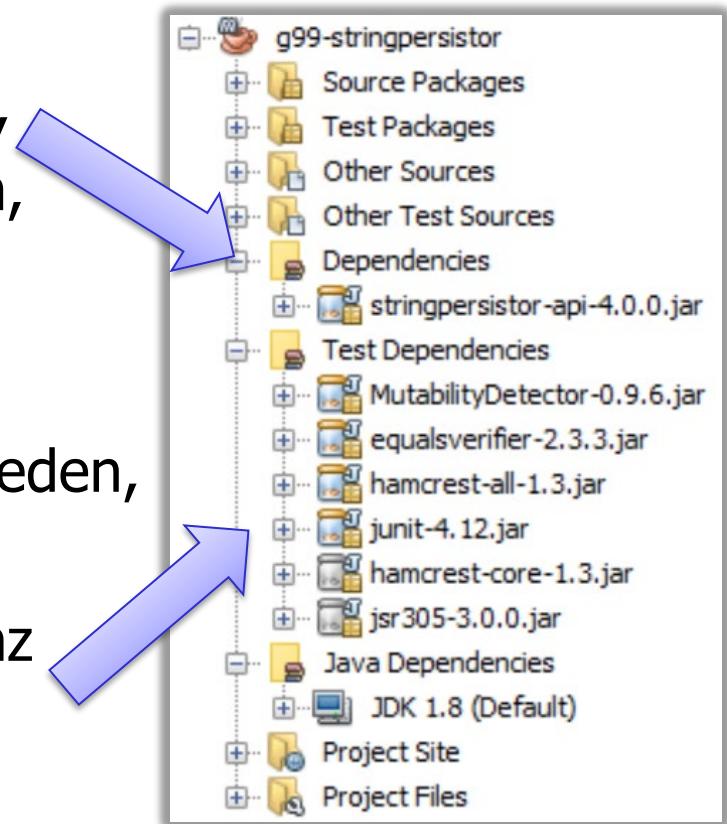
# Dependency Scopes

- Im Beispiel oben fällt das Element `<scope>compile</scope>` auf, das optional pro Abhängigkeit definiert werden kann.
  - Damit wird der Zweck und Geltungsbereich (→ Scope) der Dependency qualifiziert, **was unbedingt empfohlen wird!**
- Maven kennt verschiedene Scopes (hier nur die wichtigsten drei):
  - **compile** – Dependency wird für die Kompilation und zur Laufzeit des Programmes benötigt (Default).
  - **test** – Dependency ausschliesslich für die Kompilation und Ausführung der Testfälle (Beispiele: JUnit, AssertJ etc.).
  - **runtime** – Dependency nur für Laufzeit, aber nicht für Kompilation, z.B. für dynamisch geladene Implementationen.
- Aus den Scopes werden in Maven **spezifische** Classpaths!
  - Daraus ergibt sich eine **implizite Verifikation** des Designs.

# Dependency Scopes in der IDE

Hier zeigen sich die IDE's unterschiedlich Intelligent:

- NetBeans ist derzeit die einzige IDE welche die Scopes nicht nur visualisiert, sondern auch die daraus resultierenden, getrennten Klassenpfäde während der Entwicklung aktiv unterhält und nutzt.
  - Dadurch wird absolut zuverlässig vermieden, dass z.B. in einer produktiven Klasse (im Pfad `src/main/java`) eine Referenz auf eine Klasse aus einer Dependency vom **test**-Scope erstellt werden kann!
- Qualitätssicherung des Design's einer Software.



# **Transitive Dependencies**

# Transitive Dependencies und Konflikt-Resolving

- Ein sehr nützliches Feature des Maven-DM ist die automatische Auflösung von so genannten **transitiven** Dependencies:

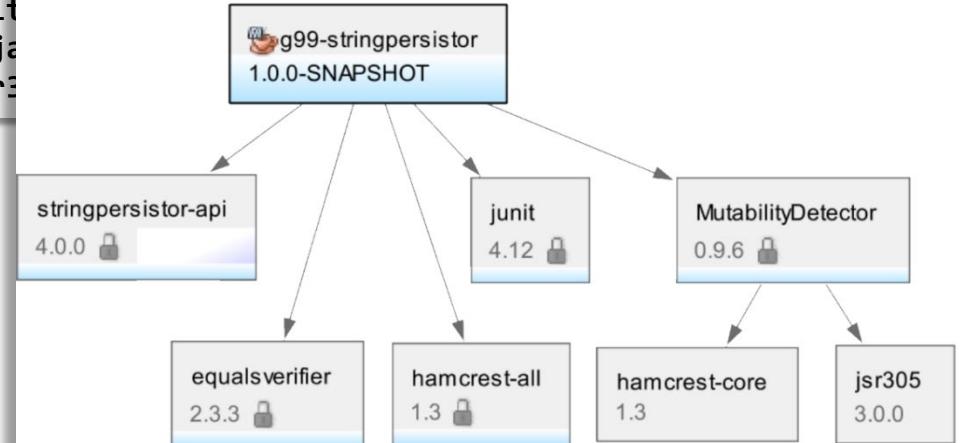


- Auflösung der Dependencies:
  - Modul **A** ist von Modul **B**, und dieses von Modul **C** abhängig.
  - Modul **A** ist somit **transitiv** auch von Modul **C** abhängig.
  - Bei Kompilation von A wird Modul **C** somit auch miteinbezogen.
- Durch direkte oder transitive Abhängigkeiten können auch (Versions-)Konflikte oder Zyklen auftreten!
  - Maven erkennt solche Konflikte und meldet diese.
  - Einfachere Versionskonflikte werden (durch wählbare Strategien) automatisch aufgelöst.

# Dependency-Graph

- Da Maven die Dependencies detailliert auswerten muss, werden diese als Graph ausgewertet → Graphentheorie im Modul AD.
  - Suche von Zyklen (die nicht erlaubt sind).
  - Auflösung von (Versions-)Konflikten z.B. über kürzesten Pfad.
- Der Graph steht sowohl über CLI als auch grafisch zur Verfügung:

```
[INFO] --- maven-dependency-plugin:3.0.1:tree (default-cli) @ g99-stringpersistor ---
[INFO] ch.hslu.vsk17hs.g99:g99-stringpersistor:jar:1.0.0-SNAPSHOT
[INFO] +- ch.hslu.vsk:stringpersistor-api:jar:4.0.0:compile
[INFO] +- junit:junit:jar:4.12:test
[INFO] +- org.hamcrest:hamcrest-all:jar:1.3:test
[INFO] +- nl.jqno.equalsverifier>equalsverifier:jar:2.3.3:test
[INFO] \- org.mutabilitydetector:Mutabilit
[INFO]     +- org.hamcrest:hamcrest-core:ja
[INFO]     \- com.google.code.findbugs:jsr3
```



# **Versionierung und Snapshots**

# Versionierung

- Grundsätzlich sind alle Dependencies versioniert.
- Der Einsatz von «Semantic Versioning» (<http://semver.org>) wird empfohlen und von vielen Plugins und Dependencies auch eingehalten.
- Auf Basis dieser Semantik kann der Dependency Resolver diverse Automatisierungen und Vereinfachungen anbieten:
  - Erkennen von neueren Versionen.
  - Automatische Verwendung des neusten Bugfixes.
  - Angabe von Versionsbereichen welche Kompatibel sind etc.
- Gute Repositories sind so konfiguriert, dass eine einmal deployte Version **nicht** mehr überschrieben werden kann!
  - Das garantiert wirklich nachvollziehbare Buildprozesse!

# Semantic Versioning ([semver.org](https://semver.org))

- **Major**-Release (**X**.x.x): Veränderungen in der API, in der fachlichen Funktion und/oder in der Konfiguration, welche zu früheren Versionen nicht kompatibel sind.
  - In den meisten Fällen sind Anpassungen notwendig.
- **Minor**-Release (x.**X**.x): Erweiterungen in der API, der fachlichen Funktion oder der Konfiguration, welche aber vollständig Rückwärtskompatibel sind.
  - Ohne Nutzung der Neuerungen keine Anpassungen notwendig.
- **Bugfix/Maintenance**-Release (x.x.**X**): Reine Korrekturen oder Änderungen in der Implementation, voll rückwärtskompatibel, keinerlei neue Funktionen, keine veränderte Funktionen.
  - Direkter, sofortiger Einsatz möglich bzw. notwendig (Bugfix).

# Versionierung mit Snapshots

- In einer dynamischen Entwicklungsphase sind fixe Versionen aber eher hinderlich, die Versionierung würde sonst (für jede kleinste Änderung) förmlich «galopieren» müssen.
  - Es würde eine Unmenge von (unnützen) Versionen produziert, welche später auch nie mehr benötigt werden.
- Darum wurde das **Snapshot**-Konzept integriert:  
Sobald man einer Version den Appendix **-SNAPSHOT** trägt, gilt diese als «erneuerbar» und (noch) **nicht** stabil, sondern in Entwicklung.
  - Sie wird bei **jedem** Build **immer wieder** vom Repository aufgelöst und aktualisiert.
  - Im Repository sind Snapshots mit einem Timestamp versehen.
- Beispiel: **6.0.3-SNAPSHOT**
  - Die noch nicht stabil veröffentlichte, zukünftige Version **6.0.3**

# **«Managed Dependencies» in Multimodul-Projekten**

# Multimodul-Projekt

- Bei Projekten die aus mehreren Submodulen bestehen, können mehrere Submodule von der gleichen Dependency abhängig sein.
  - Beispiel: Jedes Submodul verwendet z.B. Log4J oder JUnit.
- Es macht sehr viel Sinn (bzw. es ist technisch sogar notwendig) dass in jedem (Sub-)Modul die **selbe** Version verwendet wird.
  - Dependencies (GroupID, ArtifactID, Version und Scope) werden aber in jedem Modul **redundant** definiert → **Schlecht!**
- Lösung: Im übergeordneten (Master-)POM kann über das Element **dependencyManagement** eine Liste von Dependencies inkl. Version und Scopes als «Baseline» oder «Valid Version Set» vordefiniert werden.
  - Submodule müssen dann nur noch Group- und ArtifactId angeben. Rest wird vom Parent-POM einheitlich vererbt.

# Managed Dependencies - Beispiel

- Definition im (Parent-)POM:

```
<dependencyManagement>
  <dependencies>
    <dependency>
      <groupId>org.apache.logging.log4j</groupId>
      <artifactId>log4j-api</artifactId>
      <version>2.19.1</version>
      <scope>compile</scope>
    </dependency>
  </dependencies>
  ...
</dependencyManagement>
```

- Nutzung in einem Dependencies-Element, im (Sub-)POM:

```
<dependencies>
  <dependency>
    <groupId>org.apache.logging.log4j</groupId>
    <artifactId>log4j-api</artifactId>
    // Version und Scope entfällt, weil von Parent geerbt!
  </dependency>
</dependencies>
```

# Managed Dependencies – Diskussion

- Diese Technik kann auch in einem Single-Modul Projekt angewandt werden, wobei der Nutzen dort etwas weniger offensichtlich ist.
- Tatsächlich wurde das schon im «**oop\_maven\_template**» aus den Modulen OOP und AD so praktiziert.
  - ➔ Vereinfacht eine spätere Migration auf ein Mehrmodul-Projekt.
  - ➔ Versionen können quasi «vordefiniert» werden.
- Eine alternative Technik ist die Verwendung einer so genannten «**bill of material**»-Abhängigkeit (BOM).
  - Damit können verschiedene Versionen in einer «virtuellen» Release-Unit quasi als «Baseline» referenziert werden.
  - Der Lieferant bestimmt welche zueinander passenden Versionen zum Einsatz kommen sollen, wenn wir sie denn benötigen.
  - Diese BOM wird selber als (versionierte) Abhängigkeit definiert.

# **Deployment**

# Deployment von Java Modulen

- Die häufigste Art von Deployment sind JAR-Dateien.
- Beispiel für Artefakt `ch.hslu.vsk:stringpersistor-api:6.0.2:`
  - POM (Metainformationen): `stringpersistor-api-6.0.2.pom`
  - JAR (Binary): `stringpersistor-api-6.0.2.jar`
  - JavaDoc: `stringpersistor-api-6.0.2-javadoc.jar`
  - Source (bei OSS): `stringpersistor-api-6.0.2-sources.jar`
  - Zu allen Artefakten werden noch Hashes produziert.
- Dass die Quellen und die JavaDoc auch als JAR (eigentlich ZIP) geliefert werden ist Konvention und kann Unwissende verwirren.
  - Letztlich aber egal, die Einheitlichkeit ist wichtiger!
- Vorteil, z.B. für Entwicklungsumgebungen:  
Es ist implizit klar, wo die Dokumentation und ggf. der Source für ein bestimmtes JAR gefunden wird → Selbstkonfiguration.

# Deployment in Maven Repositories

- Das Deployment in öffentliche Repositories (z.B. Maven Central) wird sehr restriktiv gehandhabt, weil danach nichts mehr verändert werden darf (auch kein Löschen!)
  - Stabilität von Builds muss gewahrt bleiben!
- In Firmen kann man durchaus (z.B. veraltete) Artefakte auch mal löschen, aber auch das muss sehr gewissenhaft erfolgen.
  - «Repository-Pflege» wird häufig unterschätzt oder vergessen!
- Sehr oft dürfen Entwickler\*innen (zurecht) nicht direkt deployen!
- Man bedient sich stattdessen eines nachvollziehbaren, automatisierten und verifizierbaren Release-Prozesses, welcher von einem →Buildserver ausgeführt wird.
- Auch bei uns in VSK wird das so gehandhabt!

# **Demo?**

## Weiterführende Informationen

- Eine gute Übersicht zum Thema Dependency Management erhalten Sie in der Dokumentation von Apache Maven:

<https://maven.apache.org/guides/introduction/introduction-to-dependency-mechanism.html>

# Fragen

Verteilte Systeme und Komponenten

# **Buildserver**

## **Technologien und Funktionsweise**

Roland Gisler

# Inhalt

- Was ist ein Buildserver?
- Beispiele von Buildserver-Produkten
- Einsatzszenarien
- VSK: Buildserver im Logger-Projekt

# Lernziele

- Sie wissen was ein Buildserver ist.
- Sie kennen die Vorteile beim Einsatzes eines Buildservers.
- Sie kennen beispielhafte Produkte von Buildservern und können diese als Anwender\*in nutzen.
- Sie kennen das Potential von automatisierten CI/CD-Prozessen.

# **Bob the Builder?**



<http://www.bobthebuilder.com/de/>

# Was ist ein Buildserver?

- Serversoftware, welche einen bereits automatisierten Build eines Softwareprojektes ausführt, und die Resultate allen Entwickler\*innen im Team zur Verfügung stellt.
- Auslösung eines Builds aufgrund verschiedener Events:
  - Automatisch durch Änderungen im Versionskontrollsystem.
  - Automatisch durch Zeitsteuerung.
  - Manuell durch Anwender\*in.
- Positive Effekte
  - Entlastung von Entwickler\*innen von repetitiven Aufgaben.
  - Häufigere Verifikation (Buildprozess, Tests, Deploying etc.).
  - Statistische Information über Entwicklungsprozess.
  - Offensive (automatische) Information über den Zustand der Projekte.

# **Beispiele von Buildservern**

# Beispiele von Buildservern

The image displays three screenshots of build server interfaces:

- Jenkins:** Shows a dashboard with various projects and their status. A specific project, "HSLU Informatik - Modul VSK", is highlighted. It shows a table of builds across ten teams, with columns for Name, Last Version, # QG, # CS, # PMD, # SB, Zellenabdeckung, Letzte Status, Letzte Dauer, and Cause. Most builds show 100% coverage and a green status.
- TeamCity:** Shows a "Continuous Build" overview. It lists multiple builds with columns for Artifacts, Changes, Started, Duration, Agent, and Tags. All builds are marked as successful (green) and have a duration of 14s.
- GitLab CI:** Shows a pipeline named "g06-logger" under the project "VSK-22FS01 / g06". It displays a list of stages: Project information, Repository, Issues, Merge requests, CI/CD, Pipelines, Jobs, Schedules, Test Cases, Security & Compliance, Deployments, Monitor, Infrastructure, Packages & Registries, Analytics, Wiki, Snippets, and Settings. The Pipelines section shows several runs, with one recent run failing (red) and another succeeded (green).

# **Buildserver - Ausgewählte Produkte (Beispiele)**

- Open Source
    - Jenkins/Hudson – einer der aktuell populärsten Buildserver.
    - Go – Moderne Ansätze, vereinfacht, Pipelines, CD (status?).
    - CruiseControl – einer der ältesten Buildserver (retired)
    - Continuum – Ursprüngliche speziell für Maven-Projekte (retired).
  - Kommerzielle Server (häufig auch freie Community-Editions)
    - TeamCity – sehr funktionaler und gut skalierender Buildserver.
    - Bamboo – eng verknüpft mit JIRA (Issue-Tracking).
- Ideal für den Einsatz in Unternehmen (on-site/closed source).

# **Buildserver - Cloud-Dienste (Beispiele)**

- Hostingplattformen zur Projekt- und Codeverwaltung haben sich in der Cloud ja bereits schon lange etabliert.
  - Beispiele: Sourceforge, GitHub, BitBucket, GitLab etc.
- Dienste für Buildserver (CI) in der Cloud zogen nach, z.B.:
  - <https://travis-ci.com/>
  - <http://www.cloudbees.com/>
  - Nun häufig direkt in Hostingplattformen integrierte Services.
- Für Open Source Projekte häufig gratis!
  - Meist (sehr) gute, transparente Integration.
  - Projekte müssen aber oft «public» verfügbar sein (OSS).
- Derzeit findet eine Konsolidierung statt. Einige Dienste «sterben» gerade oder werden übernommen und integriert.

# Konfiguration von Buildservern

- Es gibt im wesentlichen **zwei** komplett **gegensätzliche** Ansätze!
- **Variante 1:** (typisch für «on-site» Produkte)
  - Konfiguration ist vom Projekt vollständig getrennt.
  - Wird (meist) interaktiv direkt auf dem Buildserver vorgenommen.
  - Infrastruktur (Buildagents etc.) wird «geschützt».
  - ➔ Es resultiert eine Art «Gewaltentrennung».
- **Variante 2:** (eher für Cloud- und Hosting-Plattform)
  - Konfiguration ist direkt im Projekt abgelegt (z.B. `.yml`-Datei).
  - Wird direkt durch Entwickler\*innen konfiguriert.
  - Weniger Restriktiv, sehr häufig mit Docker (ad-hoc Agents).
  - ➔ Mehr Freiheit und Eigenverantwortung.
- Variante **1** eher in Organisationen, Variante **2** eher bei OSS.
  - Individuelle Ausnahmen bestätigen die Regel...

# **Einsatz von Buildservern**

# Einsatz von Buildserver

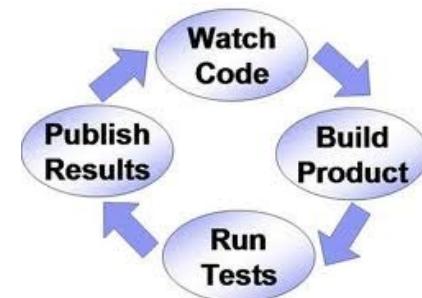
- Einsatz einer Buildservers **setzt** andere Technologien **voraus**:
  - **Automatisierte** Builds, z.B. mit Maven oder Gradle etc.
  - Einsatz eines Versionskontrollsystems, z.B. Git, Subversion etc.
- Man sollte auf eine saubere Aufgabentrennung zwischen den einzelnen Systemen und Technologien achten:
  - **Wann** wird ein Build ausgeführt: Buildserver / Anwender.
  - **Was** wird gebaut: Versionskontrollsystem.
  - **Wie** wird gebaut: Buildautomatisation.
  - **Wohin** gehen die Artefakte: Buildserver / Binary-Repo.
- Speziell das '**wie**' sollte **nie** im Buildserver umgesetzt, sondern immer durch den automatisierten Build abgedeckt werden.
  - Durch Buildtools wie z.B. Maven oder Gradle.

# Verschiedene Buildarten/-szenarien

- **Continuous** Builds: Automatisch bei einer Änderungen (Push/Merge-Request/Pull-Request) im Versionskontrollsystem.
  - Schnelle, möglichst kurze Builds.
  - Ziel: Schnelles Feedback für Entwickler\*innen.
- **Nightly** Builds: Automatisch nach Zeitsteuerung, typisch Nachts.
  - Eher umfangreiche, lange Builds.
  - Auch für zeitintensive Tests und Metriken geeignet.
  - Ziel: Am Morgen stehen umfassende Resultate zur Verfügung.
- **Release** Build: Manuell ausgelöst.
  - Build einer auslieferbaren Version, im VCS getagged.
  - Ziel: Reproduzierbarkeit gewährleisten.
  - Alternative: Build Pipeline.

# Integration und Verknüpfung

- Moderne Buildserver-Technologien zeichnen sich durch eine hohe Integrationsfähigkeit aus (analog zu Buildtools und IDE's).
  - Typisch über Plugin-Mechanismen realisiert.
- Integration von
  - verschiedenen Buildtools.
  - verschiedenen Versionskontrollsystmen.
  - verschiedenen Kommunikationstechnologien zur Notifikation.
  - verschiedene Visualisierungen / Plugins für IDE's.
  - etc.
- Verknüpfung mit
  - Issue-Tracking Systemen.
  - Code-Review Werkzeugen.
  - etc.



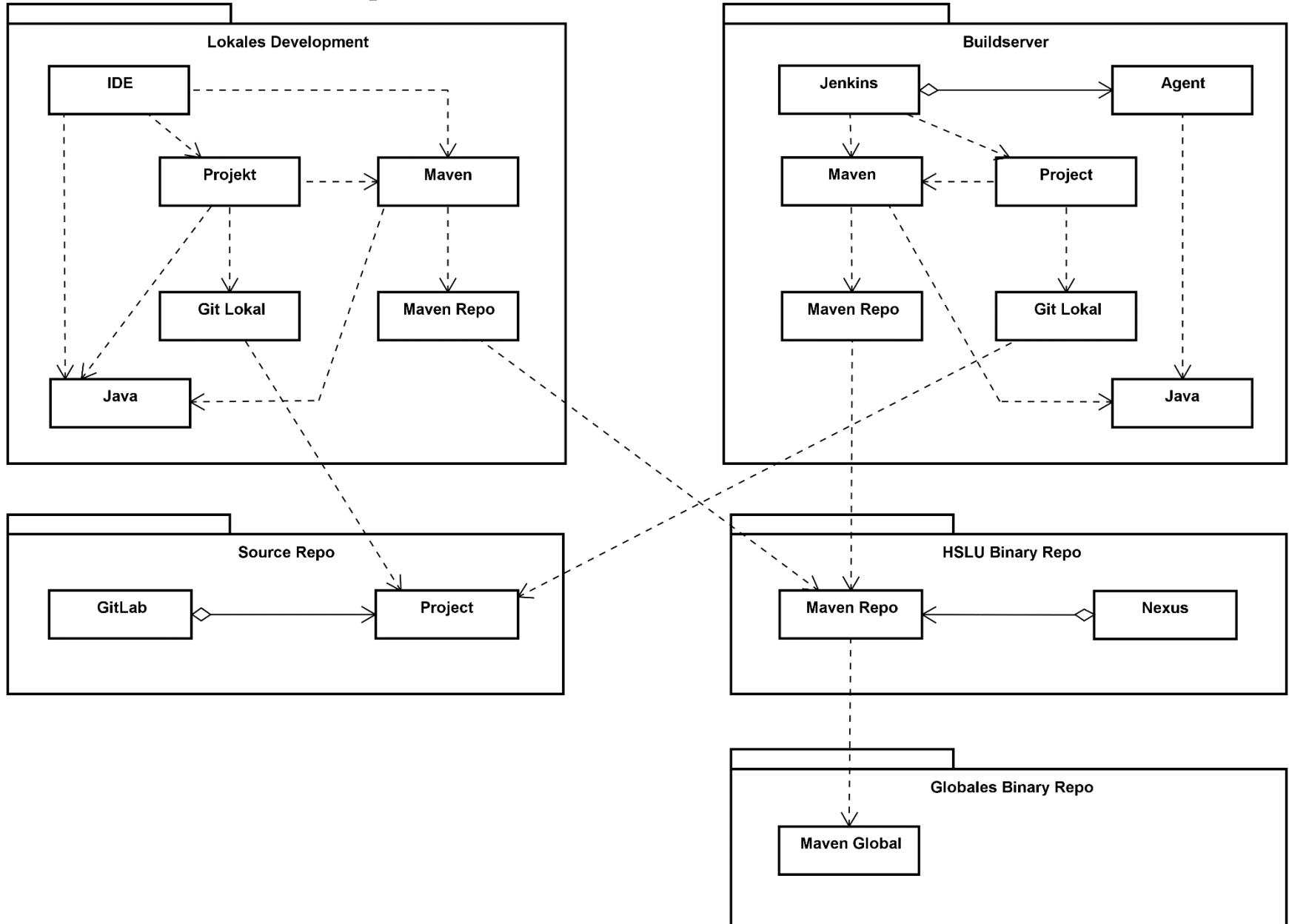
# **Logger-Projekt**

# Zugriff auf Buildserver

- Buildserver stehen **nur** im **internen** Netz der HSLU zur Verfügung
  - Zugriff von ausserhalb nur über **VPN** möglich.
- Buildserver VSK HS22: <https://jenkins-vsk.el.eee.intern/jenkins>
  - Login mit Enterpriselab-Login (analog GitLab)
  - **Achtung:** «**Neues**» HSLU-internes Root-Zertifikat (SSL), siehe:  
[https://wiki.enterpriselab.ch/el/public:help:certificate\\_import](https://wiki.enterpriselab.ch/el/public:help:certificate_import)  
<https://discuss.enterpriselab.ch/t/el-root-certificate-change/197>
- Konfiguriert für **Continuous Builds** sämtlicher Projekte
  - Automatischer Build nach Commit/Push auf Repository.
  - Build führt die Goals **deploy** und **site** aus.
  - Unüblich viel in einem Build, aber es geht ja relativ schnell... ☺
  - Besser wäre: Aufteilen in mehrere Steps, «fail fast»-Prinzip.
    - Höherer Konfigurationsaufwand ☹

# **Demo (Jenkins & GitLab)**

# Übersicht – Komplette Infrastruktur



# VSK Logger Projekt - Ziele

- Projekte auf dem Buildserver sollen möglichst «grün» sein.
  - Projekte müssen fehlerfrei gebaut werden können.
  - Jenkins gilt als «Master» für die Beurteilung!
- Ziel/Empfehlung: Alle Projekte sind **immer** fehlerfrei buildbar.
  - → Höchstes Ziel.
  - Sobald ein Build «rot» ist, wird kein neuer Code mehr committed, sondern zuerst der Build gefixt.
  - Gemeinsames Ziel (für das Team) ist es, allfällig rote Builds wieder «grün» zu machen.
- Das ist ein zentraler Schritt zur «Continuous Integration» (CI).
  - Abschliessender Input in SW07.

**Fragen?**

Verteilte Systeme und Komponenten

# Architekturbeschreibung

Martin Bättig

Letzte Aktualisierung: 12. Oktober 2022

FH Zentralschweiz

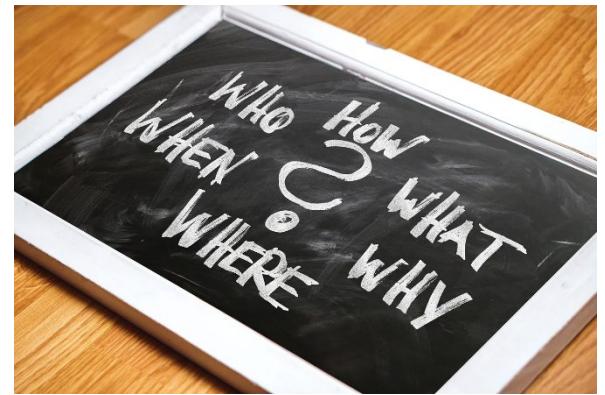


# Inhalt

- Architekturbeschreibung
- Vorlagen
- Sichten auf ein System

# Lernziele

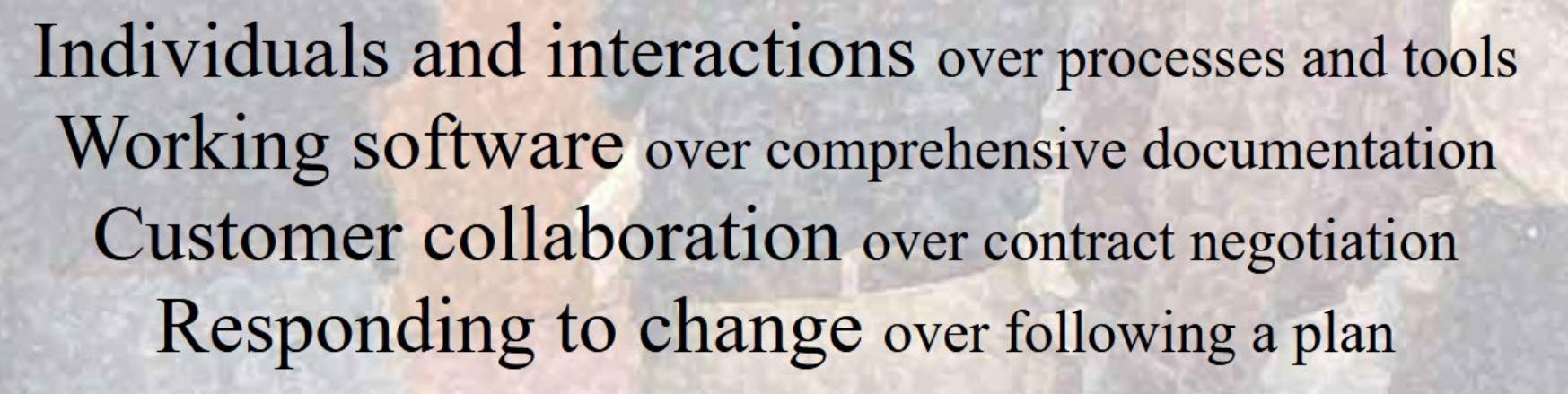
- Sie kennen das Konzept der Software-Komponenten und können Komponenten gemäss Spezifikation erstellen, dokumentieren, testen und überarbeiten.



# Architekturbeschreibung

Bildquelle: pixabay.com

# **Manifesto for Agile Software Development**



**Individuals and interactions** over processes and tools  
**Working software** over comprehensive documentation  
**Customer collaboration** over contract negotiation  
**Responding to change** over following a plan

**Quelle:** <http://agilemanifesto.org>

# Ziel der Architekturbeschreibung

**Beschreibt die Umsetzung der funktionalen und nicht-funktionalen Anforderungen, welche an ein System gestellt werden.**

Anforderungen abgeleitet aus:

- übergeordneten Anforderungsdokumenten (z.B. Lastenheft).
- übergeordneten Systemen (z.B. Schnittstellen der umgebenden Systeme).

Alternative Namen:

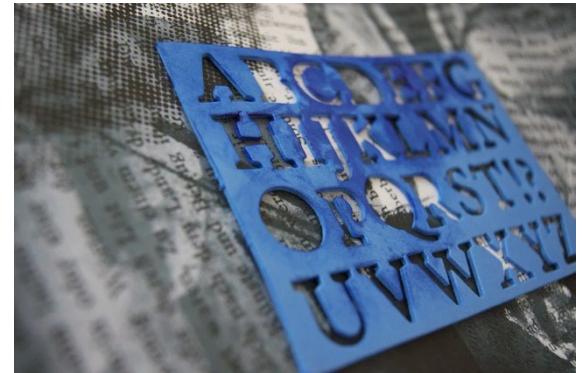
- Systemspezifikation
- Systembeschreibung

# Nutzen der Architekturbeschreibung

- Entwurf und Dekomposition der Architektur.
- Grobdesigns der Komponenten.
- Kommunikation der Informationen an die am Projekt beteiligten Akteure.

# Kompatibilität mit agilen Vorgehensmodellen

- Initiale Version (Vorprojekt / Initialisierungsphase / Sprint #0).
  - Bildet bisher erkannte Anforderungen ab.
- Anpassungen pro Sprint, sobald Anforderungen besser bekannt sind.
- Änderungen der Architekturbeschreibung in Sprint-Review kommunizieren.



# Vorlagen

Bildquelle: pixabay.com

# Vorlagen (Auswahl)

Auswahl an verschiedenen Vorlagen:



arc42

**Software  
Architecture**  
for Developers



SA4D/C4-Modell



Firmeneigene Vorlage

Warum Vorlagen verwenden?

- Gängiger Standard im deutschsprachigen Raum.
- Beantwortet zentrale Fragen:
  - **Was** sollen wir über unsere Architektur kommunizieren / dokumentieren?
  - **Wie** sollen wir kommunizieren / dokumentieren?

<b>1. Einführung und Ziele</b> 1.1 Aufgabenstellung 1.2 Qualitätsziele 1.3 Stakeholder	<b>7. Verteilungssicht</b> 7.1 Infrastruktur Ebene 1 7.2 Infrastruktur Ebene 2 ....
<b>2. Randbedingungen</b> 2.1 Technische Randbedingungen 2.2 Organisatorische Randbedingungen 2.3 Konventionen	<b>8. Querschnittliche Konzepte</b> 8.1 Fachliche Struktur und Modelle 8.2 Architektur- und Entwurfsmuster 8.3 Unter-der-Haube 8.4 User Experience ....
<b>3. Kontextabgrenzung</b> 3.1 Fachlicher Kontext 3.2 Technischer- oder Verteilungskontext	<b>9. Entwurfsentscheidungen</b> 9.1 Entwurfsentscheidung 1 9.2 Entwurfsentscheidung 2 ....
<b>4. Lösungsstrategie</b>	<b>10. Qualitätsanforderungen</b> 10.1 Qualitätsbaum 10.2 Qualitätsszenarien
<b>5. Bausteinsicht</b> 5.1 Ebene 1 5.2 Ebene 2 ....	<b>11. Risiken und technische Schulden</b>
<b>6. Laufzeitsicht</b> 6.1 Laufzeitszenario 1 6.2 Laufzeitszenario 2 ....	<b>12. Glossar</b>

Inhaltsverzeichnis, wobei blaue Kapitel den Kern des Dokuments bilden.

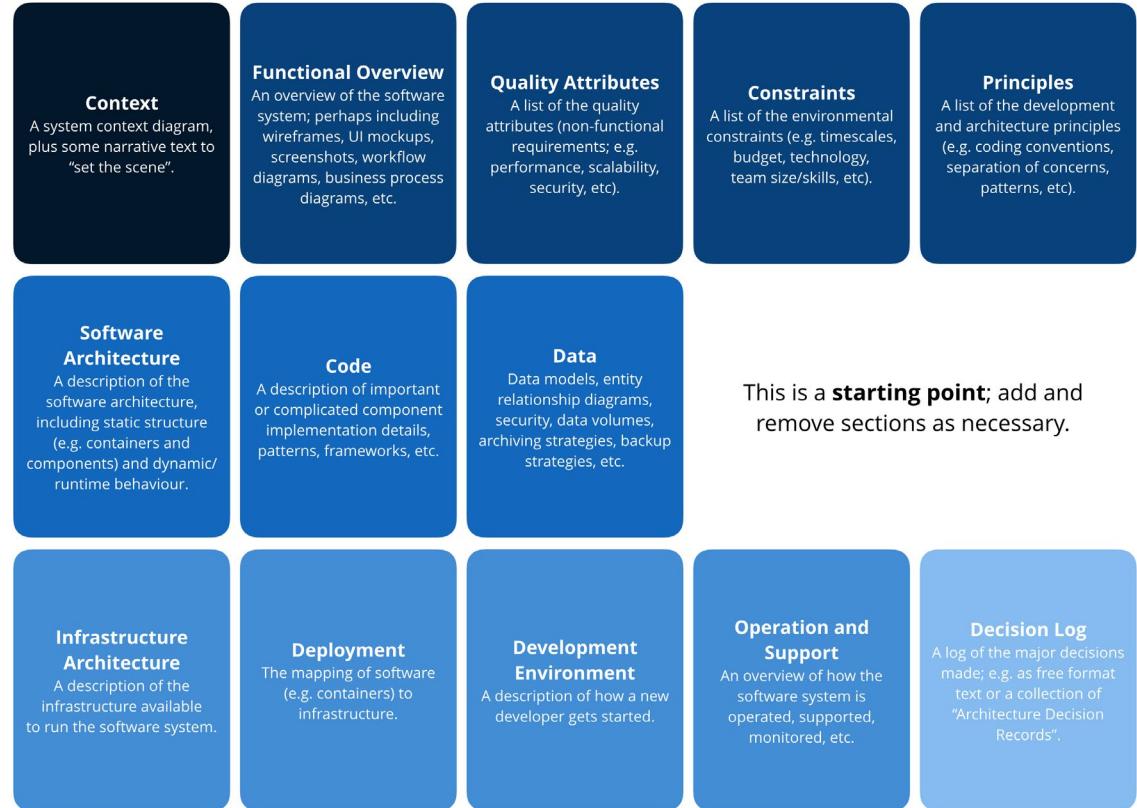
Quelle: <https://www.arc42.de>

## arc42 Beispiele

- [https://hsc.aim42.org/documentation/hsc\\_arc42.html](https://hsc.aim42.org/documentation/hsc_arc42.html)
  - Verbose example for the documentation of a Gradle plugin, created by Dr. Gernot Starke.
- <https://biking.michael-simons.eu/docs/index.html>
  - (English) A real world example for a bike activity tracker, created by Michael Simons.

# SA4D

- Motto: "Beschreibe was nicht offensichtlich aus dem Code erkennbar ist."
- Eher Produkt als Projekt-Dokumentation.
- Kombiniert mit dem C4-Architektur-visualisierungsmodell.

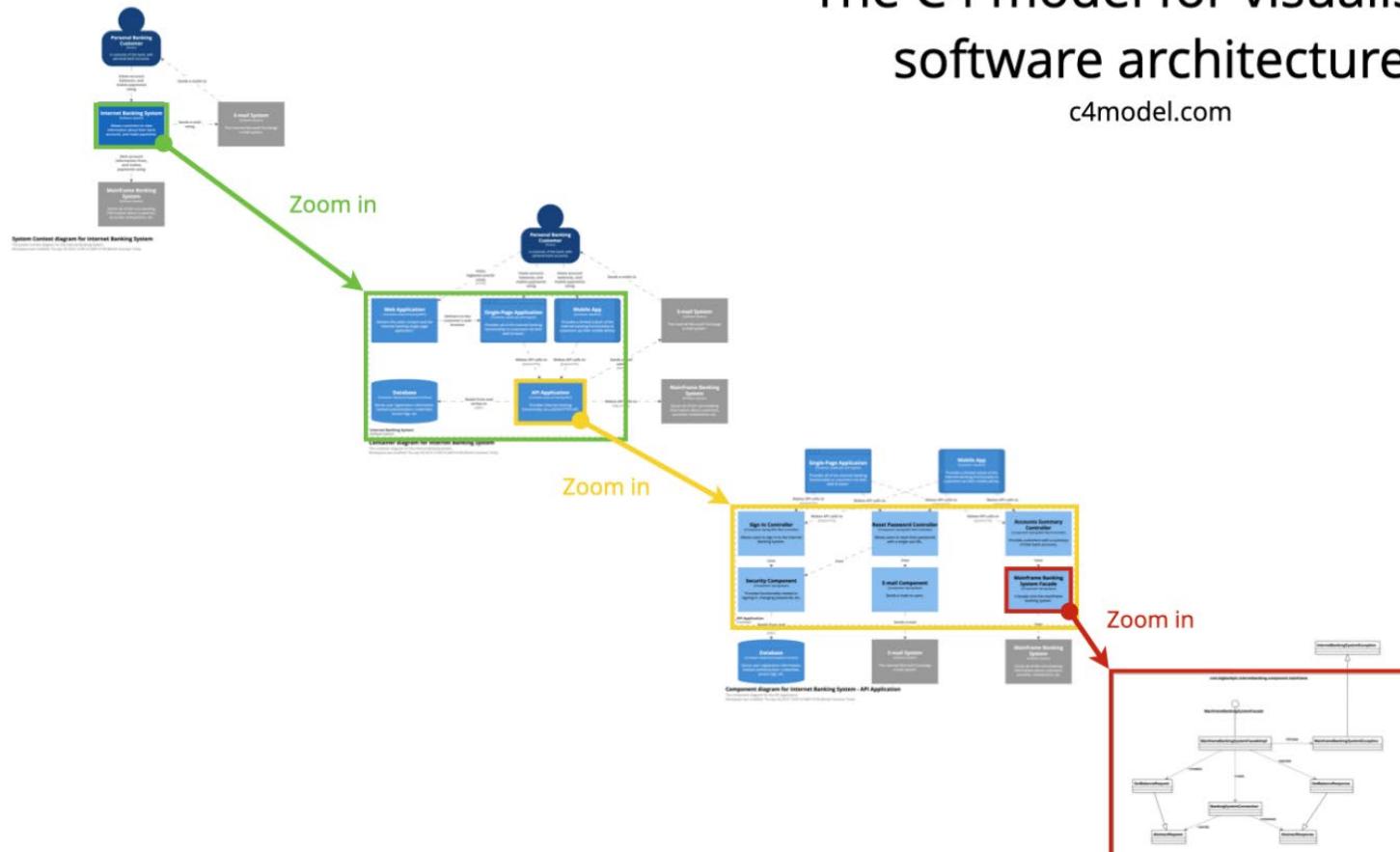


**Quelle:** Software Architecture for Developers Vol. 2, Simon Brown, Leanpub, 2021.

# C4 Modell

The C4 model for visualising  
software architecture

c4model.com



Level 1  
Context

Level 2  
Containers

Level 3  
Components

Level 4  
Code

# Firmeneigene Dokumente

- Viele grössere Firmen haben eigene Vorlagen.
- Auf Produkt, Vorgehen und/oder Branche zugeschnitten.



# Sichten auf ein System

# Sichten auf ein System

- Systemübersicht
- Schnittstellen
- Architektonische Sichten
- Systemumgebung
- Designentscheide

# Systemübersicht

- Welches Problem löst das System?
- Wie löst das System das Problem?
- Wer benutzt das System?
- Annahmen und Einschränkungen.

## 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

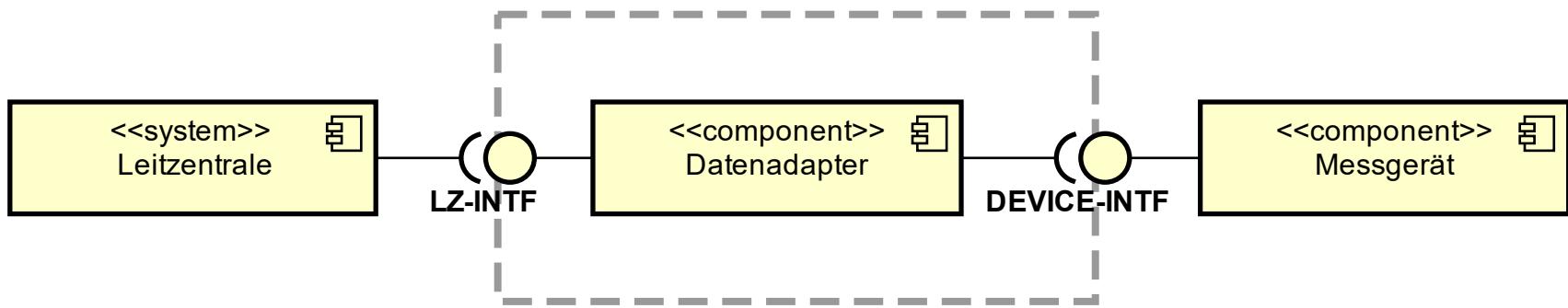
## 4. Lösungsstrategie

# Kontextabgrenzung

**Ziel:** Zeige das zu entwickelnde System in seinem Kontext (geschäftlich / technisch).

**3. Kontextabgrenzung**  
3.1 Fachlicher Kontext  
3.2 Technischer oder Verteilungskontext

**Beispiel für technischen Kontext: Kontextdiagramm eines Datenadapters**



**Wichtig:** Jedes Diagramm muss mit Text beschrieben werden!

# Schnittstellen

- Externe Schnittstellen.
  - Schnittstellen nach aussen.
  - Sowohl exportierte als auch importierte Schnittstellen.
- Benutzerschnittstellen.

## 3. Kontextabgrenzung

3.1 Fachlicher Kontext

3.2 Technischer- oder Verteilungskontext

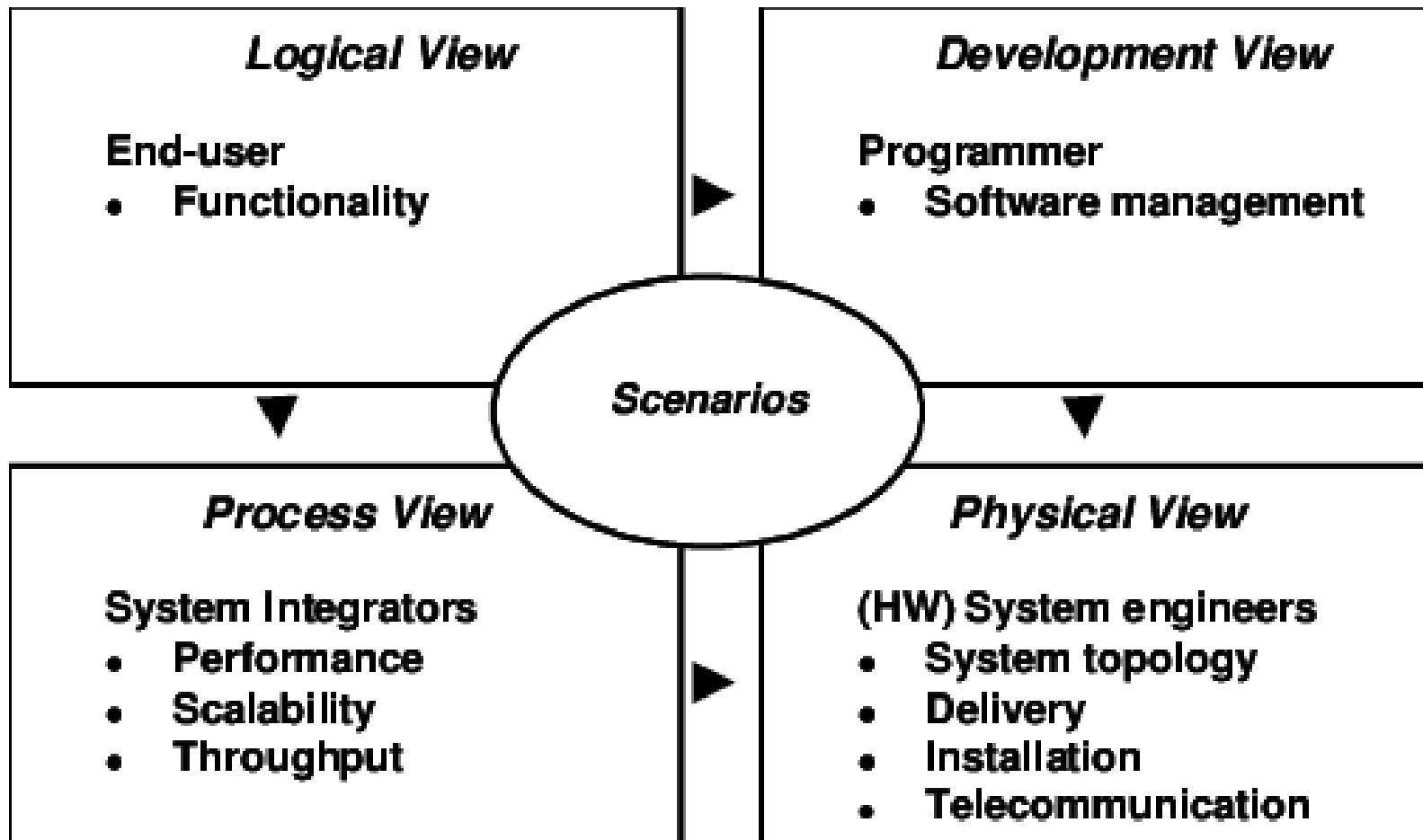
**Beispiel:** StringPersistor-Schnittstelle (Material auf ILIAS).

# Softwarearchitektur

- Unterschiedliche Sichten auf ein System aus unterschiedlichen Perspektiven **auf hohem Abstraktionsniveau.**
- Zuhilfenahme von Architekturbeschreibungsmodellen, z.B.
  - arc42 (Bausteinebenen).
  - 4 + 1-Sichten-Modell von Philippe Kruchten.
  - C4-Modell von Simon Brown (Fokus auf statische Sicht).

# 4+1-Sichten-Modell von Philipp Kruchten (1995)

Weit verbreitetes Modell:



# Statische Sicht und Laufzeitsicht (Logical View)

- Funktionalität des Systems auf unterschiedlichen Flughöhen zeigen z.B. arc42 (Bausteinebenen) oder C4-Modell.
- Pro System mehrere Ebenen (falls zutreffend):
  - Teilsystem / Services: z.B. Block- oder Komponentendiagramme.
  - Komponenten: z.B. Komponentendiagramme.
  - Code: i.d.R. Klassendiagramm oder Sequenzdiagramme.
  - Richtlinie 1: Kein Reverse-Engineering von Diagrammen aus dem Code!
  - Richtlinie 2: Nur interessante / hilfreiche Stellen dokumentieren.
- Dokumentation auf das Notwendigste reduzieren.
  - nur relevante Klassen und Attribute.

## 5. Bausteinsicht

5.1 Ebene 1

5.2 Ebene 2

....

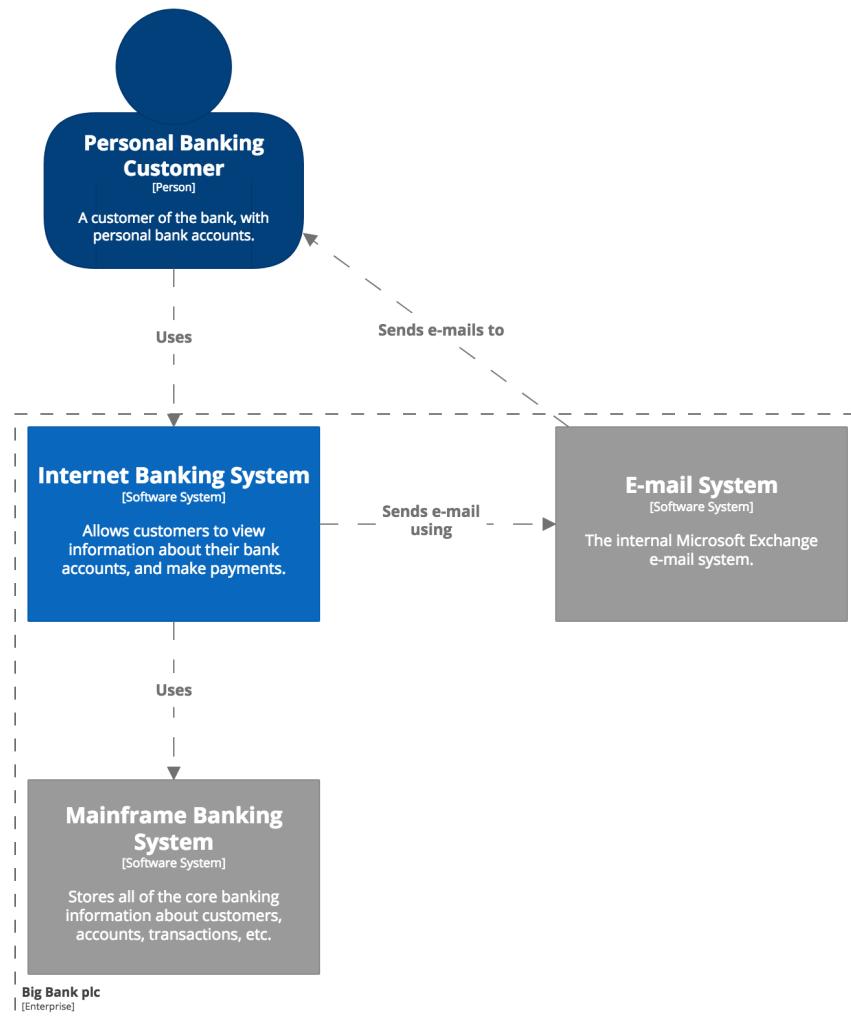
## 6. Laufzeitsicht

6.1 Laufzeitszenario 1

6.2 Laufzeitszenario 2

....

# Statische Sicht am Beispiel des C4-Modells (Ebene 1)



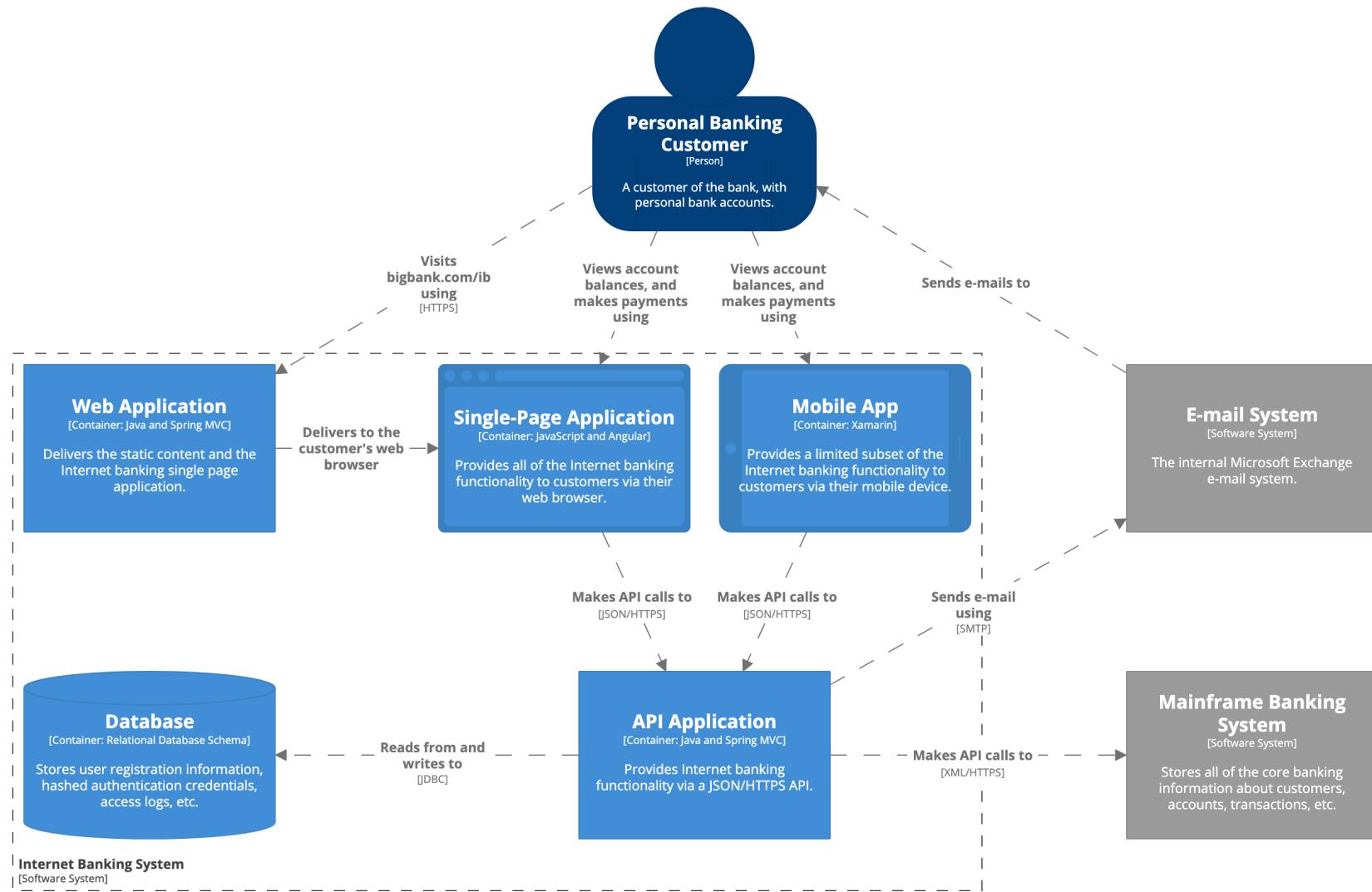
## System Context diagram for Internet Banking System

The system context diagram for the Internet Banking System.

Last modified: Wednesday 02 May 2018 13:41 BST

Quelle: Software Architecture for Developers Vol. 2, Simon Brown, Leanpub, 2021.

# Statische Sicht am Beispiel des C4-Modells (Ebene 2)



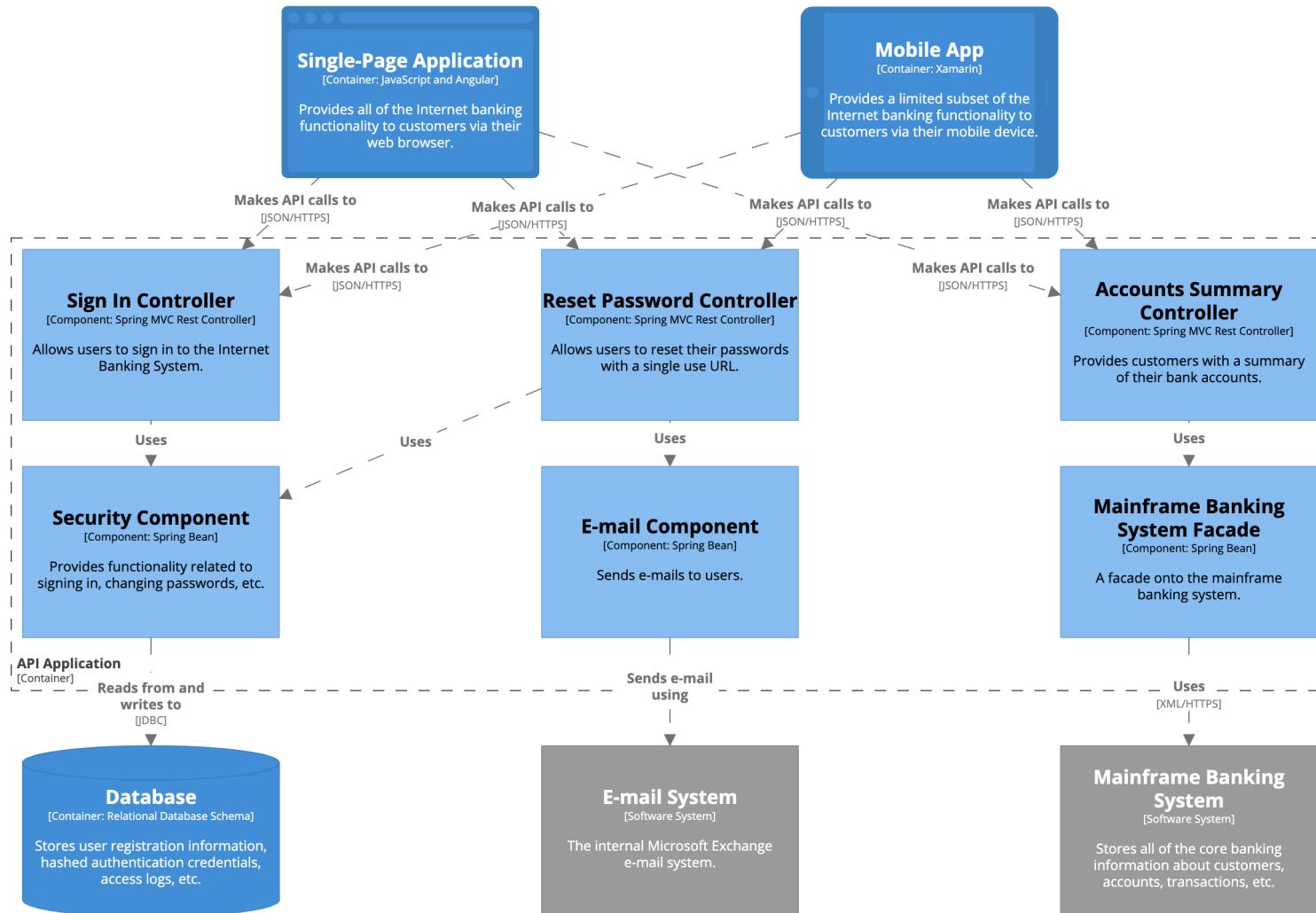
**Container diagram for Internet Banking System**

The container diagram for the Internet Banking System.

Workspace last modified: Thu Apr 04 2019 13:09:10 GMT+0100 (British Summer Time)

Quelle: Software Architecture for Developers Vol. 2, Simon Brown, Leanpub, 2021.

# Statische Sicht am Beispiel des C4-Modells (Ebene 3)



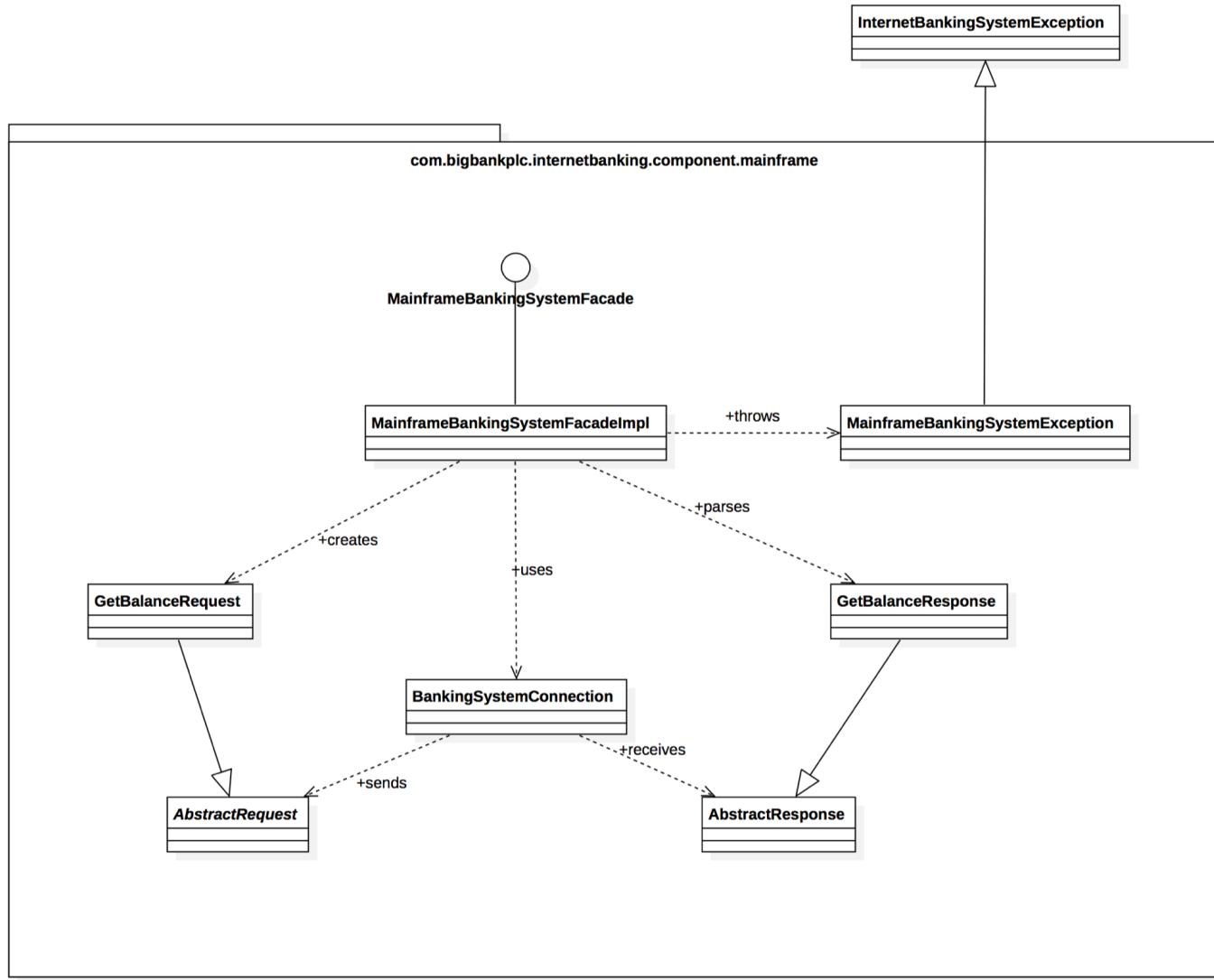
**Component diagram for Internet Banking System - API Application**

The component diagram for the API Application.

Workspace last modified: Thu Apr 04 2019 13:09:10 GMT+0100 (British Summer Time)

Quelle: Software Architecture for Developers Vol. 2, Simon Brown, Leanpub, 2021.

# Statische Sicht am Beispiel des C4-Modells (Ebene 4)



Quelle: Software Architecture for Developers Vol. 2, Simon Brown, Leanpub, 2021.

# Systemumgebung (Physical View)

- Sind mehrere physische Systeme involviert?  
Welche?
- Auf welchem System wird welche Komponente eingesetzt?  
Ergeben sich unterschiedliche Einsatzszenarien?
- Was sind die Anforderungen an das jeweilige System (Hardware,  
Betriebssystem, installierte Software, Netzwerkfreigaben / Firewall-Regeln,  
Netzwerkbandbreite, etc.).
- Wie werden die Komponenten in Betrieb genommen?
- Müssen die Komponenten konfiguriert werden? Wie?
- Praktisch: Angabe einer Beispielkonfiguration.

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

# Verarbeitungsansicht (Process View)

## Datenverarbeitung:

- Persistente Daten und deren Strukturierung (z.B. Benutzerkonten, Transaktionsdaten).
- Beziehungen zwischen den Daten (z.B. ER-Modell).
- Wie werden die Daten gespeichert? Datenbank (relational, NoSQL, eigenes Fileformat, via Webservice, in der Cloud, etc.).
- (Wie) wird die Konsistenz sichergestellt? Wird dies überprüft?
- u.s.w.

### 8. Querschnittliche Konzepte

- 8.1 Fachliche Struktur und Modelle
- 8.2 Architektur- und Entwurfsmuster
- 8.3 Unter-der-Haube
- 8.4 User Experience
- ....

## Wichtige Schnittstellen:

- Interne Schnittstellen zwischen Komponenten.

# Verarbeitungsansicht (Process View, forts.)

## Qualitätsanforderungen:

- Wie viele Daten pro Zeiteinheit muss das System bzw. einzelne Komponenten verarbeiten können?
- (Wie) werden die Daten wieder gelöscht?
- Maintainability: Wie werden die Daten gesichert (Backup)?  
Wie schnell kann ein System wiederhergestellt werden?
- u.s.w.

### 10. Qualitätsanforderungen

- 10.1 Qualitätsbaum
- 10.2 Qualitätsszenarien

# Designentscheide

- Was sind die wesentlichen Überlegungen, welche Sie beim Design des Systems gemacht haben?
- Sind bestimmte Randfälle absichtlich nicht unterstützt? Welche?
- Sollen bestimmte Techniken (nicht) verwendet werden? Welche?
- Bestimmte Programmierparadigmen, Patterns?
- Bestimmte Libraries, Frameworks, Laufzeitumgebungen?
- usw.

## 9. Entwurfsentscheidungen

9.1 Entwurfsentscheidung 1

9.2 Entwurfsentscheidung 2

....

# Zusammenfassung

- Grobdesign des Systems (verschiedene Sichten, Daten & Mengengerüste, sowie Designentscheide).
- Kommunikation mit beteiligten Akteuren (Kunden, Entwickler, Tester, Betrieb, Wartung).
- Schnittstellen: Extern, wichtige interne (Komponentengrenzen), Benutzerschnittstellen.
- Anforderungen an die Systemumgebung.

# Literatur

- arc42: better software architectures von Gernot Starke, Peter Hruschka, Ralf D. Müller.  
<https://arc42.org/>
- Software Architecture for Developers Vol. 2 von Simon Brown, Leanpub, 2021.

# Fragen?

Verteilte Systeme und Komponenten

# **Modularisierung und Schichtenarchitektur**

Martin Bättig



# Inhalt

- Modularisierung: Konzepte und Vorgehen
- Modularisierung mittels Schichten und Schichtenarchitektur

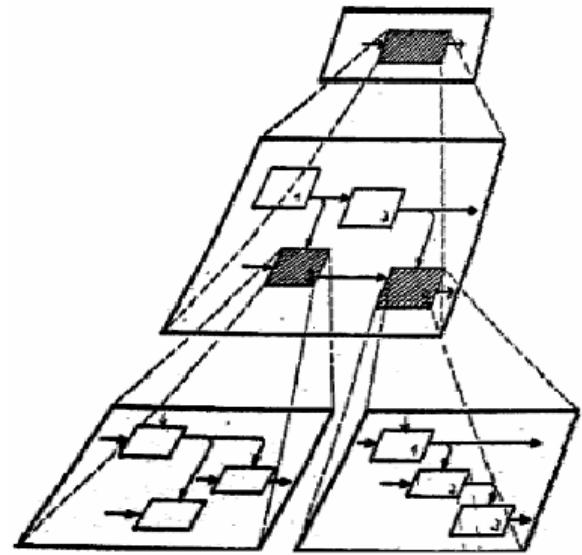
# Lernziele

- Sie verstehen das Konzept der Software-Komponenten und kennen die Kriterien zur Modularisierung.
- Sie verstehen das Schichtenkonzept und wissen wie dieses Konzept bei der Schichtenarchitektur angewandt wird.

# **Modularisierung: Konzepte und Vorgehen**

# Modul: Begriff

- In sich abgeschlossener Teil des gesamten Programm-Codes, bestehend aus einer Folge von Verarbeitungsschritten und Datenstrukturen.
- Die Anwendung des Modulkonzepts im Software Engineering wurde bereits 1972 von David Parnas publiziert.



# Big Ball of Mud

- Am Beispiel der Klassenbeziehungen einer realen Webapplikation (Java).



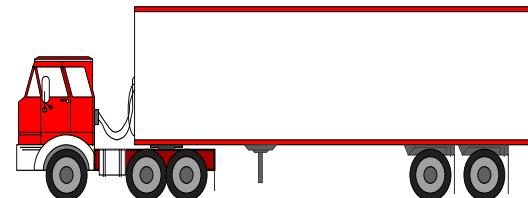
**Quelle:** Handbuch moderner Softwarearchitektur, Richards und Ford

# Kopplung und Kohäsion: Konzept



hohe Kopplung

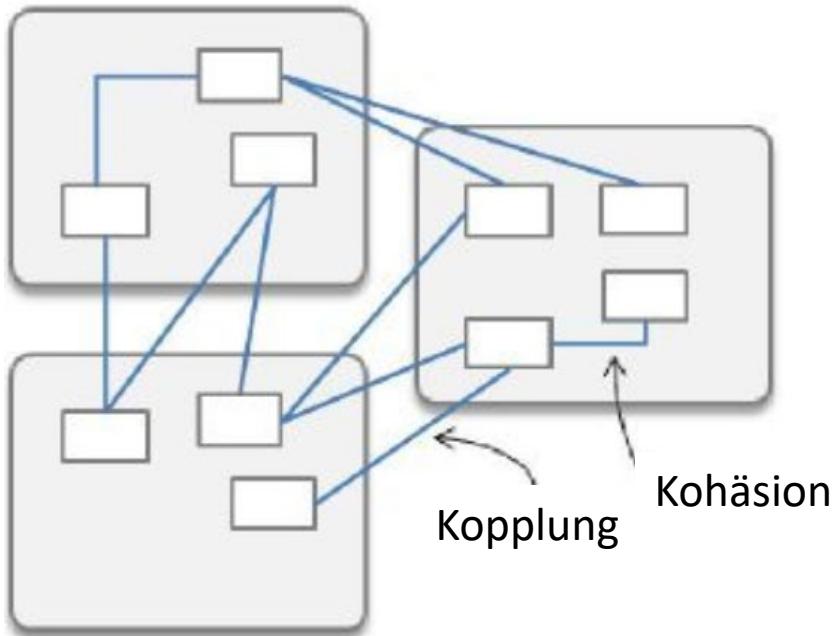
- "Module":
- Frachtraum
  - Führerkabine



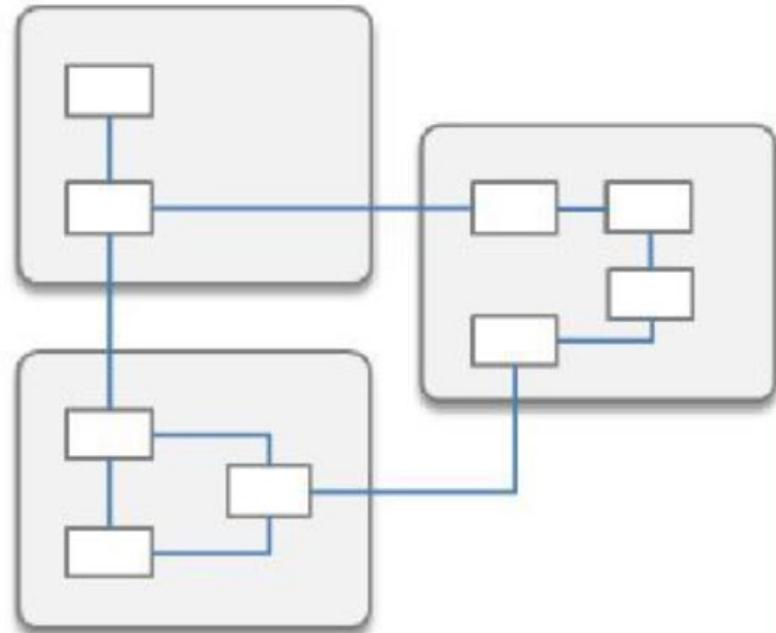
geringe Kopplung

- **Kopplung (Coupling):**  
Ausmass der Kommunikation zwischen Modulen.  
- Unabhängigkeit der einzelnen Module.  
**=> Minimiere die Kopplung!**
- **Kohäsion (Cohesion):**  
Ausmass der Kommunikation innerhalb eines Moduls.  
- interner Zusammenhalt innerhalb eines Moduls.  
**=> Maximiere die Kohäsion!**

# Kopplung und Kohäsion: Beispiel (Abstrakt)



- hohe Kopplung
- geringe Kohäsion

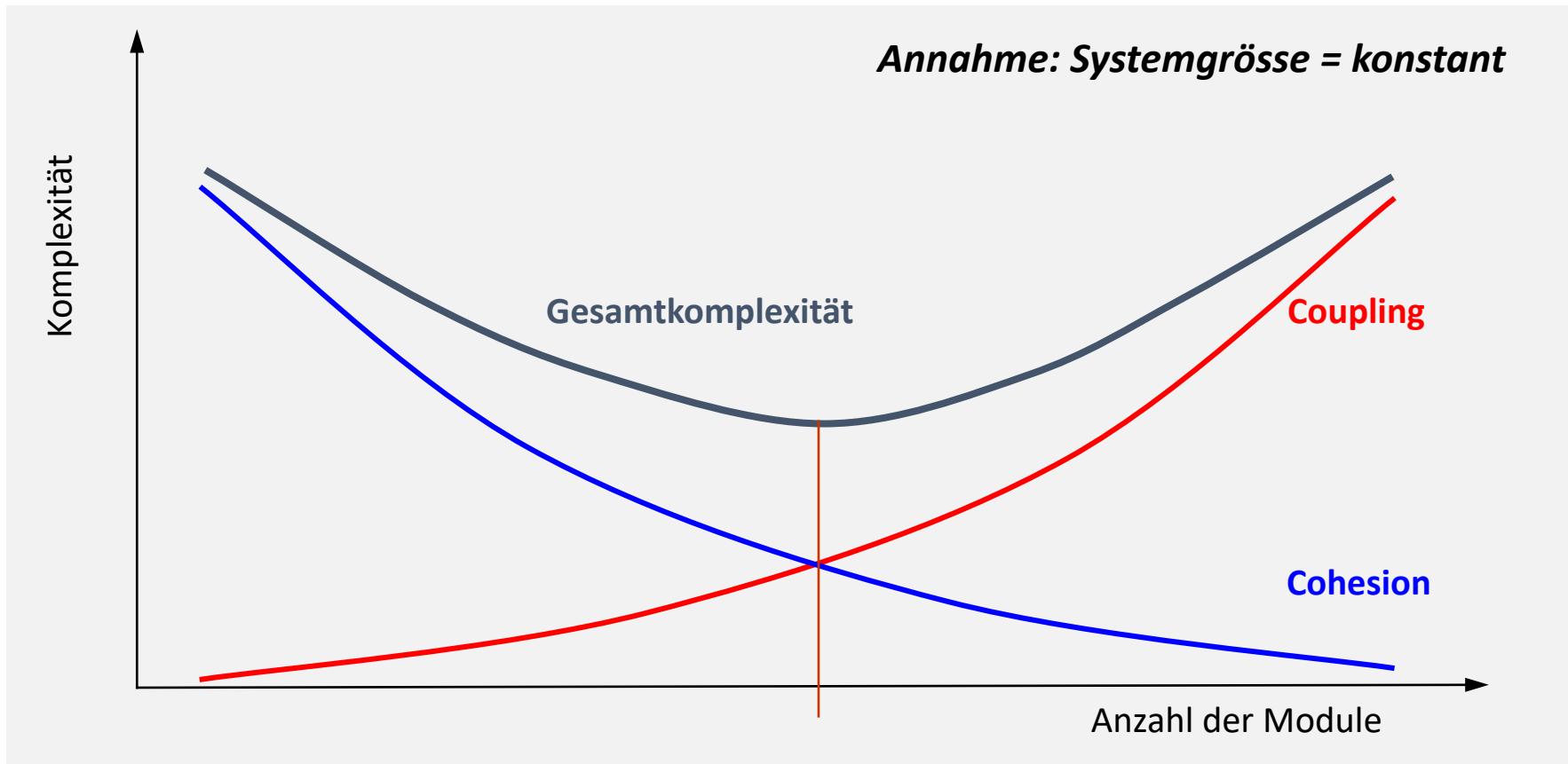


- geringe Kopplung
- hohe Kohäsion

**Quelle:** Modularisierung mit Java 9: Grundlagen und Techniken für langlebige Softwarearchitekturen von Guido Oelmann

# Umgang mit Kopplung und Kohäsion

- Kohäsion maximieren und gleichzeitig Kopplung minimieren ist nicht immer möglich.
- Optimierungsaufgabe!



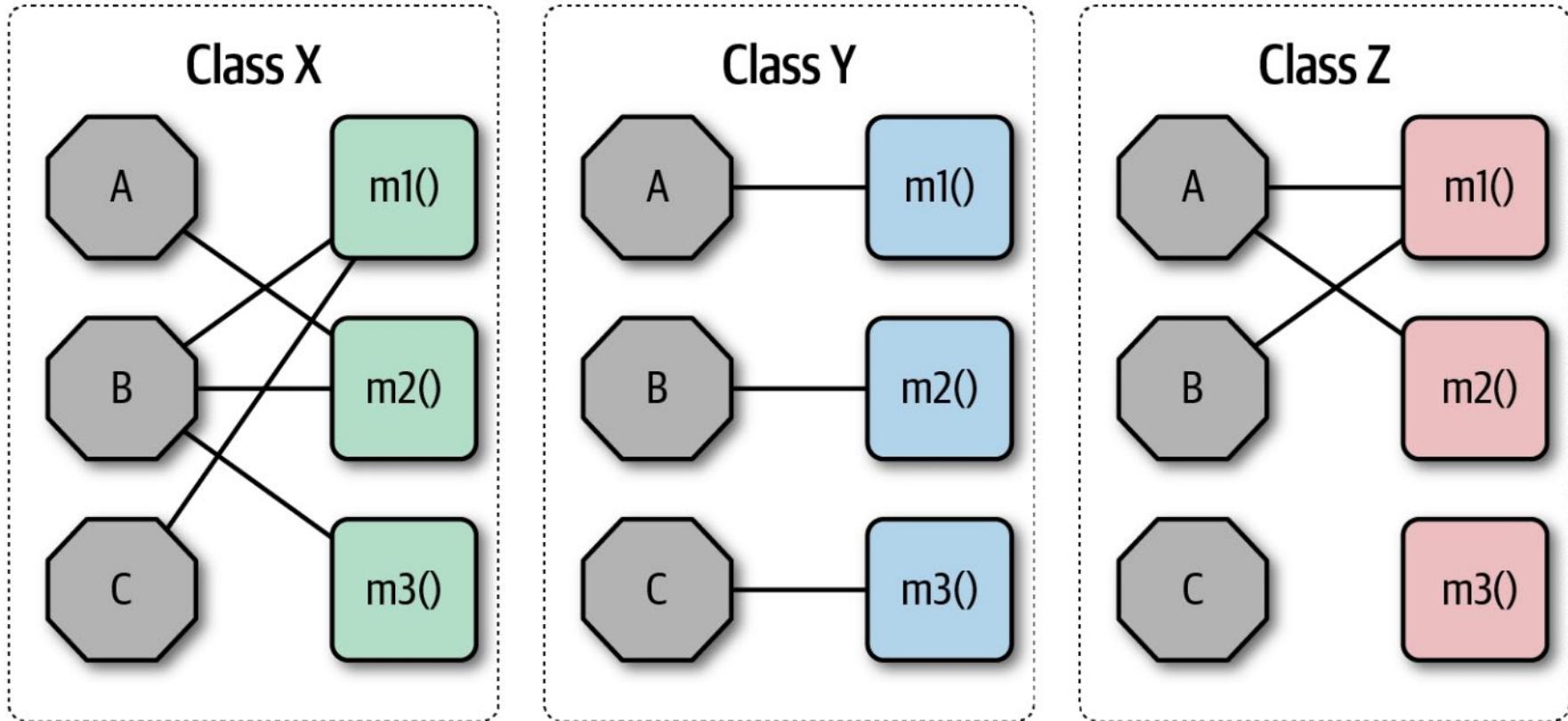
# Arten der Kohäsion

- **Funktionale Kohäsion:** Alle Teile haben eine ineinandergreifende Beziehung zu einander.
- **Sequentielle Kohäsion:** Input/Output-Beziehungen.
- **Kommunikatorische Kohäsion:** Kommunikationsketten bzgl. Informationsverarbeitung.
- **Prozedurale Kohäsion:** Ausführung in bestimmter Reihenfolge nötig.
- **Temporale Kohäsion:** Zeitbezogene Abhängigkeit, z.B. beim Startup.
- **Logische Kohäsion:** Logische, aber nicht funktionale Beziehung.
- **Zufällige Kohäsion:** Einheiten sind zufällig im selben Modul.



# Messung der Kohäsion

- LCOM (Lack of Cohesion in Methods): Summe der nicht gemeinsam genutzten Methodensätze, welche nicht auf geteilte Felder zugreifen.



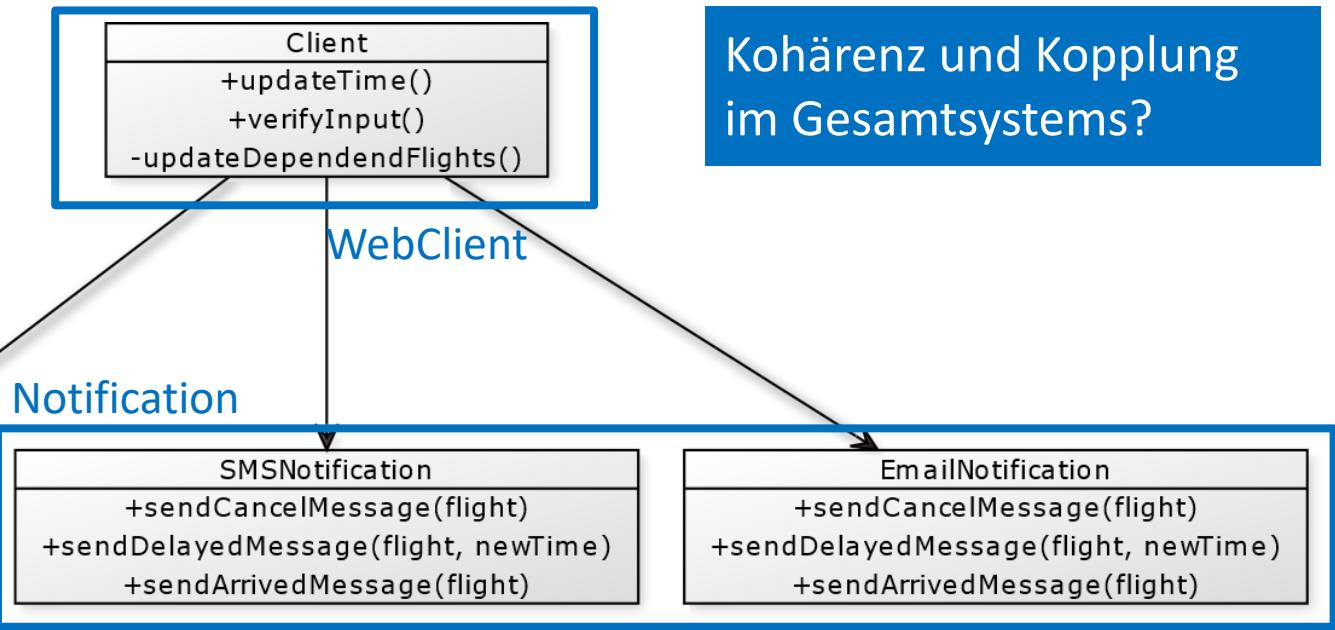
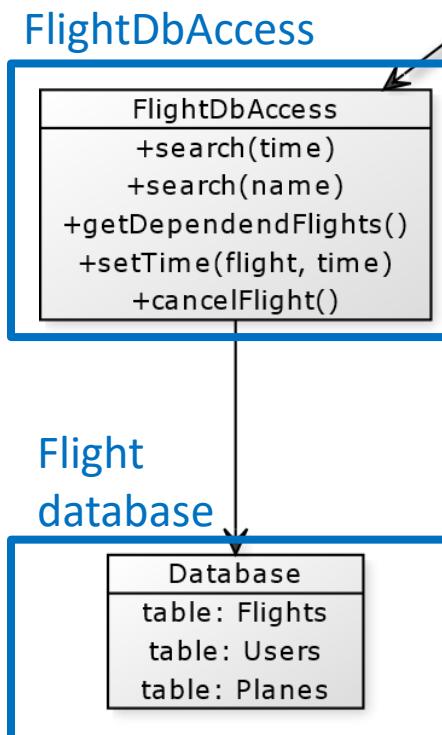
**Quelle:** Handbuch moderner Softwarearchitektur, Richards und Ford

# Arten der Kopplung

- **Laufzeitumgebung, Ausführungsort:** Module müssen in der selben Laufzeitumgebung oder auf dem selben System ausgeführt werden.
- **Technologie:** Gekoppelte Module müssen (teilweise) dieselben Technologien verwenden.
- **Zeit:** Module müssen zur selben Zeit aktiv sein.
- **Daten und Formate:** Module müssen die selben Datenformate parsen und verstehen (z.B. Datum oder Headers).

# Übung: Anpassung von Abflugzeiten.

System aus vier Teilsystemen (blau umrandet):



Kohärenz und Kopplung im Gesamtsystems?

## Grober Ablauf:

- Mitarbeiter sucht verspäteten Flug via Client und gibt die neue Abflugzeit ein.
- Client sendet dann die neue Zeit an die FlightDbAccess.
- Anschliessend ruft Client Anschlussflüge bei FlightDbAccess ab. Falls nicht genügend Zeit zum Umsteigen bleibt, wird der Client auch für die Anschlussflüge die Abflugzeit anpassen.
- Anschliessend werden die Kunden über das bevorzugte Medium (SMS od. Email) informiert.

# Arten von Modulen

- **Bibliotheken:** Sammlung von oft benötigten und thematisch zusammengehörenden Funktionen.  
**Beispiele:** Datum-Modul (Operationen auf Kalenderdaten), Trigonometrie-Modul (Winkelfunktionen), Systemschnittstellen-Modul.
- **Abstrakte Datentypen:** Modul implementiert einen neuen Datentyp und stellt die darauf definierten Operationen zur Verfügung.  
**Beispiele:** Mehrdimensionale Tabellen, komplexe Zahlen und Koordinaten.

## Arten von Modulen (forts.)

- **Physische Systeme** insbesondere in technischen Anwendungen der Informatik.  
**Beispiele:** Sensorsystem, Gerätetreiber, Kommunikation, Anzeigetafel, usw.
- **Logisch-konzeptionelle Systeme:** logisch-konzeptionelle Systeme modellieren und für andere Komponenten auf hoher Abstraktionsstufe nutzbar machen.  
**Beispiele:** Grafik, Datenbank, Messaging, GUIs, usw.

# Wichtige Kriterien des modularen Entwurfs

- **Zerlegbarkeit (Top-Down)**

Teilprobleme sind unabhängig voneinander entwerfbar.

- **Kombinierbarkeit (Bottom-Up)**

Module sind unabhängig voneinander (wieder-)verwendbar.

- **Verständlichkeit**

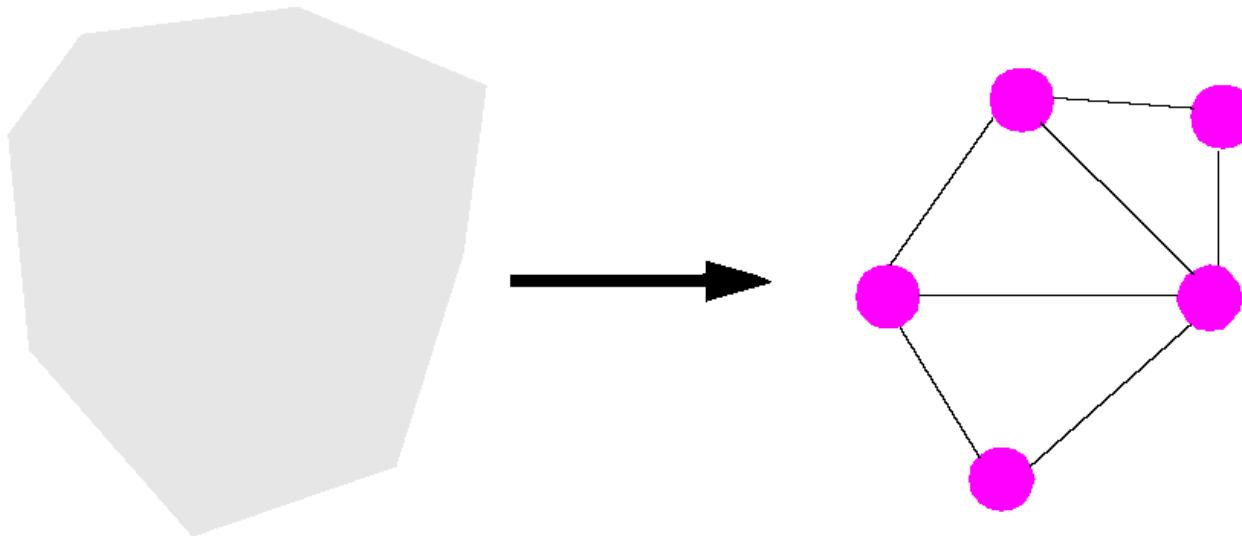
Module sind unabhängig voneinander zu verstehen.

- **Stetigkeit**

Kleine Änderungen der Spezifikation führen nur zu kleinen Änderungen im Code.

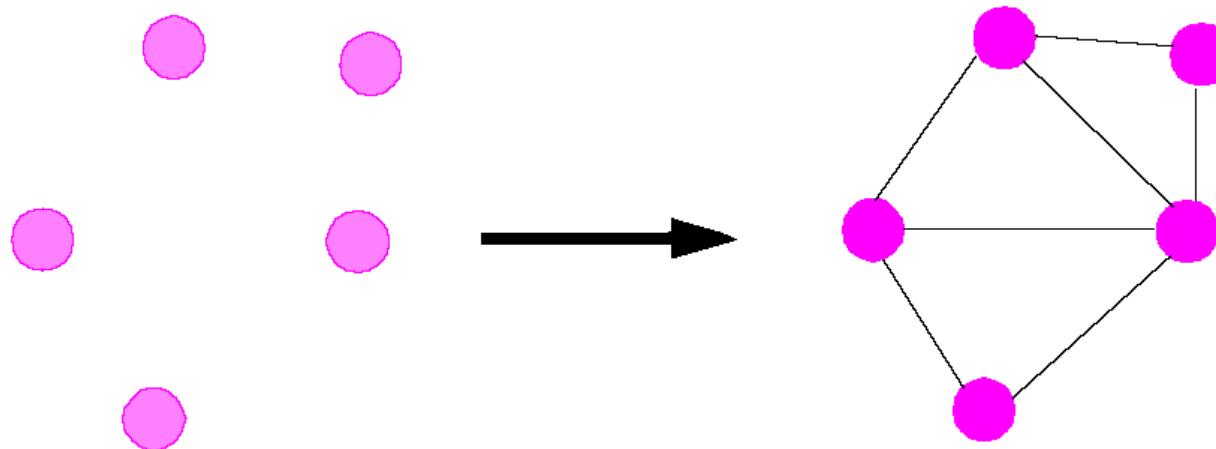
# Zerlegbarkeit (Top-Down)

- Zerlege ein Softwareproblem in eine Anzahl weniger komplexe Teilprobleme und verknüpfe diese so, dass die Teile möglichst unabhängig voneinander bearbeitet werden können.
- Die Zerlegung wird häufig rekursiv angewendet: Teilprobleme können so komplex sein, dass sich eine weitere Zerlegung aufdrängt.



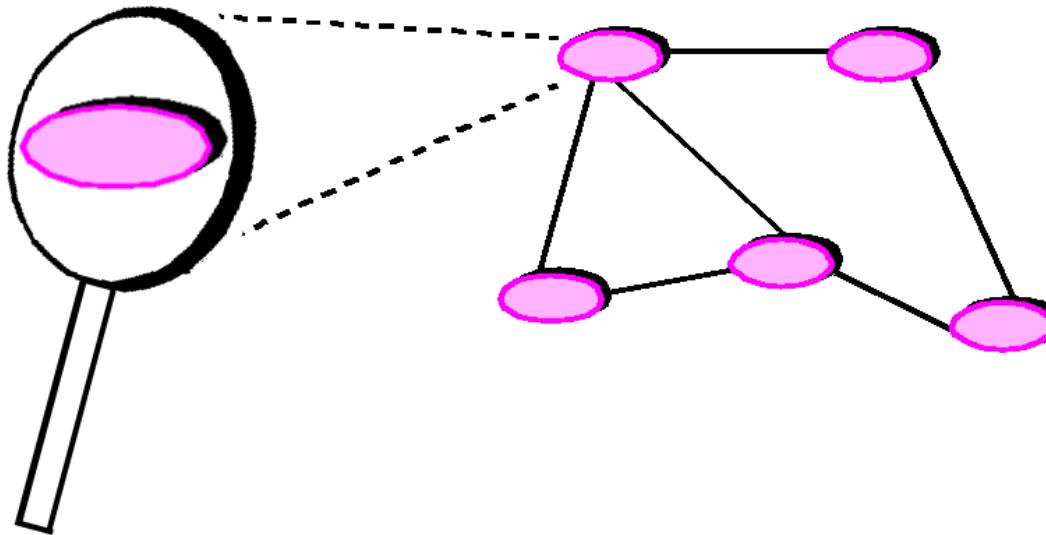
# Kombinierbarkeit (Bottom-Up)

- Strebe möglichst frei kombinierbare Software-Elemente an, die sich auch in einem anderen Umfeld wieder einsetzen lassen.
- Kombinierbarkeit und Zerlegbarkeit sind voneinander unabhängige Eigenschaften.



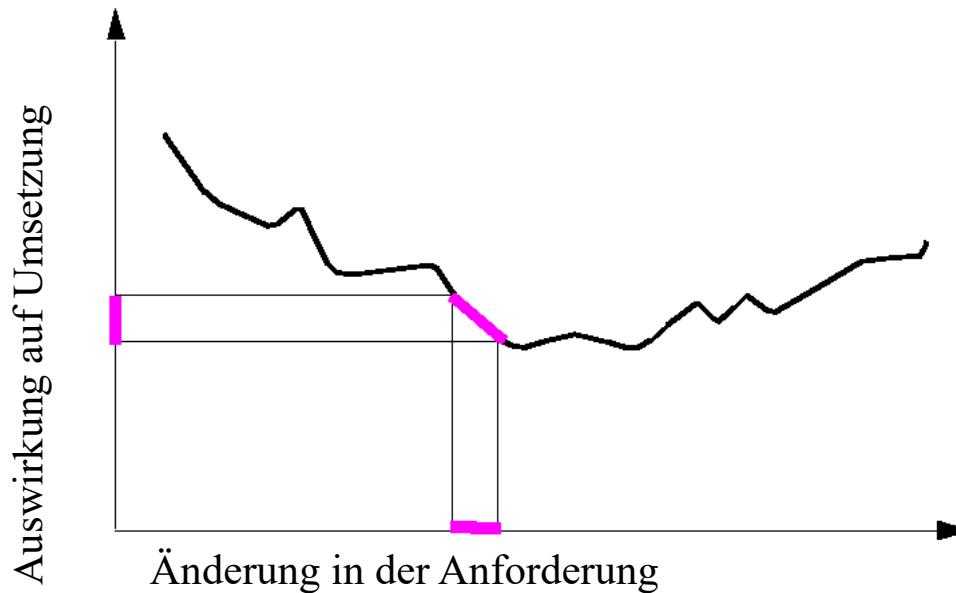
# Verständlichkeit

- Der Quellcode eines Moduls soll auch verstehtbar sein, ohne dass man die anderen Module des Systems kennt.
- Softwareunterhalt setzt voraus, dass die Teile eines Systems unabhängig von einander zu verstehen und zu warten sind.



# Stetigkeit

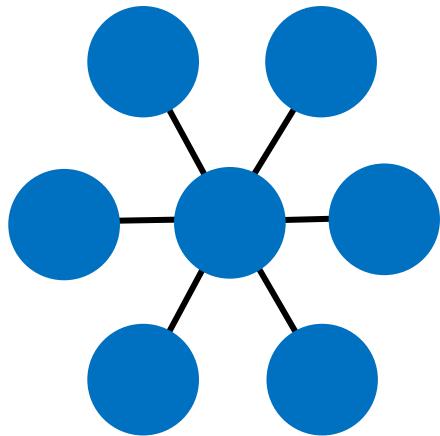
- Von einer kleinen Änderung der Anforderungen soll auch nur ein kleiner Teil der Module betroffen sein.
- Es ist oft unvermeidlich, dass sich im Laufe eines Projektes die Anforderungen ändern. Stetigkeit bedeutet, dass dies nicht die ganze Systemstruktur beeinflusst, sondern sich lediglich auf einzelne Module auswirkt.



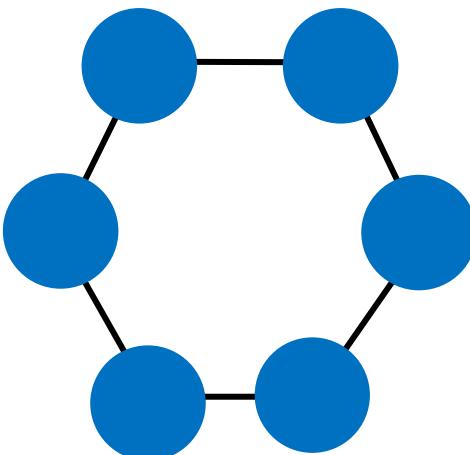
# Beispiel: Auswirkung von Änderungen

- Wie wirkt sich eine Änderung an einem Modul aus?

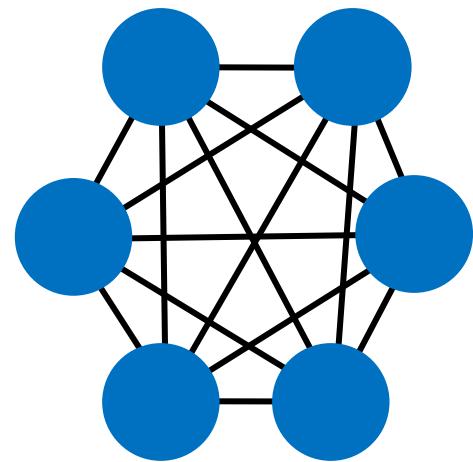
**Aufteilung A:**



**Aufteilung B:**



**Aufteilung C:**



# Prinzipien des modularen Entwurfs

- **Lose Kopplung:** Schmale Schnittstellen um nur das wirklich Benötigte auszutauschen.
- **Starke Kohäsion:** Hoher Zusammenhalt innerhalb eines Moduls.
- **Information Hiding:** Modul ist nach aussen nur über seine Schnittstelle bekannt.
- **Wenige Schnittstellen:** minimale Anzahl Schnittstellen (Aufrufe, Daten).
- **Explizite Schnittstellen:** Aufrufe und gemeinsam genutzte Daten sind im Code ersichtlich.
- **Wenige Abhängigkeiten pro Modul:** Reduktion der Auswirkung von Änderungen auf andere Module.

# **Modularisierung: Iteratives Vorgehen**

- 1. Zerlegung (Top-Down oder Bottom-Up) unter Anwendung der Prinzipien:**
  - Wenig Kopplung und viel Kohäsion.
  - Information-Hiding.
  - Wenige und explizite Schnittstellen.
- 2. Beurteilung hinsichtlich der Kriterien:**
  - Zerlegbarkeit: Module aufgeteilt und unabhängig bearbeitbar?
  - Kombinierbarkeit: Können Module wiederverwendet werden?
  - Verständlichkeit: Viel Kohärenz und wenig Kopplung?
  - Stetigkeit: Auswirkung von Änderungen?
  - Korrekte Modularität: Bibliothek, abstrakter Datentyp, physische Kapsel oder logische Kapsel.
- 3. Falls Kriterien nicht zufriedenstellend: Zurück zu 1.**

# Übung: Modularer Entwurf für Testautomationssystem

Ein System zum Test von Kommandozeilenprogrammen soll in Module zerlegt werden:

- Das System liest seine Konfiguration aus einer Datei. Die Datei enthält die Angabe über das auszuführende Programm und den Testfällen.
- Jeder Testfall besteht aus einer Eingabe und einer erwarteten Ausgabe.
- Das System muss das zu testende Programm für jeden Testfall isoliert in einem neuen Prozess ausführen.
- Anschliessend soll das System die Ausgabe des Programms mit der zu erwartenden Ausgabe des Testfalls vergleichen.
- Nach Ausführung aller Testfälle erstellt das System einen Bericht über den Testlauf. Dieser Bericht enthält für jeden ausgeführten Testfall: Eingabe, erwartete Ausgabe, effektive Ausgabe und das Resultat (Passed, Failed).

Dekomposition des Systems in Module? Top-Down? Bottom-Up?

# Modularisierung

Ein System sinnvoll in Module aufzuteilen ist eine der anspruchsvollsten Aufgaben in der Informatik.

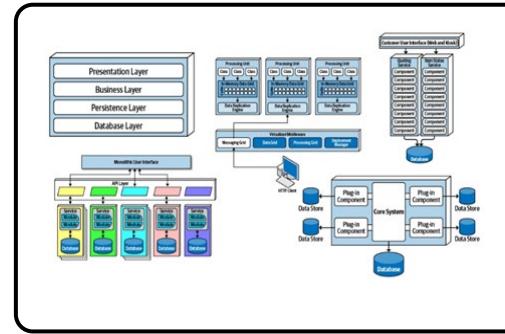
- Lesen Sie dazu den Klassiker **On the Criteria To Be Used in Decomposing Systems into Modules** von David L. Parnas.  
[https://www.win.tue.nl/~wstomv/edu/2ip30/references/criteria\\_for\\_modularization.pdf](https://www.win.tue.nl/~wstomv/edu/2ip30/references/criteria_for_modularization.pdf)
- Oder je nach Vorliebe folgenden Blog-Beitrag, welcher den Inhalt von Parnas mit der modernen Softwareentwicklung von Heute vergleicht:  
<https://blog.acolyer.org/2016/09/05/on-the-criteria-to-be-used-in-decomposing-systems-into-modules/>

# **Schichtenarchitektur**

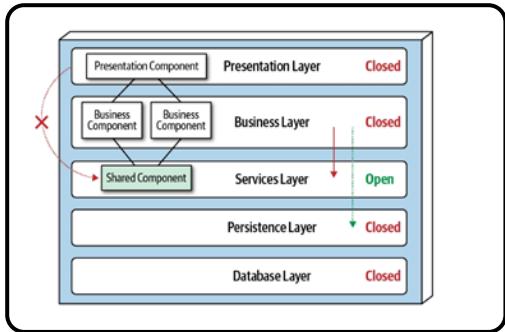
# Was ist Softwarearchitektur?

Availability	Reliability	Testability
Scalability	Security	Agility
Fault Tolerance	Elasticity	Recoverability
Performance	Deployability	Learnability

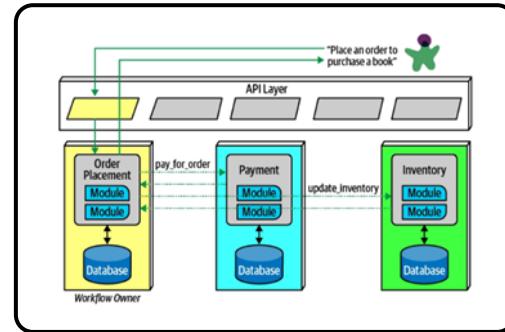
Architektonische Eigenschaften



Struktur



Architektonische Entscheidungen

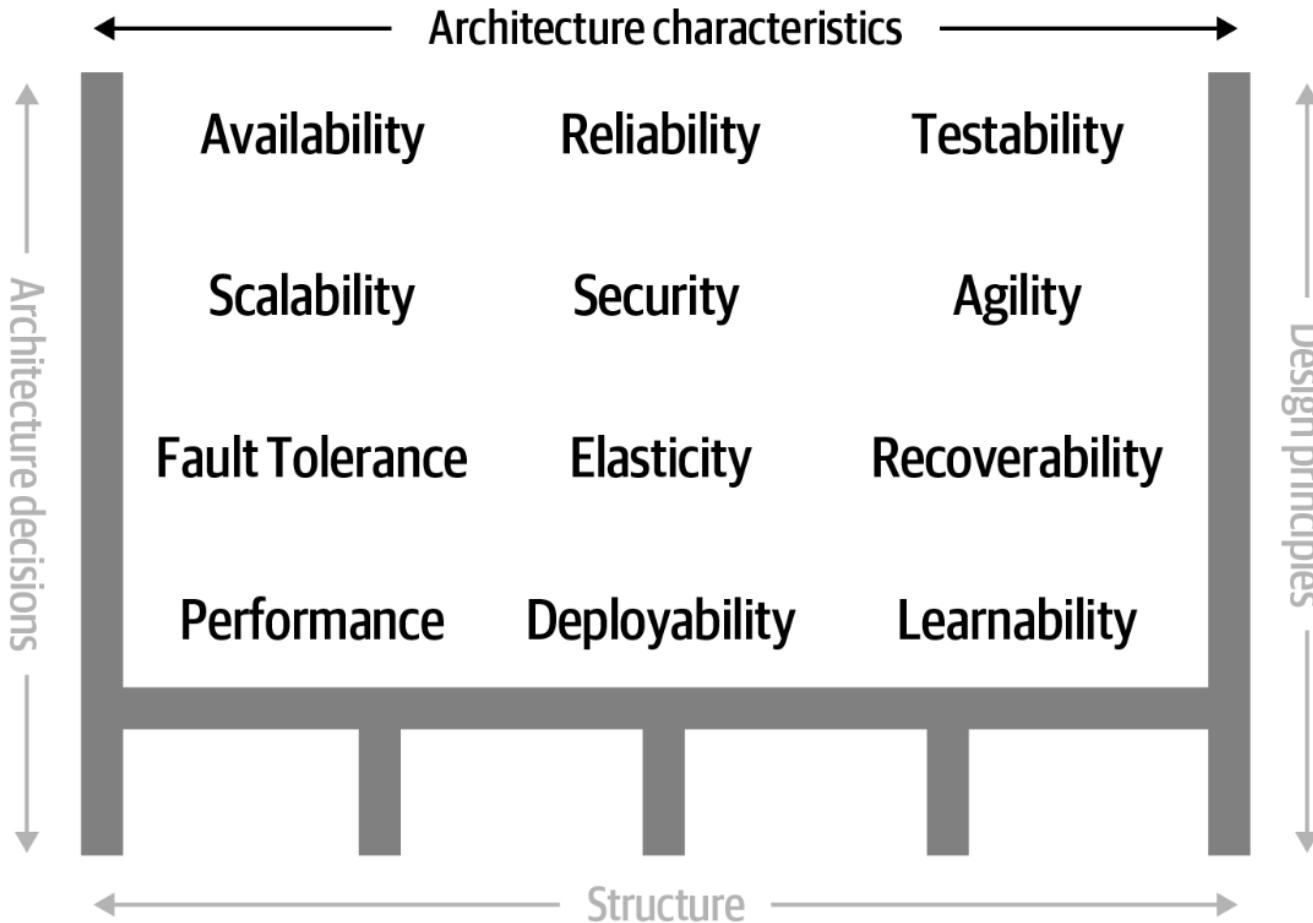


Entwurfsprinzipien

Quelle: Handbuch modernder Softwarearchitektur

# Architektonische Eigenschaften

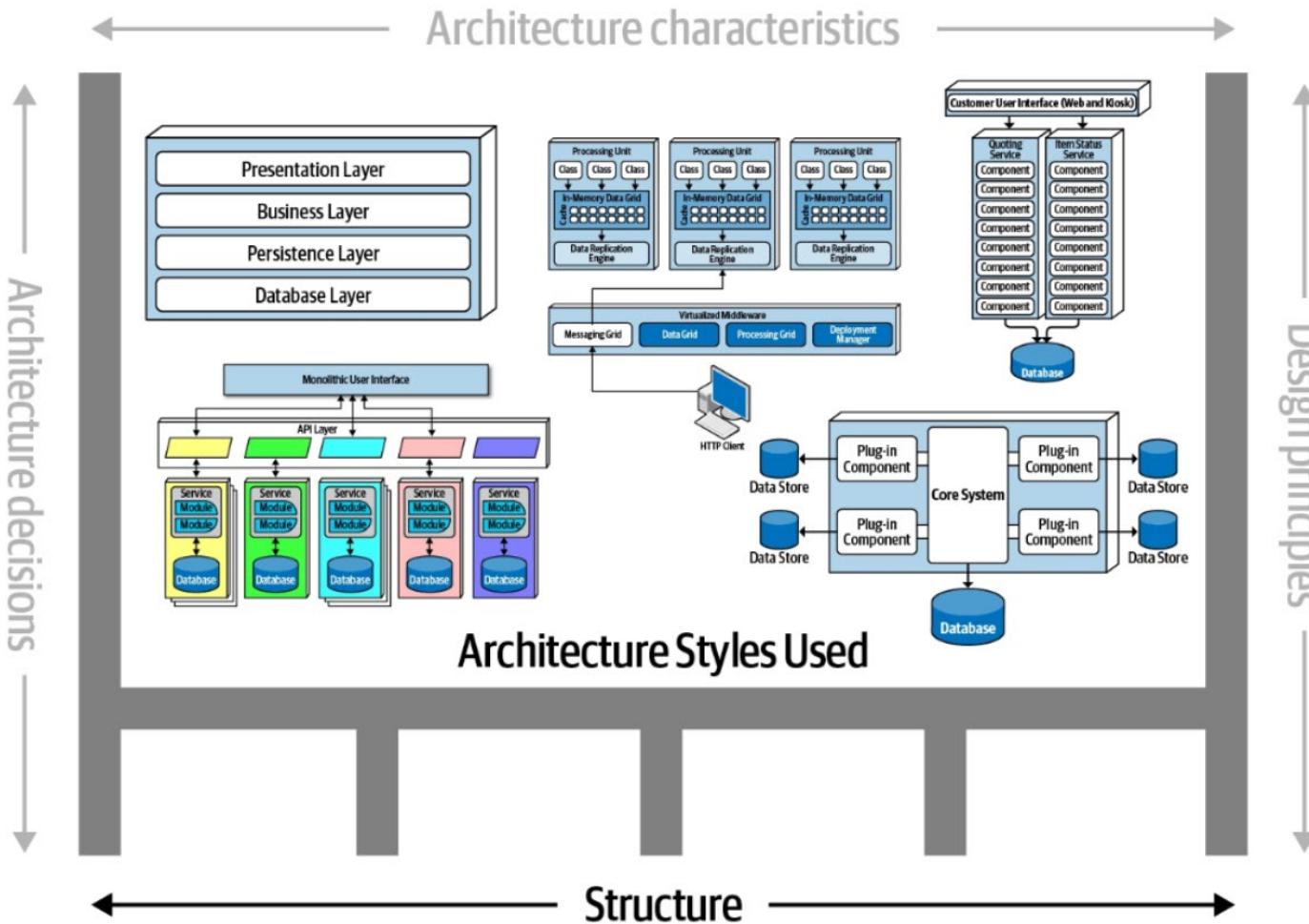
- Nichtfunktionale Eigenschaften eines Systems, welche zur ordentlichen Funktionsweise notwendig sind.



Quelle: Handbuch modernder Softwarearchitektur

# Struktur

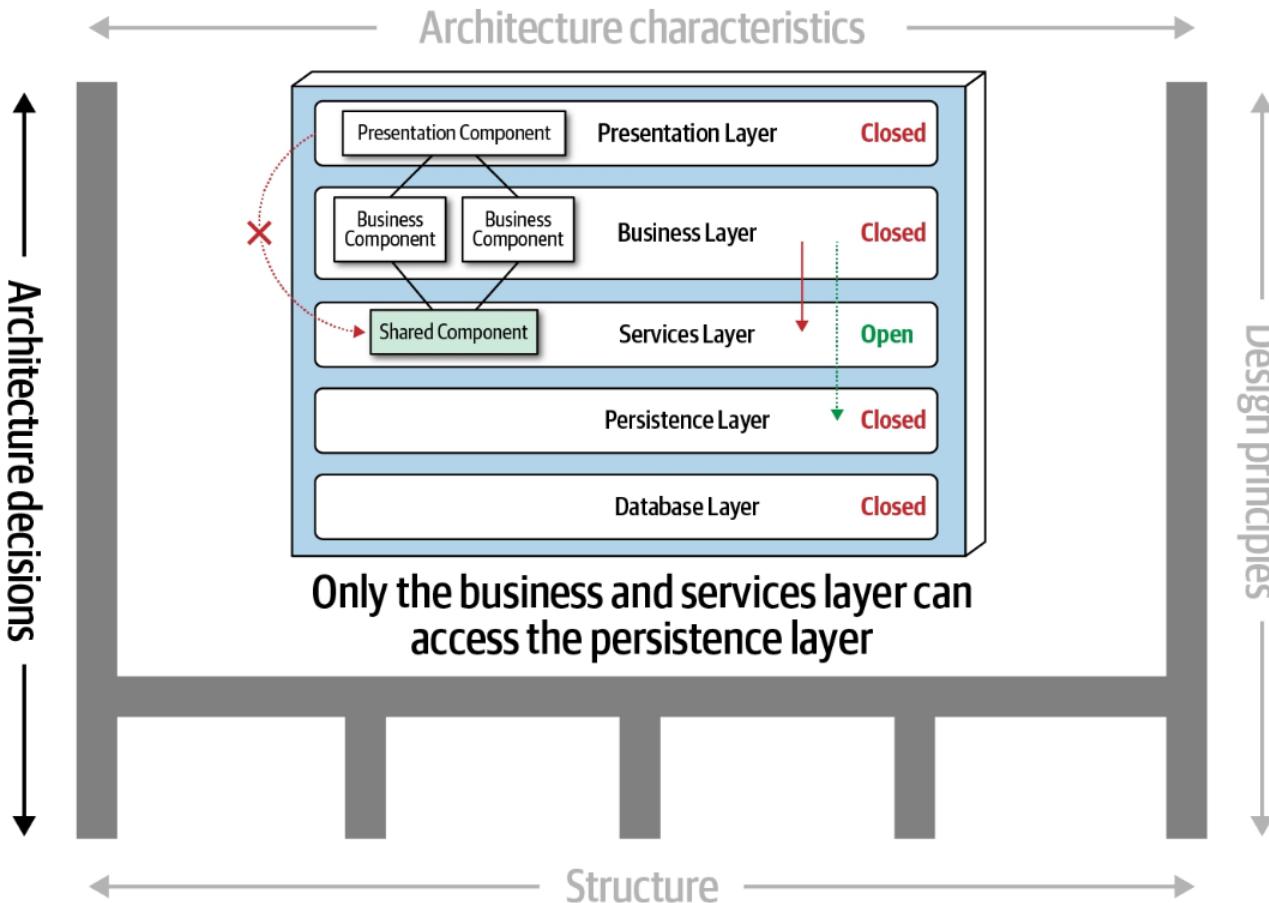
## – Verwendete Architekturstile



Quelle: Handbuch modernder Softwarearchitektur

# Architektonische Entscheidungen

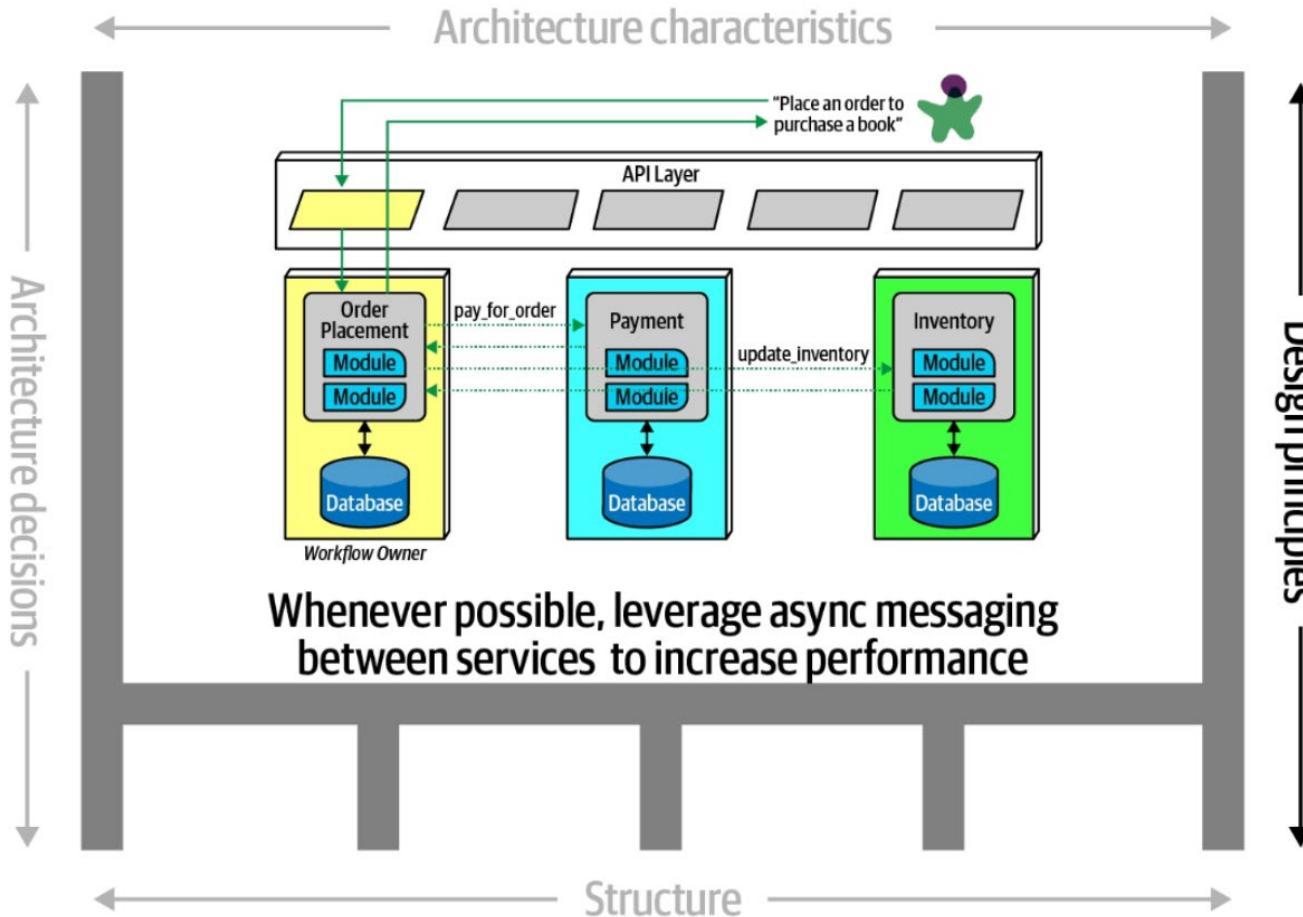
- Regeln, welche Entwickler beim Entwurf der Komponenten befolgen müssen.



Quelle: Handbuch modernder Softwarearchitektur

# Entwurfsprinzipien

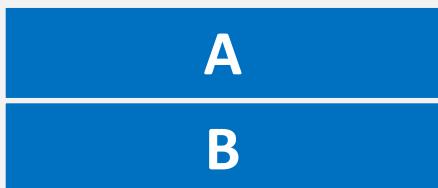
- Richtlinien, welche Entwickler bei Entwurf der Komponenten des Systems anwenden **sollen**.



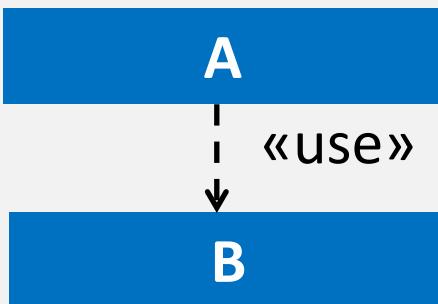
Quelle: Handbuch modernder Softwarearchitektur

# Was sind Schichten?

## Darstellung:



oder

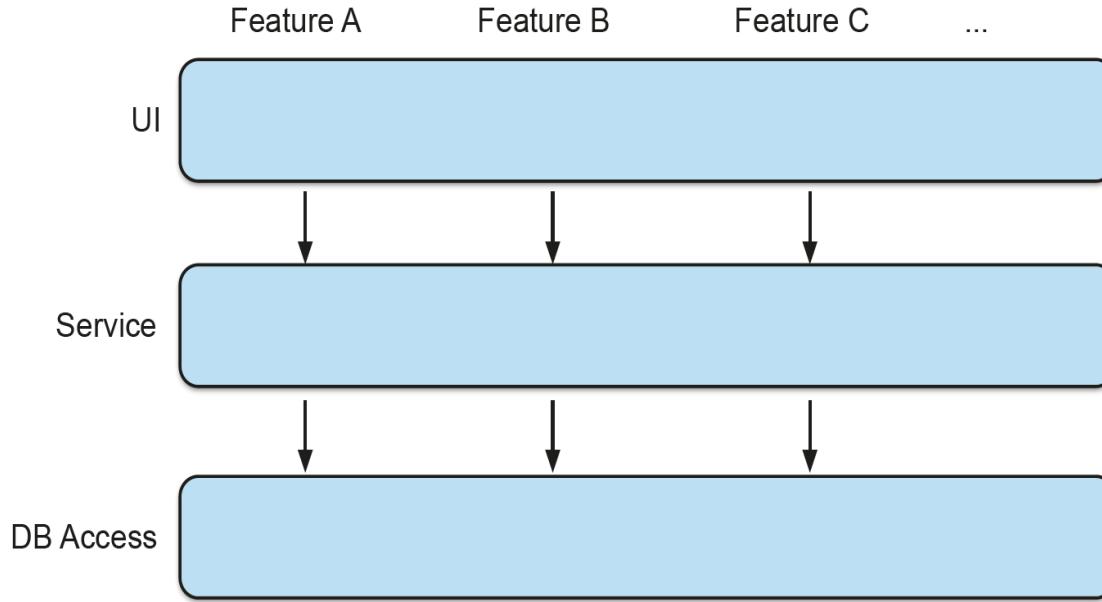


In diese Richtung ist die Verwendung erlaubt

- Modulhierarchie, in welcher öffentliche Methoden in Schicht B von der Software in Schicht A **genutzt werden dürfen**, aber nicht umgekehrt.
- Man spricht von einer **use-Beziehung** wenn das korrekte Funktionieren von A von einer korrekten Implementation von B **abhängt**.

# Schichtenarchitektur

- Technische Modularisierung oft entlang organisatorischen Einheiten.
- Komponenten einzelnen Schichten zugeordnet:

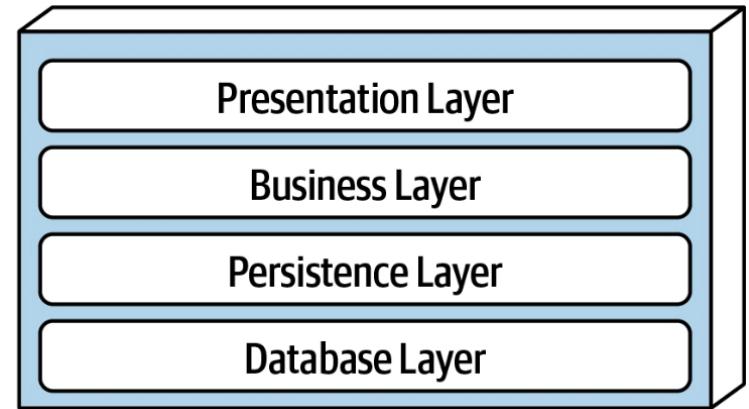


- Basis für komplexere Architekturen.
- Deployment-Monolith: Typischerweise müssen immer alle Schichten gleichzeitig angepasst und vollständig ausgeliefert / installiert werden.

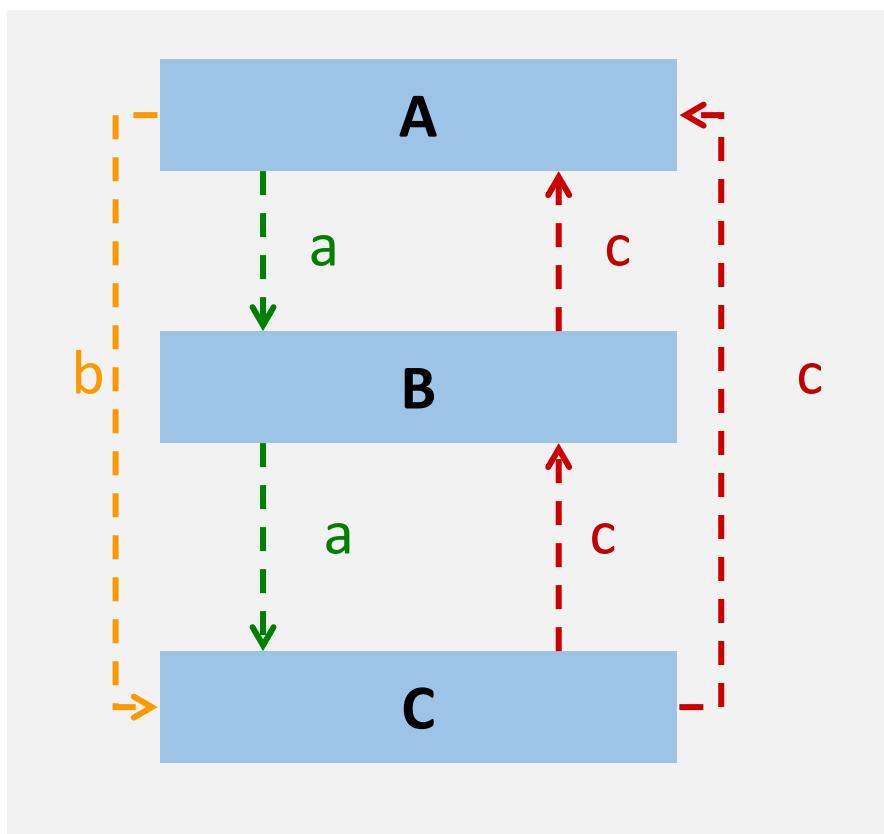
Quelle: Modulare Software Architektur, Herbert Dowalil

# Typische Schichten

- **Presentation Layer (Anwendungen):** Darstellung und Benutzerinteraktion mit der Geschäftslogik einer Applikation.
- **API Layer (Services):** Bereitstellung des Zugriffs auf die Geschäftslogik.
- **Business Layer:** Geschäftslogik (Funktionalität und Datenstrukturen).
- **Service Layer:** Hilfsfunktionen für Komponenten einer darüberliegenden Schicht.
- **Persistence Layer:** Abstraktion des Datenzugriffs (z.B. möchten wir Kundendaten oder Kontoinformationen laden und nicht einfach Textfiles).
- **Database Layer:** Zugriff auf den Storage (z.B. Definition des Schemas bei relationalen Datenbanken).

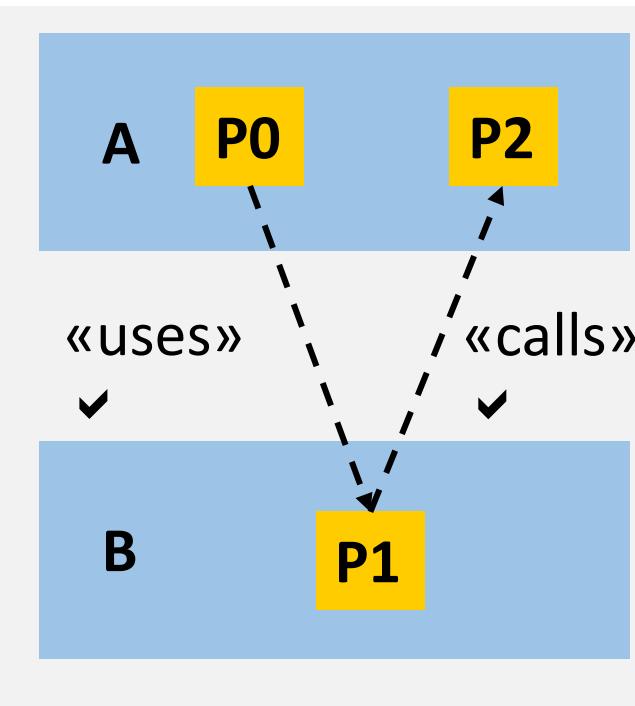


# Schichtenbeziehungen: Zulässigkeit



- **a:** ok.
- **b:** gefährlich, falls nicht vorgesehen, wie z.B. bei offenen Schichtarchitekturen.
- **c:** nicht zulässig (keine zyklischen Abhängigkeiten zwischen Schichten)!

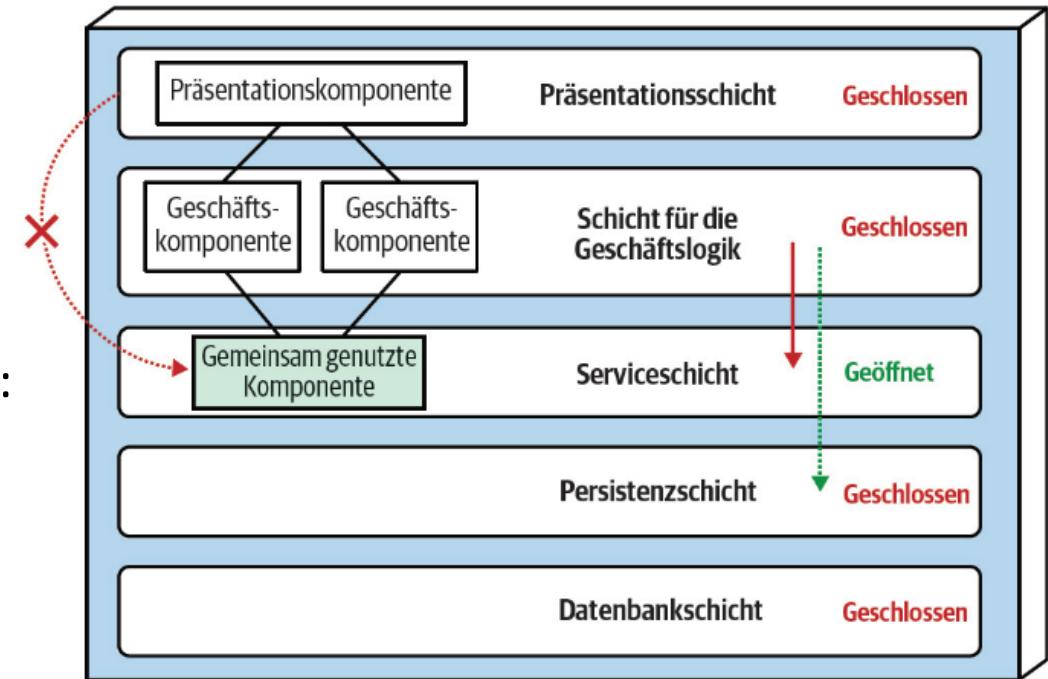
# Weitere Schichtenbeziehung



- Eine Klasse P1 kann P2 aufrufen ohne eine use-Beziehung mit P2 zu haben.
- **Beispiel:** P2 sei ein Errorhandler, dessen Referenz von P0 an P1 übergeben wurde. Die Referenz des Errorhandlers ist nicht in P1 festkodiert

# Offene Schichtenarchitektur: Offene und geschlossene Schichten

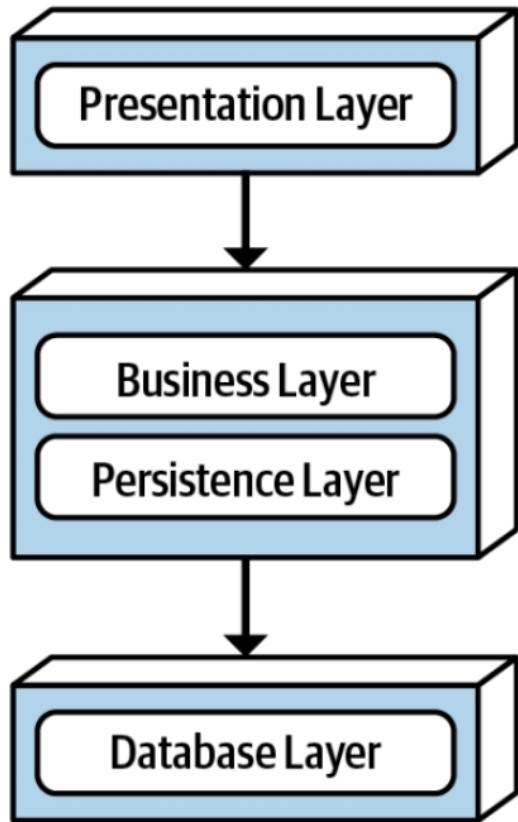
- Offene Schichtenarchitektur:  
Schichten können **offen**  
oder **geschlossen** sein.
- Offene Schichten können von  
direkt darüberliegender  
Schicht **übersprungen** werden:
  - Vermeidung horizontaler  
Abhängigkeiten.
  - Performancegewinn,  
falls eine Schicht Anfragen  
nur weiterreichen würde.



⇒ **Alternative:** Schichten rekursiv verschachteln.

**Quelle:** Handbuch moderner Softwarearchitektur, Richards und Ford

# Schichten vs. Tier

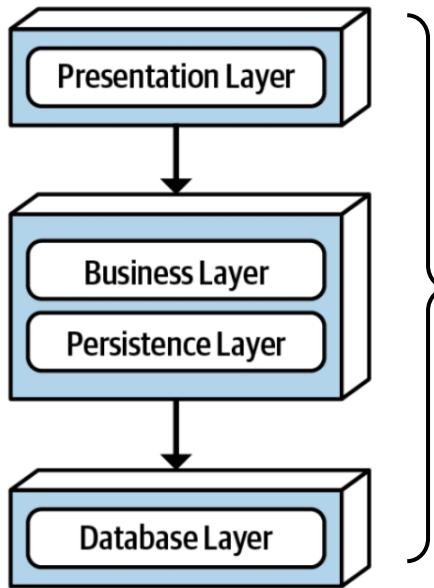


- **Schichten:** Logische Separierung.
  - **Tier:** Zusätzlich eine physische Separierung oft kombiniert mit Verteilung (\*).
- (\*) heutzutage oft als Service.

**Beispiel (links):** vier Schichten auf drei Tiers.

**Quelle:** Handbuch moderner Softwarearchitektur, Richards und Ford

# Bewertung der Schichtbasierte Architektur

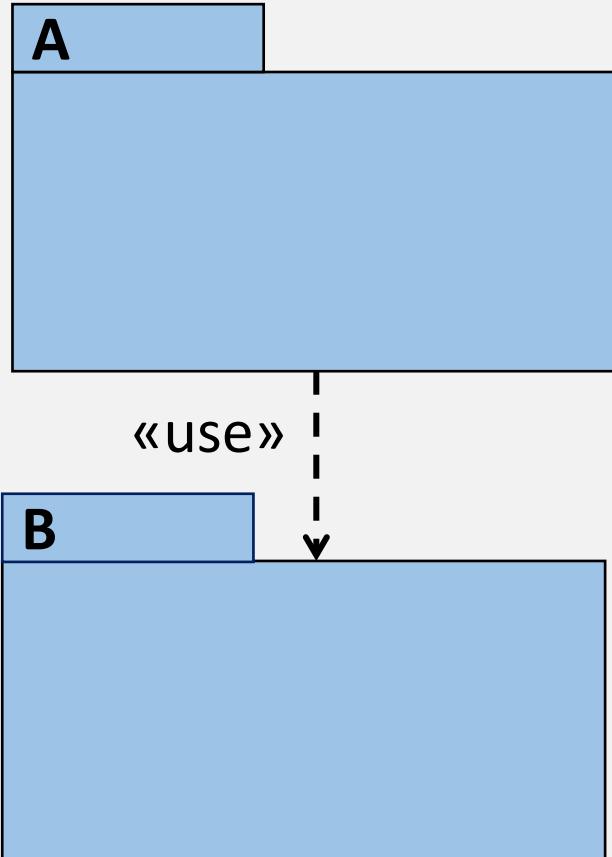


- Einfacher und kostengünstiger Architekturstil.
- Auslieferung als eine Einheit.
- Verteilung des Business-Layers i.d.R. nicht (einfach) möglich -> Skalierung nur innerhalb eines Systems.

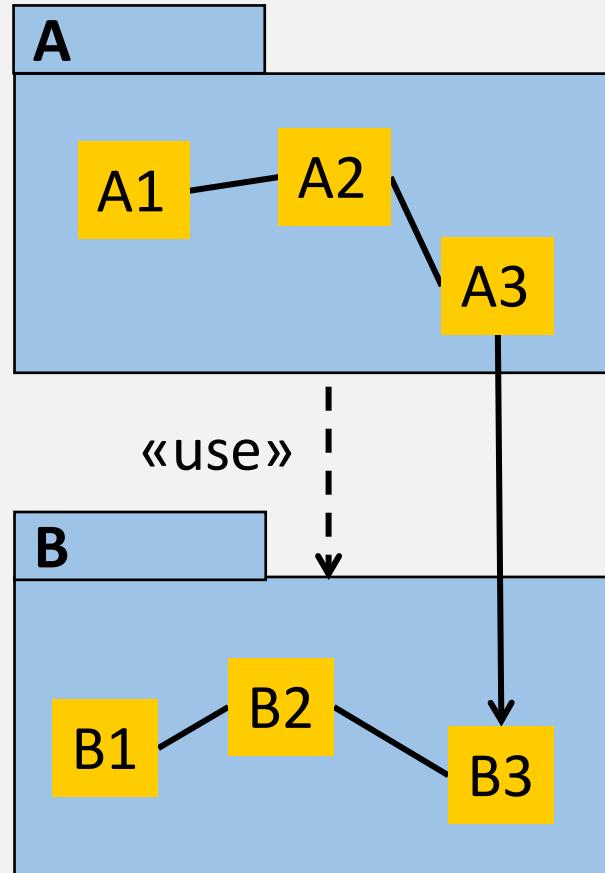
Architektonische Eigenschaft	Bewertung
Partitionierungstyp	Technisch
Anzahl der Quanten	1
Bereitstellbarkeit	★
Elastizität	★
Entwicklungsfähigkeit	★
Fehlertoleranz	★
Modularität	★
Gesamtkosten	★★★★★
Performance	★★
Verlässlichkeit	★★★
Skalierbarkeit	★
Einfachheit	★★★★★
Testbarkeit	★★

Quelle: Handbuch modernder Softwarearchitektur

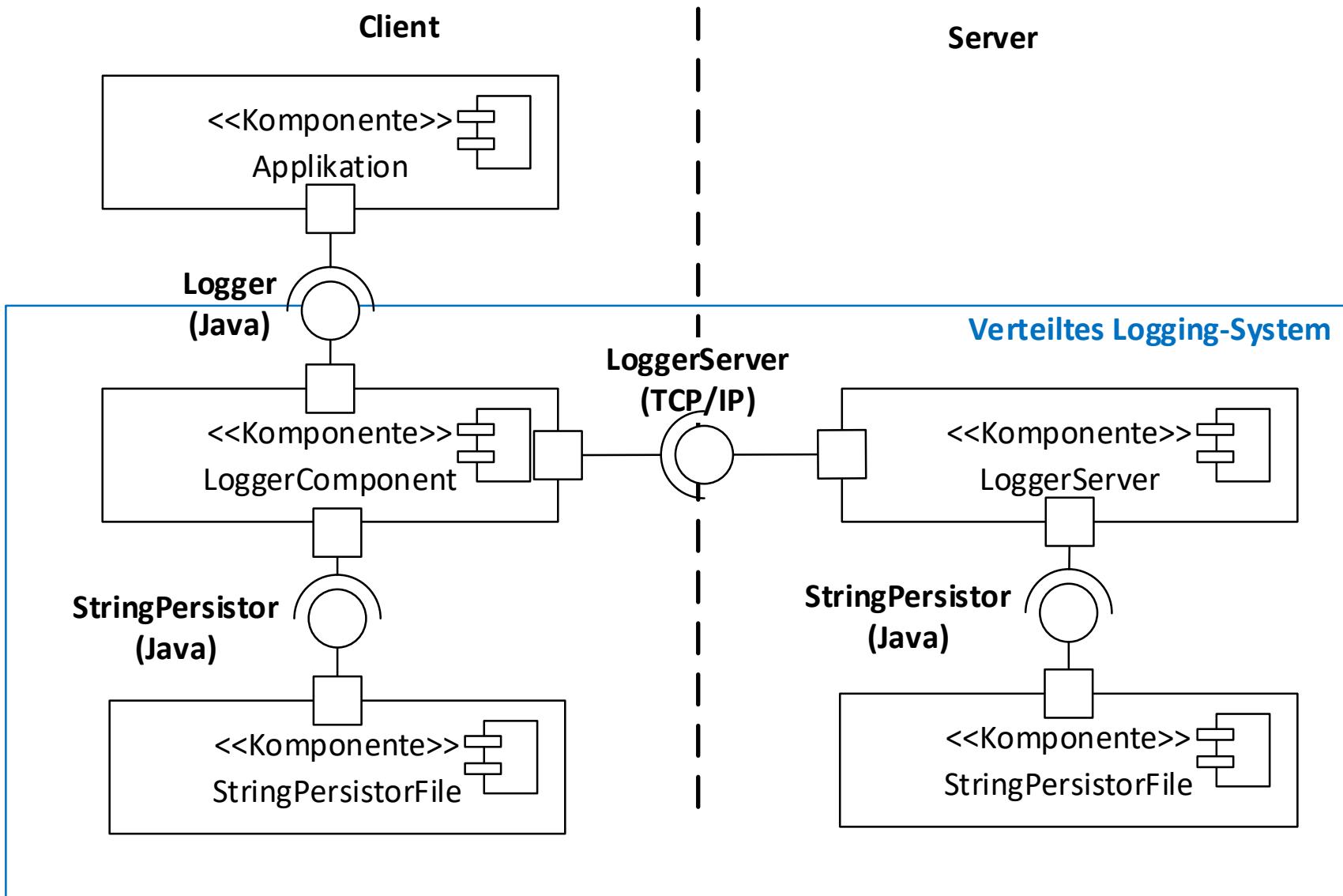
# Schichten in UML



more details:



# Modularität im Logger?



# Zusammenfassung

- Modul: in sich abgeschlossener Teil des gesamten Programmcodes.
- Modulkonzept 1972 David Parnas.
- Kopplung und Kohäsion optimieren!
- Entwurfskriterien: Zerlegbarkeit / Kombinierbarkeit / Verständlichkeit / Stetigkeit.
- Entwurfsprinzipien: lose Kopplung / starke Kohäsion / Information Hiding / wenige & explizite Schnittstellen.
- Entwurfsvorgehen: anspruchsvolle Aufgabe.
- Schichtenarchitektur: Architekturstil basierend auf dem Schichtenkonzept.
- Eigenschaften der Schichtenarchitektur: Günstig, verständlich, aber wenig flexibel.

# **Fragen?**

# Literatur und Quellen

- Modulare Software Architektur, Herbert Dowalil, 2020, Carl Hanser Verlag.
- Handbuch moderner Softwarearchitektur, Mark Richards und Neal Ford, 2021, O'Reilly / dpunkt.verlag GmbH.
- Grundlagen des modularen Softwareentwurfs, Herbert Dowalil, 2018, Carl Hanser Verlag.
- Modularisierung mit Java 9: Grundlagen und Techniken für langlebige Softwarearchitekturen von Guido Oelmann, 2017, dpunkt.verlag GmbH.
- Object-Oriented Software Construction (second edition) von Bertrand Meyer, 1997, Prentice Hall.

Verteilte Systeme und Komponenten

# Containervirtualisierung

## Docker Container in der (Java-)Entwicklung

Roland Gisler

# Inhalt

- Grundlagen – Was ist Docker?
- Installation von Docker
- Einsatzszenarien
- Nutzung bestehender Images
- Erstellen eigener Images (Basis)
- Docker im Java-Umfeld
- Zusammenfassung

# Lernziele

- Sie kennen die fundamentalen Konzepte von Docker.
- Sie können Docker Container auf Basis bestehender Images starten und nutzen.
- Sie können eigene, einfache Images erstellen und Deployen.
- Sie kennen das Potential, das Docker für Java-Anwendungen bietet.
- Sie können Java-Anwendungen in Docker-Images deployen.

# **Grundlagen – Was ist Docker?**

# Docker – Was ist das?

- Sehr stark vereinfacht erklärt: Leichtgewichtige, virtuelle Maschinen, die zwar vollständig isoliert, aber trotzdem direkt auf dem Host ausgeführt werden → schlanker und schneller.
- Motivation und Idee: Lauffähige Applikationen inklusive ihrer Laufzeitumgebung in einer standardisierten Form verteilen können, ohne komplizierte Installation. Persistente Nutzdaten in getrennten, virtuellen Dateisystemen auslagern. Einfache Updates.
- Zentrale, wichtige Konzepte:
  - Container, Images, Volumes, Netzwerke, Registry, Repository
- Docker ist klar in der Linux-Welt zuhause (2013 dort entstanden), kann aber auch auf Windows und Mac genutzt werden.
- **Wichtig:** Unix-Kenntnisse sind für Docker unabdingbar!

# Funktionsweise von Docker

- Verwendet Techniken aus dem Linux-Kernel (z.B. «cgroups» und «namespaces») um Prozesse und Speicher vollständig zu trennen.
  - Prozesse laufen «isoliert» auf dem selben Betriebssystemkern (Scheduling), nutzen das selbe Memory (aber isoliert), können auf ein (virtuelles) Dateisystem zugreifen und auch auf Netzwerkdienste.
- Die verschiedenen Basis-Techniken wurden später von Docker in einer Schnittstelle «libcontainer» zusammengefasst.
- Docker kann selber problemlos in einer virtuellen Maschine (z.B. VirtualBox, VMWare etc.) ausgeführt werden. Und seit einiger Zeit auch auf dem Hyper-V (Microsoft, Basis für WSL) auf Windows.
  - siehe Hinweis zu → Installation.

# Wichtige Begriffe und Konzepte von Docker

- **Container:** Ist eine laufende Instanz einer Anwendung in Docker. Ein Container basiert immer auf einem →Image.
- **Image:** Enthält die im Docker-Container ausführbare Anwendung in Form eines virtuellen, geschichteten Dateisystems. Images sind read-only und werden in →Repositories auf →Registries abgelegt.
- **Tags:** Versionslabels für Images (nicht analog zu VCS!)
- **Volume:** Virtuelles Dateisystem, das für die persistente (Nutzdaten-)Speicherung in Containern genutzt werden kann.
- **Registry:** Ein Zugangspunkt (Server) für eine Anzahl von →Repositories für die Verteilung von Docker Images.
- **Repository:** Identifikation einer Gruppe von →Images in verschiedenen Versionen (Tags) in einer →Registry.

# Erste Demo

The screenshot shows a Docker Desktop interface with several containers running. A terminal window displays the execution of the fibo(80) function across multiple runs, comparing recursive, iterative, and memoized approaches.

**Docker Desktop**

**Containers** (Showing 1 items)

NAME	IMAGE	STATUS	PORT(S)	STARTED	ACTIONS
clever_franklin	rgisler/fiboperf:f1	Running	-	0 seconds ago	[...]

**Extensions** (BETA)

- Disk usage
- Logs Explorer
- Portainer
- Snyk
- Add Extensions

**RAM 5.17GB CPU 1.47% Connected to Docker Desktop**

**ctop - 08:32:10 CEST 7 containers**

NAME	CID	CPU	MEM	NET RX/TX	IO R/W	PIDS
cool_habit	f0f3b8109ea6	97%	657M / 24.92G	586B / 0B	0B / 0B	27
determined_habit	4d3aca0a32ce	98%	632M / 24.92G	1K / 0B	0B / 0B	27
eager_swirls	d8c961bc2965	99%	754M / 24.92G	586B / 0B	0B / 0B	27
ecstatic_gates	b105a214ade6	98%	723M / 24.92G	586B / 0B	0B / 0B	27
loving_burnell	b3991f405711	99%	575M / 24.92G	516B / 0B	0B / 0B	27
mystifying_boyd	a7f8fff69e79e	98%	556M / 24.92G	586B / 0B	0B / 0B	27
sharp_sammet	fd317da35bf2	98%	674M / 24.92G	516B / 0B	0B / 0B	27

**Windows PowerShell**

```
2022-10-19 06:33:05,954 INFO - Fibonacci f(20) = 6765 in 108ms rekursiv mit Array-Cache berechnet mit 19 Additionen.
2022-10-19 06:33:06,349 INFO - Fibonacci f(20) = 6765 in 493ms rekursiv mit Map-Cache berechnet mit 19 Additionen.2022-10-19 06:33:06,350
INFO - Fibonacci f(20) = 6765 in 0.02181ms rekursiv berechnet mit 10945 Additionen.
2022-10-19 06:33:06,350 INFO - === fibo(20) - 2. Messung (warm) ===
2022-10-19 06:33:06,356 INFO - Fibonacci f(20) = 6765 in 7ms iterativ berechnet mit 19 Additionen.
2022-10-19 06:33:06,389 INFO - Fibonacci f(20) = 6765 in 40ms nach Binet berechnet.
2022-10-19 06:33:06,473 INFO - Fibonacci f(20) = 6765 in 104ms rekursiv mit Array-Cache berechnet mit 19 Additionen.
2022-10-19 06:33:06,863 INFO - Fibonacci f(20) = 6765 in 487ms rekursiv mit Map-Cache berechnet mit 19 Additionen.
2022-10-19 06:33:06,864 INFO - Fibonacci f(20) = 6765 in 0.030539ms rekursiv berechnet mit 10945 Additionen.
2022-10-19 06:33:06,864 INFO - .
2022-10-19 06:33:06,864 INFO - === fibo(40) - 1. Messung (warm) ===
2022-10-19 06:33:06,871 INFO - Fibonacci f(40) = 102334155 in 8ns iterativ berechnet mit 39 Additionen.
2022-10-19 06:33:06,983 INFO - Fibonacci f(40) = 102334155 in 39ns nach Binet berechnet.
2022-10-19 06:33:07,043 INFO - Fibonacci f(40) = 102334155 in 174ms rekursiv mit Array-Cache berechnet mit 39 Additionen.
2022-10-19 06:33:07,043 INFO - Fibonacci f(40) = 102334155 in 1034ms rekursiv mit Map-Cache berechnet mit 39 Additionen.
2022-10-19 06:33:08,283 INFO - Fibonacci f(40) = 102334155 in 332.518ms rekursiv berechnet mit 165580146 Additionen.
2022-10-19 06:33:08,284 INFO - .
2022-10-19 06:33:08,284 INFO - === fibo(40) - 2. Messung (warm) ===
2022-10-19 06:33:08,210 INFO - Fibonacci f(40) = 102334155 in 7ns iterativ berechnet mit 39 Additionen.
2022-10-19 06:33:08,240 INFO - Fibonacci f(40) = 102334155 in 37ns nach Binet berechnet.
2022-10-19 06:33:08,375 INFO - Fibonacci f(40) = 102334155 in 168ms rekursiv mit Array-Cache berechnet mit 39 Additionen.
2022-10-19 06:33:09,139 INFO - Fibonacci f(40) = 102334155 in 954ms rekursiv mit Map-Cache berechnet mit 39 Additionen.
2022-10-19 06:33:09,477 INFO - Fibonacci f(40) = 102334155 in 338.3276ms rekursiv berechnet mit 165580146 Additionen.
2022-10-19 06:33:09,478 INFO - .
2022-10-19 06:33:09,478 INFO - === fibo(80) - 1. Messung (warm) ===
2022-10-19 06:33:09,497 INFO - Fibonacci f(80) = 23416728348467685 in 17ms iterativ berechnet mit 79 Additionen.
2022-10-19 06:33:09,523 INFO - Fibonacci f(80) = 23416728348467744 in 38ms nach Binet berechnet.
2022-10-19 06:33:09,782 INFO - Fibonacci f(80) = 23416728348467685 in 320ms rekursiv mit Array-Cache berechnet mit 79 Additionen.
2022-10-19 06:33:11,451 INFO - Fibonacci f(80) = 23416728348467685 in 2085ms rekursiv mit Map-Cache berechnet mit 79 Additionen.
2022-10-19 06:33:11,451 WARN - NOP - Rekursiver Fibonacci ist fuer n>80 viel zu langsam.
2022-10-19 06:33:11,451 INFO - === fibo(80) - 2. Messung (warm) ===
2022-10-19 06:33:11,473 INFO - Fibonacci f(80) = 23416728348467685 in 27ms iterativ berechnet mit 79 Additionen.
2022-10-19 06:33:11,504 INFO - Fibonacci f(80) = 23416728348467744 in 38ms nach Binet berechnet.
```

# **Installation von Docker**

# Wichtig: Account auf <https://hub.docker.com/>

- **WICHTIG:** Auch wenn Sie im Schulnetz nur wenig mit Docker arbeiten wollen, sind wir **ALLE** darauf angewiesen, dass Sie noch vor den ersten Versuchen bitte einen **kostenfreien Account (Personal Plan \$0)** auf <https://hub.docker.com/> eröffnen.
  - Danach können Sie u.a. auch eigene Registries nutzen.
- Grund: Ohne Login greifen Sie «anonym» auf die öffentlich Docker-Registries zu, und Docker **beschränkt** die Anzahl Pulls auf das Netzwerk der Organisation in der Sie arbeiten → HSLU-I.
  - Darum immer mit eigenem Account arbeiten!
- Login (**nach** Docker-Installation) in der Shell nicht vergessen:

```
docker login [repository-url]
```

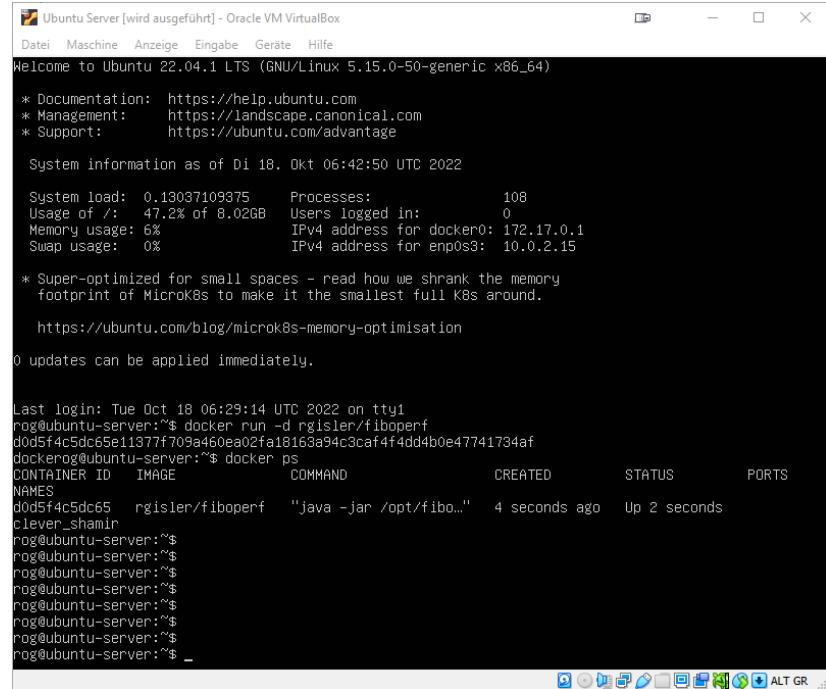
Hinweis: Die Credentials werden in Ihrem \$HOME gespeichert.

# Docker auf Linux (oder Mac)

- Docker ist auf Linux zu Hause. Just do it! 😊
- Installation gemäss Anleitung von «Docker Desktop»:
  - Linux: <https://docs.docker.com/desktop/install/linux-install/>
  - Mac: <https://www.docker.com/>
- Viel Spass!

# Docker auf Windows – Virtuelle Maschine (alt)

- Das ist quasi die «alte» Form der Installation. Wenn Sie nur wenig Experimente in einem völlig isolierten Umfeld machen wollen.
- Nutzen Sie dazu eine Software wie VMWare oder VirtualBox.
- Installieren Sie darin z.B. einen Ubuntu Server 22.04 und ergänzen Sie über `sudo apt install docker.io` die Docker-Installation.
- Das ist vergleichbar mit einem entfernten Docker-Server, wie er z.B. produktiv in einer Cloud betrieben würden.
  - Nachteil: Eingeschränkte Möglichkeiten in unserer Rolle als Entwickler\*in.  
(Integration)



```
Ubuntu Server [wird ausgeführt] - Oracle VM VirtualBox
Datei  Maschine  Anzeige  Eingabe  Geräte  Hilfe
Welcome to Ubuntu 22.04.1 LTS (GNU/Linux 5.15.0-50-generic x86_64)

 * Documentation:  https://help.ubuntu.com
 * Management:    https://landscape.canonical.com
 * Support:        https://ubuntu.com/advantage

System information as of Di 18. Okt 06:42:50 UTC 2022

System load: 0.13037109375  Processes:          108
Usage of /: 47.2% of 8.02GB  Users logged in:      0
Memory usage: 6%             IPv4 address for docker0: 172.17.0.1
Swap usage: 0%               IPv4 address for enp0s3: 10.0.2.15

* Super-optimized for small spaces - read how we shrank the memory
  footprint of MicroK8s to make it the smallest full K8s around.

https://ubuntu.com/blog/microk8s-memory-optimisation

0 updates can be applied immediately.

Last login: Tue Oct 18 06:29:14 UTC 2022 on ttu1
rog@ubuntu-server:~$ docker run -d rgisler/fiboperf
d0d5f4c5dc65e11377f709a460ea02fa18163a94c3caf4f4dd4b0e47741734af
dockero@ubuntu-server:~$ docker ps
CONTAINER ID IMAGE COMMAND CREATED STATUS PORTS
NAMES
d0d5f4c5dc65 rgisler/fiboperf "Java -jar /opt/fibo..." 4 seconds ago Up 2 seconds
clever_shamir
rog@ubuntu-server:~$ 
rog@ubuntu-server:~$ 
rog@ubuntu-server:~$ 
rog@ubuntu-server:~$ 
rog@ubuntu-server:~$ 
rog@ubuntu-server:~$ 
rog@ubuntu-server:~$ 
rog@ubuntu-server:~$ 
```

# Docker auf Windows – auf/mit wsl (aktuell)

- Es lohnt sich, zuerst ein virtuelles (Ubuntu-)Linux auf Windows zu haben, so dass das **WSL** (**Windows Subsystem Linux**) bereits installiert ist (und in der notwendigen Version **2**) vorliegt.
- Windows Store: «**Ubuntu 22.04**» installieren. Dabei werden Sie von Microsoft wenn nötig zu weiteren Schritten angeleitet.

- Ubuntu steht danach (über Hyper-V virtualisiert) in einer Shell zur Verfügung.

- **Wichtig:** Unter `/mnt` sind alle lokalen Laufwerke gemounted!

- So können Sie sich ganz nebenbei auch ein paar grundlegende Linux-Kenntnisse aneignen, die für Docker unabdingbar sind.

```
rog@DEP-ROG:~$ neofetch
  .--/+oossssoo+/-.
   `:+ssssssssssssssssssss+`:
    -+ssssssssssssssssssssyssss+-+
     .osssssssssssssssssdMMMNyyssso.
     /sssssssssssssssssshdmmNNnmyNMNMMyssssss/
     +sssssssssssssssshydMMMMMMNdddyssssssss+
     /sssssssssssssssshhNMMyhhyyyhhNMNMNhssssss/
     .ssssssssssssssssdMMMNyssssssssssssssssss.
     +ssssshhhyNMMyssssssssssssyNMNMMyssssss+
     ossyNMNMMyMhssssssssssssssssssssssssssss.
     ssyNMNMMyMhssssssssssssssssssssssssssss.
     +ssssshhhyNMMyssssssssssssyNMNMMyssssss+
     /ssssssssssssssssdMMMNyssssssssssssssss.
     .sssssssssssssssshhNMMyhhyyyhdNMNMNhssssss/
     +sssssssssssssshydMMMMMMMdddyssssssss+
     /sssssssssssssssshdmNNNNnyNMNMMyssssss.
     .osssssssssssssssssssssdMMMNyyssso.
      -+ssssssssssssssssssssyssss+-+
       `:+ssssssssssssssssss+`:
        .-/+oossssoo+/-.
```

rog@DEP-ROG

-----

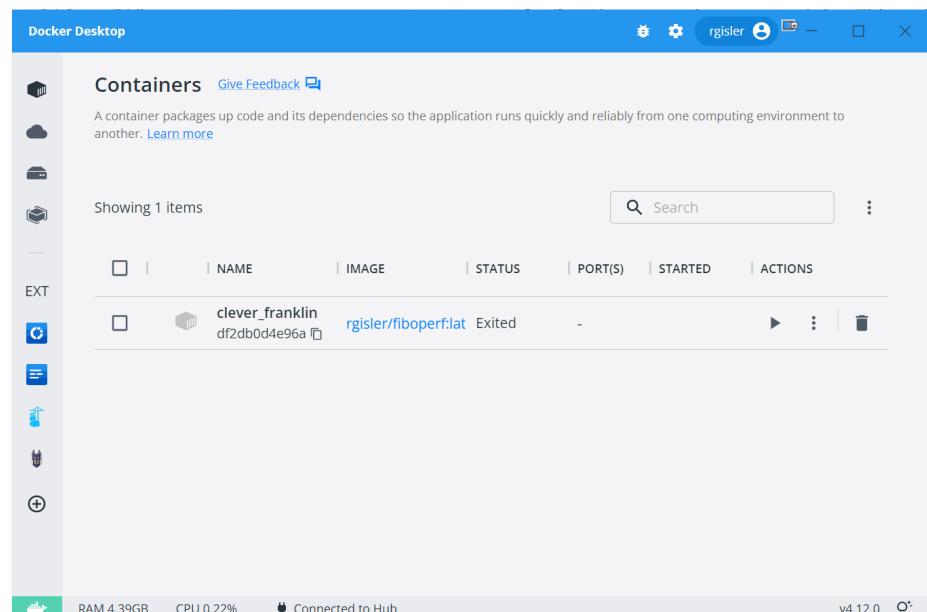
OS: Ubuntu 22.04.1 LTS on Windows 10 x86\_64  
Kernel: 5.10.102.1-microsoft-standard-WSL2  
Uptime: 1 min  
Packages: 790 (dpkg)  
Shell: bash 5.1.16  
Terminal: /dev/pts/0  
CPU: Intel i9-9880H (16) @ 2.304GHz  
GPU: 15c1:00:00.0 Microsoft Corporation Device 008e  
Memory: 543MiB / 25435MiB

# Docker auf Windows – Installation

- Erst jetzt installieren Sie **Docker Desktop** von <https://www.docker.com/> und aktivieren dabei die Option, dass Docker auf Basis von **WSL2** (nutzt Hyper-V Virtualisierung) arbeiten soll. Vorteil: Docker läuft dann im Linux Subsystem und steht auch dort zur Verfügung. So haben Sie eine vernünftige Shell zur Hand!

- **Wichtig:** Nach der Installation als erstes ein `docker login` in der Shell ausführen.

- Speichert Ihre Login-Daten verschlüsselt in Ihrem \$HOME-Verzeichnis.
- Erlaubt ihnen mehr pulls zu machen, und andere nicht zu blockieren.



# **Einsatzszenarien von Docker für/in der Entwicklung**

# Docker – Einsatzszenarien in der Entwicklung

- Spontanes, einfache ausprobieren von (Server-)Anwendungen.
  - Ohne aufwändige Installation, einfach «out-of-the-box».
- Nutzung als Buildumgebung für unterschiedliche, fremde, auf dem Host gar nicht installierte Plattformen.
  - Oder sogar für die komplette Entwicklungsumgebung (!)
- Deployment und Betrieb (eigener) produktiver Anwendungen
  - Komplett konfigurierte, lauffähige Docker-Images.
  - Verteilung und Archivierung der Images in Repositories.
- Schnelle und flexible Testumgebungen.
  - Integration in eigene Testfälle möglich (→Testcontainer)!
  - Wird im →Testing-Input noch vertieft.

# **Installation und Test**

# Aus Modul AD: Performancemessung bei Fibonacci

- Im Modul AD haben wir mal vergleichende Performancemessungen für verschiedene Implementationen zur Berechnung von Fibonacci-Zahlen gemacht.
  - Iterativ, Berechnet, Cache mit Array oder Map und Rekursiv.
- Haben Sie Lust, das mal schnell auszuprobieren? Docker installiert?

```
docker run --rm rgisler/fiboperf
```

- Hinweis: `--rm` löscht den Container nach dessen Ende sofort wieder.

```
rolan@DEP-ROG: ~$ docker run --rm rgisler/fiboperf
> 2022-10-16 15:48:44,649 INFO - == fibo(80) - 0. Messung (kalt) ==
2022-10-16 15:48:44,672 INFO - - Fibonacci f(80) = 25416728348467685 in 25ns iterativ berechnet mit 79 Additionen.
2022-10-16 15:48:44,716 INFO - - Fibonacci f(80) = 25416728348467744 in 55ns nach Blinet berechnet.
2022-10-16 15:48:45,074 INFO - - Fibonacci f(80) = 25416728348467685 in 447ns rekursiv mit Array-Cache berechnet mit 79 Additionen.
2022-10-16 15:48:45,242 INFO - - Fibonacci f(80) = 25416728348467685 in 2709ns rekursiv mit Map-Cache berechnet mit 79 Additionen.
2022-10-16 15:48:45,242 INFO - - Fibonacci f(80) = 162354155 in 346.455ms rekursiv berechnet mit 165580140 Additionen.
2022-10-16 15:48:47,598 INFO - == fibo(20) - 1. Messung (warm) ==
2022-10-16 15:48:47,591 INFO - - Fibonacci f(20) = 6765 in 10ns iterativ berechnet mit 19 Additionen.
2022-10-16 15:48:47,644 INFO - - Fibonacci f(20) = 6765 in 55ns nach Blinet berechnet.
2022-10-16 15:48:47,644 INFO - - Fibonacci f(20) = 6765 in 489ns rekursiv mit Array-Cache berechnet mit 19 Additionen.
2022-10-16 15:48:48,203 INFO - - Fibonacci f(20) = 6765 in 576ns rekursiv mit Map-Cache berechnet mit 19 Additionen.
2022-10-16 15:48:48,203 INFO - - Fibonacci f(20) = 6765 in 0.024ms rekursiv berechnet mit 18945 Additionen.
2022-10-16 15:48:48,205 INFO - == fibo(20) - 2. Messung (warm) ==
2022-10-16 15:48:48,208 INFO - - Fibonacci f(20) = 6765 in 5ns iterativ berechnet mit 19 Additionen.
2022-10-16 15:48:48,208 INFO - - Fibonacci f(20) = 6765 in 489ns nach Blinet berechnet.
2022-10-16 15:48:48,340 INFO - - Fibonacci f(20) = 6765 in 210ns rekursiv mit Array-Cache berechnet mit 19 Additionen.
2022-10-16 15:48:48,801 INFO - - Fibonacci f(20) = 6765 in 569ns rekursiv mit Map-Cache berechnet mit 19 Additionen.
2022-10-16 15:48:48,802 INFO - - Fibonacci f(20) = 6765 in 0.032ms rekursiv berechnet mit 18945 Additionen.
2022-10-16 15:48:48,802 INFO - == fibo(0) - 1. Messung (warm) ==
2022-10-16 15:48:48,802 INFO - - Fibonacci f(0) = 162354155 in 9ns iterativ berechnet mit 39 Additionen.
2022-10-16 15:48:48,853 INFO - - Fibonacci f(0) = 162354155 in 50ns nach Blinet berechnet.
2022-10-16 15:48:49,027 INFO - - Fibonacci f(0) = 162354155 in 217ns rekursiv mit Array-Cache berechnet mit 39 Additionen.
2022-10-16 15:48:49,949 INFO - - Fibonacci f(0) = 162354155 in 115ns rekursiv mit Map-Cache berechnet mit 39 Additionen.
2022-10-16 15:48:58,326 INFO - == fibo(40) - 2. Messung (warm) ==
2022-10-16 15:48:58,326 INFO - - Fibonacci f(40) = 162354155 in 377.33618ms rekursiv berechnet mit 165580140 Additionen.
2022-10-16 15:48:58,326 INFO - - Fibonacci f(40) = 162354155 in 9ns iterativ berechnet mit 39 Additionen.
2022-10-16 15:48:58,376 INFO - - Fibonacci f(40) = 162354155 in 51ns nach Blinet berechnet.
2022-10-16 15:48:58,376 INFO - - Fibonacci f(40) = 162354155 in 217ns rekursiv mit Array-Cache berechnet mit 39 Additionen.
2022-10-16 15:48:58,376 INFO - - Fibonacci f(40) = 162354155 in 1159ns rekursiv mit Map-Cache berechnet mit 39 Additionen.
2022-10-16 15:48:58,537 INFO - == fibo(0) - 2. Messung (warm) ==
2022-10-16 15:48:58,537 INFO - - Fibonacci f(0) = 162354155 in 201ns rekursiv mit Array-Cache berechnet mit 39 Additionen.
2022-10-16 15:48:58,537 INFO - - Fibonacci f(0) = 162354155 in 1195ns rekursiv mit Map-Cache berechnet mit 39 Additionen.
2022-10-16 15:48:58,537 INFO - - Fibonacci f(0) = 162354155 in 340.3913ms rekursiv berechnet mit 165580140 Additionen.
2022-10-16 15:48:51,834 INFO - == fibo(80) - 1. Messung (warm) ==
2022-10-16 15:48:51,851 INFO - - Fibonacci f(80) = 25416728348467685 in 21ns iterativ berechnet mit 79 Additionen.
2022-10-16 15:48:51,894 INFO - - Fibonacci f(80) = 25416728348467744 in 51ns nach Blinet berechnet.
2022-10-16 15:48:52,228 INFO - - Fibonacci f(80) = 25416728348467685 in 416ns rekursiv mit Array-Cache berechnet mit 79 Additionen.
2022-10-16 15:48:52,228 INFO - - Fibonacci f(80) = 25416728348467685 in 2479ns rekursiv mit Map-Cache berechnet mit 79 Additionen.
2022-10-16 15:48:54,211 WARN - - NOPE: Rekursive Fibonacci ist fuer n=80 viel zu langsam.
2022-10-16 15:48:54,212 INFO - == fibo(80) - 2. Messung (warm) ==
2022-10-16 15:48:54,228 INFO - - Fibonacci f(80) = 25416728348467744 in 408ns iterativ berechnet mit 79 Additionen.
2022-10-16 15:48:54,228 INFO - - Fibonacci f(80) = 25416728348467685 in 289ns rekursiv mit Array-Cache berechnet mit 79 Additionen.
2022-10-16 15:48:54,228 INFO - - Fibonacci f(80) = 25416728348467685 in 289ns rekursiv mit Map-Cache berechnet mit 79 Additionen.
2022-10-16 15:48:56,451 WARN - - NOPE: Rekursive Fibonacci ist fuer n=80 viel zu langsam.
2022-10-16 15:48:56,651 INFO - - .
```

# **Docker Images als Build-Umgebung**

# Docker Images als Buildumgebung

- Haben Sie auf jedem Rechner immer das gewünschte JDK in der richtigen Version und auch das richtige Buildtool verfügbar?
  - Dafür ist Docker extrem praktisch!

- Viele Projekte bieten für Ihre Produkte fertige Images an. Seien Sie aber vorsichtig, achten Sie auf seriöse Quellen!

Auf Docker Hub:  DOCKER OFFICIAL IMAGE

- Beispiel: Apache bietet verschiedene Versionen von Maven, und diese auch auf unterschiedlichen (!) JDK's an.
- Für uns sind folgende Tags interessant:
  - **3.8.6-eclipse-temurin-17**
  - **3.8.6-amazoncorretto-17**
  - **3.8.6-openjdk-18** (kein LTS!)

## Supported tags and respective Dockerfile links

- 3.8.6-openjdk-18, 3.8-openjdk-18, 3-openjdk-18
- 3.8.6-openjdk-18-slim, 3.8-openjdk-18-slim, 3-openjdk-18-slim
- 3.8.6-eclipse-temurin-11, 3.8-eclipse-temurin-11, 3-eclipse-temurin-11
- 3.8.6-eclipse-temurin-11-alpine, 3.8-eclipse-temurin-11-alpine, 3-eclipse-temurin-11-alpine
- 3.8.6-eclipse-temurin-11-focal, 3.8-eclipse-temurin-11-focal, 3-eclipse-temurin-11-focal
- 3.8.6-eclipse-temurin-17, 3.8.6, 3.8.6-eclipse-temurin, 3.8-eclipse-temurin-17, 3.8, 3.8-eclipse-temurin, 3-eclipse-temurin-17, 3, latest, 3-eclipse-temurin, eclipse-temurin
- 3.8.6-eclipse-temurin-17-alpine, 3.8-eclipse-temurin-17-alpine, 3-eclipse-temurin-17-alpine
- 3.8.6-eclipse-temurin-17-focal, 3.8-eclipse-temurin-17-focal, 3-eclipse-temurin-17-focal
- 3.8.6-eclipse-temurin-19, 3.8-eclipse-temurin-19, 3-eclipse-temurin-19
- 3.8.6-eclipse-temurin-19-alpine, 3.8-eclipse-temurin-19-alpine, 3-eclipse-temurin-19-alpine
- 3.8.6-eclipse-temurin-19-focal, 3.8-eclipse-temurin-19-focal, 3-eclipse-temurin-19-focal
- 3.8.6-eclipse-temurin-8, 3.8-eclipse-temurin-8, 3-eclipse-temurin-8
- 3.8.6-eclipse-temurin-8-alpine, 3.8-eclipse-temurin-8-alpine, 3-eclipse-temurin-8-alpine
- 3.8.6-eclipse-temurin-8-focal, 3.8-eclipse-temurin-8-focal, 3-eclipse-temurin-8-focal
- 3.8.6-ibmjava-8, 3.8.6-ibmjava, 3.8-ibmjava-8, 3-ibmjava-8, 3-ibmjava, ibmjava
- 3.8.6-ibm-semeru-11-focal, 3.8-ibm-semeru-11-focal, 3-ibm-semeru-11-focal
- 3.8.6-ibm-semeru-17-focal, 3.8-ibm-semeru-17-focal, 3-ibm-semeru-17-focal
- 3.8.6-amazoncorretto-11, 3.8.6-amazoncorretto, 3.8-amazoncorretto-11, 3.8-amazoncorretto, 3-amazoncorretto-11, 3-amazoncorretto, amazoncorretto
- 3.8.6-amazoncorretto-17, 3.8-amazoncorretto-17, 3-amazoncorretto-17
- 3.8.6-amazoncorretto-18, 3.8-amazoncorretto-18, 3-amazoncorretto-18
- 3.8.6-amazoncorretto-19, 3.8-amazoncorretto-19, 3-amazoncorretto-19
- 3.8.6-amazoncorretto-8, 3.8-amazoncorretto-8, 3-amazoncorretto-8
- 3.8.6-sapmachine-11, 3.8-sapmachine-11, 3-sapmachine-11
- 3.8.6-sapmachine-17, 3.8.6-sapmachine, 3.8-sapmachine-17, 3.8-sapmachine, 3-sapmachine-17, 3-sapmachine, sapmachine

# Beispiel: JDK 17 «Eclipse Temurin» mit Maven 3.8.6

- Starten Sie doch einfach mal das folgende Image:

```
docker pull maven:3.8.6-eclipse-temurin-17
```

```
docker run -it --rm maven:3.8.6-eclipse-temurin-17 bash
```

- Prüfen Sie mit **mvn -version** was wirklich drauf ist! ☺

- Wir könnten nun mit **git** ein Projekt clonen und bauen. Aber es geht noch einfacher: Wir können auch Teile unseres Host-Dateisystems und ein →**Volume** in den Container mounten!
- Somit kann der Container transparent auf Bereiche unseres Host-Dateisystems zugreifen, und das Maven-Repository zur Wiederverwendung persistieren.
  - Das Volume ist für das Maven-Repository, und kann so in verschiedenen Containern wiederverwendet werden!  
(als Beispiel, nur ein möglicher Ansatz)

# Build auf JDK 17 «Eclipse Temurin» mit Maven 3.8.6

- Hinweis: Pfade an Ihre eigene Umgebung anpassen!
- Beispiel für ein Volume (Erstellung, einmalig):

```
docker volume create "maven-repo"
```

- Container mit gemountetem Verzeichnis und Volume starten:

```
docker run -it --rm
```

```
  -v "C:\path\project\:/work"
```

```
  -v "maven-repo:/root/.m2"
```

```
maven:3.8.6-amazoncorretto-17 bash
```

- Im Container einfach ins Verzeichnis /work wechseln (`cd work`) und den Maven-Build starten: `mvn package`
  - Hinweis: Beim ersten Mal ist das Repo natürlich noch leer.
- **Achtung:** Nicht vergessen, den Container mit `exit` zu verlassen.

# Docker Image als Grundlage für Buildprozess

- Ist im Arbeitsalltag eine enorme Erleichterung!
  - Wir müssen nicht (mehr) verschiedene Java-Laufzeitumgebungen installieren (und umschalten). Deutlich weniger Installationen.
  - Wir können relativ einfach auf/mit/für die effektive Zielumgebung bauen und natürlich auch **testen!** Isolation!
  - Mit ausgewählten IDE's (IntelliJ und Visual Studio Code) ist es sogar möglich, sich mit einem entsprechenden (laufenden) Container zu verbinden, und direkt darin zu entwickeln!
    - Für diese Fälle lohnt es sich, nach eigenen Bedürfnissen erstellte und konfigurierte Images als Basis zu verwenden!
- ➔ Potential: Standardisierte Entwicklungs(-basis)-Umgebungen für alle Entwickler\*innen in einem Team bzw. Organisation!

# **Erstellen eigener Docker-Images**

# Wie werden Images erstellt?

- Images werden in →Layers erstellt, wobei jeder Layer eine Anzahl Dateien ergänzt oder Befehle ausführt (die Dateien verändern). Images basieren typisch auf bereits existierenden Images!
- Die einzelnen Schritte werden über Befehle in einem →**Dockerfile** beschrieben (Text-Datei mit dem Namen **Dockerfile**), das ist quasi der «Quellcode» für die Image-Erstellung.
  - Images werden somit reproduzierbar erstellt, vergleichbar mit einem Buildprozess – **Infrastructure as code** (IaC).
- Images werden typisch in einer Registry abgelegt, wo sie zur Verwendung zur Verfügung gestellt werden (Deployment).
  - Default: Docker Hub (<https://hub.docker.com/search?q=>)
- Die Dockerfiles (und alle weiteren Quellartefakte) stellt man typisch unter Versionskontrolle (VCS).

# Beispiel: Ein (sehr) einfaches Dockerfile

- Das folgende Beispiel steht Ihnen im bekannten Projekttemplate «oop\_maven\_template» (seit Version 3.2.0) zur Verfügung!
- Inhalt des **Dockerfile**:

```
FROM amazoncorretto:17.0.4-alpine
COPY target/oop_maven_template.jar /opt/demo/oop_maven_template.jar
CMD ["java", "-jar", "/opt/demo/oop_maven_template.jar"]
```

- Erklärung der einzelnen Zeilen:
  - **FROM**: Das Image basiert auf einem Alpine Linux (klein) mit einer JRE-Distribution «Amazon Corretto», LTS-Version 17.0.4.
  - **COPY**: Aus dem **target**-Verzeichnis des Projektes wird das direkt ausführbare JAR-Datei (→ Shade-JAR mit Manifest, hier eine wichtige Voraussetzung!) in das Image kopiert.
  - **CMD**: Der Startbefehl des Images (für **run**) wird hinterlegt.

# Beispiel: Bau und Push eines Images

- Befehl zum Bauen (mit docker):

```
docker build . -t "rgisler/oop-demo:latest"
```

- Das Image steht danach im lokalen Repository zur Verfügung:

- **rgisler** – Namespace des Images.

- **oop-demo** – Name des Images.

- **latest** – Neuste Version (vgl. SNAPSHOT in Apache Maven)

- Vorausgesetzt Sie haben ein Docker-Login (sollten Sie!) kann das Image danach direkt in das Docker-Repository gepushed werden:

```
docker push "rgisler/oop-demo:latest"
```

- Wird dann sichtbar auf <https://hub.docker.com/repositories>



# Bau von Images - Herausforderungen

- Die Images müssen natürlich der jeweiligen Plattform und Distribution entsprechend konfiguriert werden. Das setzt relativ gute Kenntnisse des Zielsystems voraus (oder viel try&error ☹).
- Man kann viel gutes (aber auch schlechtes) von bestehenden Images (bzw. deren Dockerfiles) abgucken.
- Die Referenz für die Dockerfile-Syntax:  
<https://docs.docker.com/engine/reference/builder/>
- Es gibt mittlerweile gute Syntax-Checker/Editoren für Dockerfiles.
- Man sollte darauf achten, «sinnvolle» Layers zu bilden.
  - So wenige wie möglich, so viele wie nötig.
  - Möglichst ähnliche Änderungshäufigkeit (pro Layer).
  - Möglichst kompakte Images (möglichst klein).

# Komplexeres Beispiel: Dockerfile von maven

- Siehe <https://github.com/carlossg/docker-maven/blob/ac292f26884bf2be9fe69f6e397da3b124c1e35c/eclipse-temurin-17/Dockerfile>
- Interessante Details:
  - **apt update** und **install** in einem Kommando, inkl. wieder löschen der Update-Listen. Alles in einem Layer, Platz gespart.
  - Installation von Maven in einem einzigen Kommando, inkl. Download mit sha-Check, dann aufräumen und Symlink setzen. Auch hier: Nur ein Layer erstellt, Platz gespart.
- Beim Build eines Containers werden die Dateiinhalte und Kommandos «gehashed». Ein Layer wird nur neu erstellt, wenn der Hash ändert → schnellerer Build des Images, weniger Bandbreite.
  - Problem: **apt install** – Befehl (Beispiel) unverändert, aber...?

# **Docker im Java Umfeld**

# Dockerisierung von Java-Anwendungen

- Java-Applikationen brauchen im Regelfall eine JRE (kostet Platz).
- Java-Applikationen können sehr unterschiedlich deployed werden:
  - Single-JAR – alles in einem einzigen JAR verpackt.
  - App-JAR und Dependency-JAR's: **classpath** notwendig, typisch über ein Startscript (**shell**) gelöst.
  - Alles in einem, oder in getrennten Layern? (Vor- und Nachteile)
  - Modularisierte Applikation (JPMS, **Java Plattform Modul System**).
- Gewähltes Szenario ist individuell abzuklären, es gibt keine Empfehlung für eine Standard-Variante, sondern nur Meinungen. ☺
- Positiv: Es gibt mehrere Maven-Plugins, welche verschiedene Techniken unterstützen und automatisieren, und sich sogar kombinieren lassen.

# Maven Plugins für Docker

- **Fabric8 Maven Docker Plugin** (Standard)

- Setzt ein Dockerfile voraus. Alle Möglichkeiten offen.
- Konfiguration teilweise über das Maven POM.
  - Minimal, z.B. für lokalen Start über Maven.
- Projekt und Dokumentation:

<https://github.com/fabric8io/docker-maven-plugin>

- **Google Jib Docker Plugin** (Alternative)

- Sehr hohe Automatisierung.
- Konfiguration ausschliesslich über das Maven POM.
- Image-Build und Deploy auch ohne (!) Docker-Runtime möglich.
- Projekt und Dokumentation:

<https://github.com/GoogleContainerTools/jib/tree/master/jib-maven-plugin>

# Einfaches Beispiel: Java EchoServer in Docker

- Projekt steht auf ILIAS als ZIP-Datei zur Verfügung.
- **Wichtig:** Vor Versuchen im POM den Benutzer (`docker.user`) anpassen und Plugin-Konfigurationen (Fabric8 und Jib) ansehen.
- **EchoServer** (auf Port 5555/tcp) - kann gestartet werden:
  - Direkt in der IDE.
  - Binär als Single-JAR packetiert (mvn package) aus /target
  - Verpackt in einer Container (als Single-JAR) mit **Fabric8**-Plugin.
  - Verpackt als Layered-Java App mit **Jib**-Plugin.
- **EchoClient** – ist die passende Client-Applikation:
  - Direkt in der IDE ausführen. Default auf **localhost:5555**
  - Eingabe «quit» beendet nur den Client, Server läuft weiter.
  - Eingabe «exit» sendet Kill-Message, beendet Client & Server.

# Bau und Start/Stop des EchoServer Beispiels (Fabric8)

- Das Projekt kann mit Apache Maven über `mvn package` normal gebaut werden. Es wird über das shade-Plugin ein «Singel-JAR» erzeugt, dass sämtliche Dependencies integriert.
- JAR kann mit `java -jar vsk-echoserver.jar` gestartet werden, der Server bindet sich an sämtliche Interfaces auf Port 5555.
  - Empfehlung: Gleich mit Client testen!
- Bauen des Docker Images mit Maven: `mvn docker:build`
- Starten des Images mit Maven: `mvn docker:start`
  - Anzeigen des Logs über Docker Desktop, oder in Shell über `mvn docker:logs` oder mit `docker logs <hash> -f`
- Stoppen des Images mit Maven: `mvn docker:stop`
  - Löscht den Container automatisch auch gleich!

# Netzwerk mit Docker

- Für die Netzwerk-Kommunikation mit einem Container, müssen die dafür genutzten Ports auf Ports des Host-Systems gemappt werden.
  - Analog zu den Dateisystemen (mit `-v "host:container"`)
  - Im EchoServer-Beispiel ist das für die Maven-Plugins bereits in der Konfiguration im POM hinterlegt.

- Dafür gibt es ein weiteres Argument: `-p "hostport:cont.port"`
- Wenn wir das EchoServer-Beispiel mit Docker starten wollen:

```
docker run -d -p "5555:5555" rgisler/echoserver
```

- Fehlerquelle: Vergisst man das Portmapping startet der Container trotzdem fehlerfrei, aber Kommunikation klappt nicht
  - Praktisch: Mapping auf andere Ports möglich (z.B. `4444:5555`)
- Tipp: Das gute alte `telnet` ist hier praktisch. ☺

# Bau und des EchoServer Beispiels mit Jib

- Das Images kann wahlweise mit Fabric8 (`docker:help`) oder mit Jib (`jib:help` gibt es leider **nicht**) gebaut werden.
- Jib baut mit dem Kommando `mvn jib:build` das Image nicht nur ohne (!) die lokale Docker-Instanz (muss nicht laufen!), sondern pusht es auch direkt in die Registry hoch.
  - Dabei wird die Java-Applikation bzw. das POM analysiert und die Dependencies (und Konfig) von der eigentlichen Applikation (classes und Konfig) in getrennten Layers im Image abgelegt.
  - Das Reduziert den Pull-Aufwand bei Applikationen mit grossen Dependencies markant (weil diese typisch viel seltener ändern).
- Jib kann aber auch rein lokal bauen (dann aber **mit** Docker):  
`mvn jib:dockerBuild`

## **Ein Ausblick: Testcontainer mit Docker!**

# Docker für automatisierte Integrationstests

- Docker Container sind dafür ausgelegt, dass man sie sehr schnell hoch- und runterfahren kann. Das wäre doch perfekt für (Integrations-)Tests!

- **YEAH! Das gibt es:**

<https://www.testcontainers.org/>



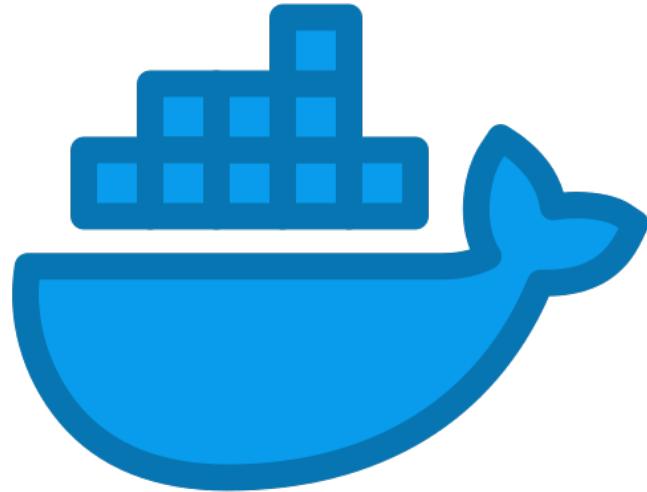
- Eine kleine Demo hab ich noch...😊
- Stichworte: Automatischen hoch-/runterfahren von generischen oder vorbereiteten Containern. Zugriff auf Logs. Dynamische, kollisionsfreie Vergabe von Ports. Automatisches Aufräumen.

# Noch ein paar Tool-Tipps zum Schluss

- **cTop** - <https://github.com/bcicen/ctop>
  - Shell-Tool für die permanente Anzeige aller Container.
  - Man kann auch Container inspizieren.
- **dive** - <https://github.com/wagoodman/dive>
  - Shell-Tool für die Analyse von Images.
  - Für die Fehlersuche praktisch.
- Tipps:
  - Lassen Sie **ctop** einfach in einer Shell laufen, dann sehen Sie immer kompakt was auf Ihrem System grad so los ist.
  - Untersuchen Sie mit **dive** die von den Fabric8- und Jib-Plugins unterschiedlich aufgebauten Images!
- Es gibt noch viel mehr! Weitere Tipps werden im Forum gerne aufgenommen!

# Zusammenfassung

- Docker für leichtgewichtige Virtualisierung auf verschiedenen Plattformen.
- Für einfaches Deployment und Betrieb von Applikationen (typisch headless, Server-Applikationen).
- Aber auch für die Entwicklung(-sumgebung) sehr interessant.
- Sie brauchen definitiv auch Linux/Unix-Kenntnisse!
- Sehr hoher Automatisierungsgrad, Basis für CD (Continuous Deployment).
- Umfangreiches Tooling, grosse Popularität.



**Zeit für eigene Versuche!**

**Fragen?**

Verteilte Systeme und Komponenten

# Integrations- und Systemtest

Martin Bättig



# Inhalt

- Einleitung und Überblick
- Teststrategie
- Integrationstest
- Systemtest
- Testing in Scrum

# Lernziele

- Sie können Komponenten entwerfen, dokumentieren, in Java realisieren, **testen** und deployen.
- Sie kennen die Zusammenhänge zwischen Analyse/Design und **Test/Abnahme** von Softwarekomponenten.
- Sie können geeignete Systemtests definieren, diese dokumentieren und die Durchführung protokollieren.
- Sie wissen, welche Informationen über die zu entwickelnde Software wann, wie und wo dokumentiert werden sollen.
- Sie wissen, welche Informationen aus dem Entwicklungsprozess gemäss Scrum wann, wie und wo dokumentiert werden sollen.
- Sie können Sprintbacklogs für ein kleines Team formulieren, schätzen und **geeignete Abnahmekriterien** festlegen.

# **Einleitung und Überblick**

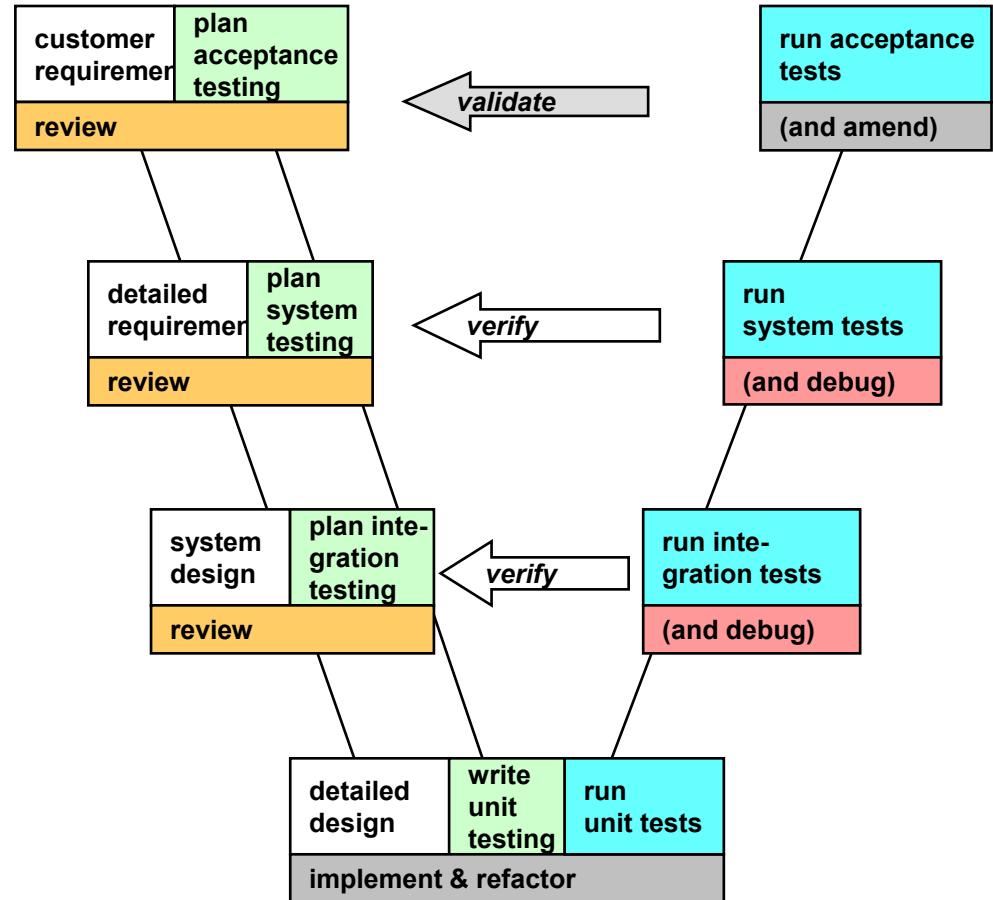
# Test Design

- "Testing by poking around is a waste of time" –  
(Zitat aus *Testing Object-Oriented Systems* von Robert Binder).
- Nur dokumentierte oder automatisierte Tests lassen sich **wiederholen** (=> Regressionstests!).

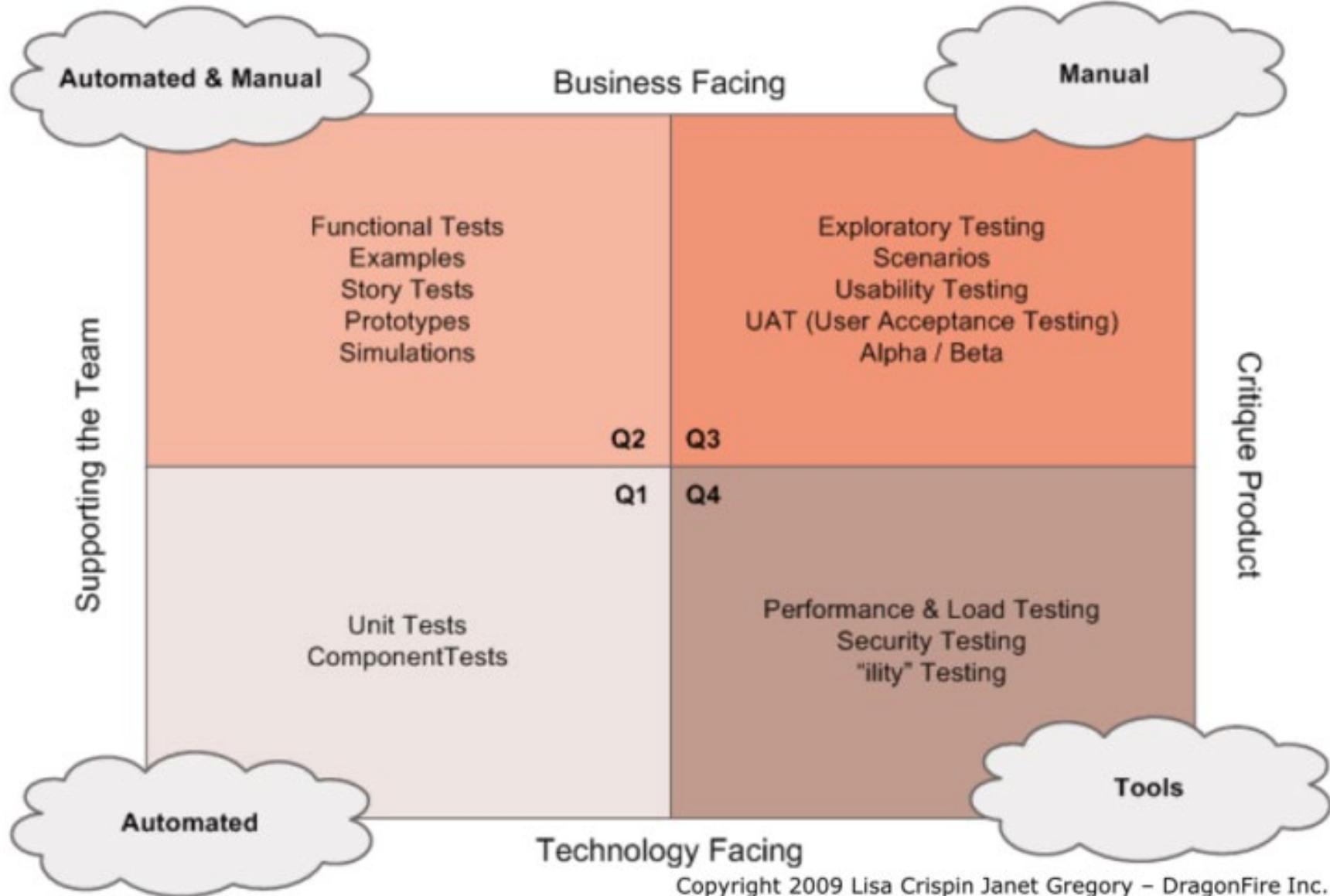


# Klassisches Testen mit dem V-Modell

- Zu jeder Disziplin gibt es eine Entsprechung auf der Test-Seite.
- Anforderungen und Spezifikationen sind Grundlage für weitere Arbeiten => Review!
- Validieren (haben wir das Richtige entwickelt?) **und** verifizieren (haben wir es richtig gemacht?)



# Agiles Testen



# Ein Test genügt nicht

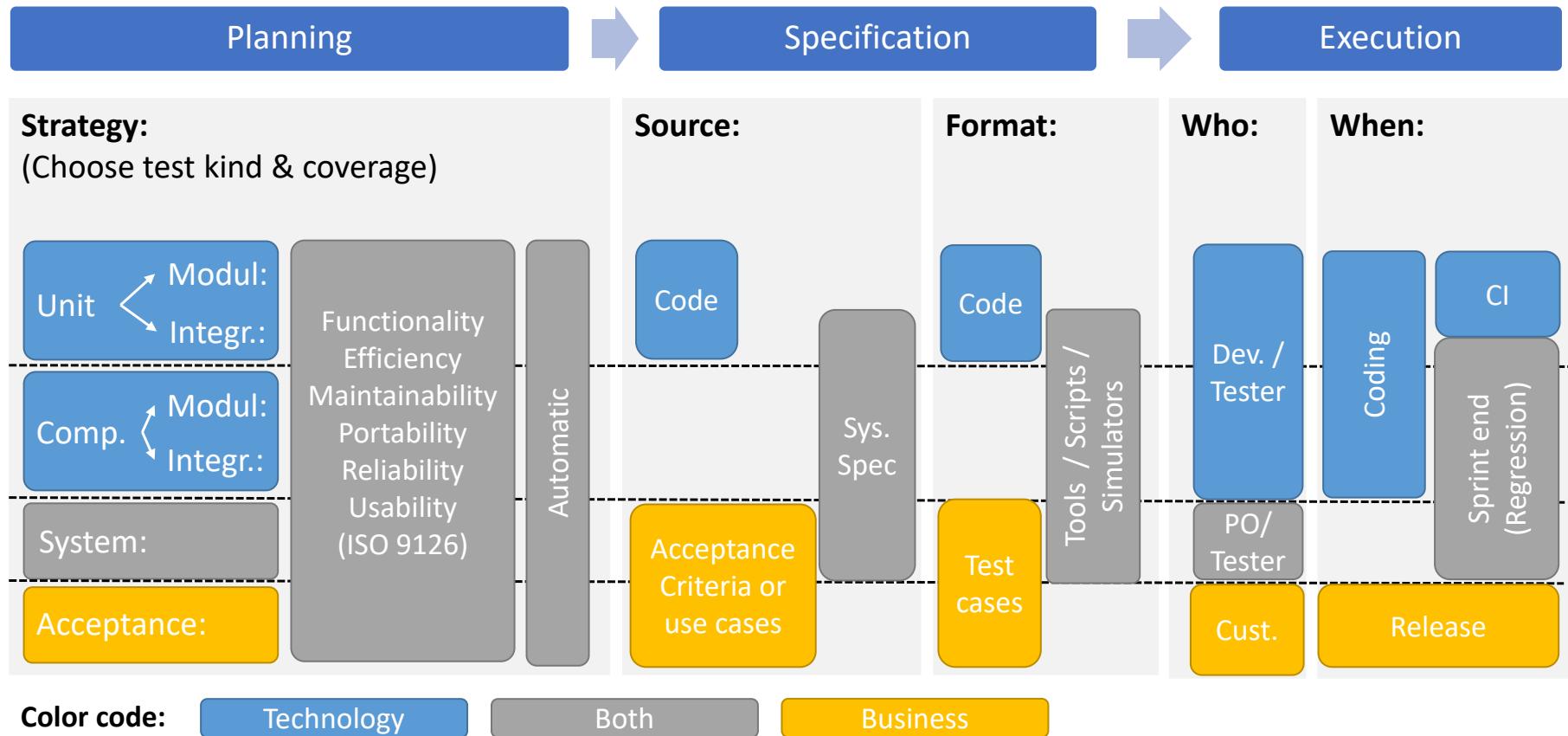
- Je nach Statistik findet man mit jeder Teststart und jedem Review etwa 33% bis 67% der Fehler, welche in einer Software stecken.
- Für ein reales Beispiel (Betriebssystem mit 40 Mio. LOC) und einer durchschnittlichen Testausbeute von 50% pro Teststart, würde dies so aussehen:

Test	Bugs found	Bugs remaining
-	-	2'000'000
Code reviews	1'000'000	1'000'000
Unit test	500'000	500'000
Integration test	250'000	250'000
System test	125'000	125'000

# Alles herausholen!

- Mit keiner Testart und mit keinem Review findet man alle Fehler.
- Nur im Zusammenwirken der unterschiedlichen Techniken findet man ein Maximum an Fehlern.
- Vollständiges Testen ist schlicht nicht machbar.
- Es stellt sich deshalb die Frage: Welche Testarten und welche Tests haben die besten Chancen ein Maximum an Fehlern mit einem Minimum an Kosten aufzudecken? => **Teststrategie**.

# Testing im Überblick



# **Teststrategie**

# Festlegung der Teststrategie

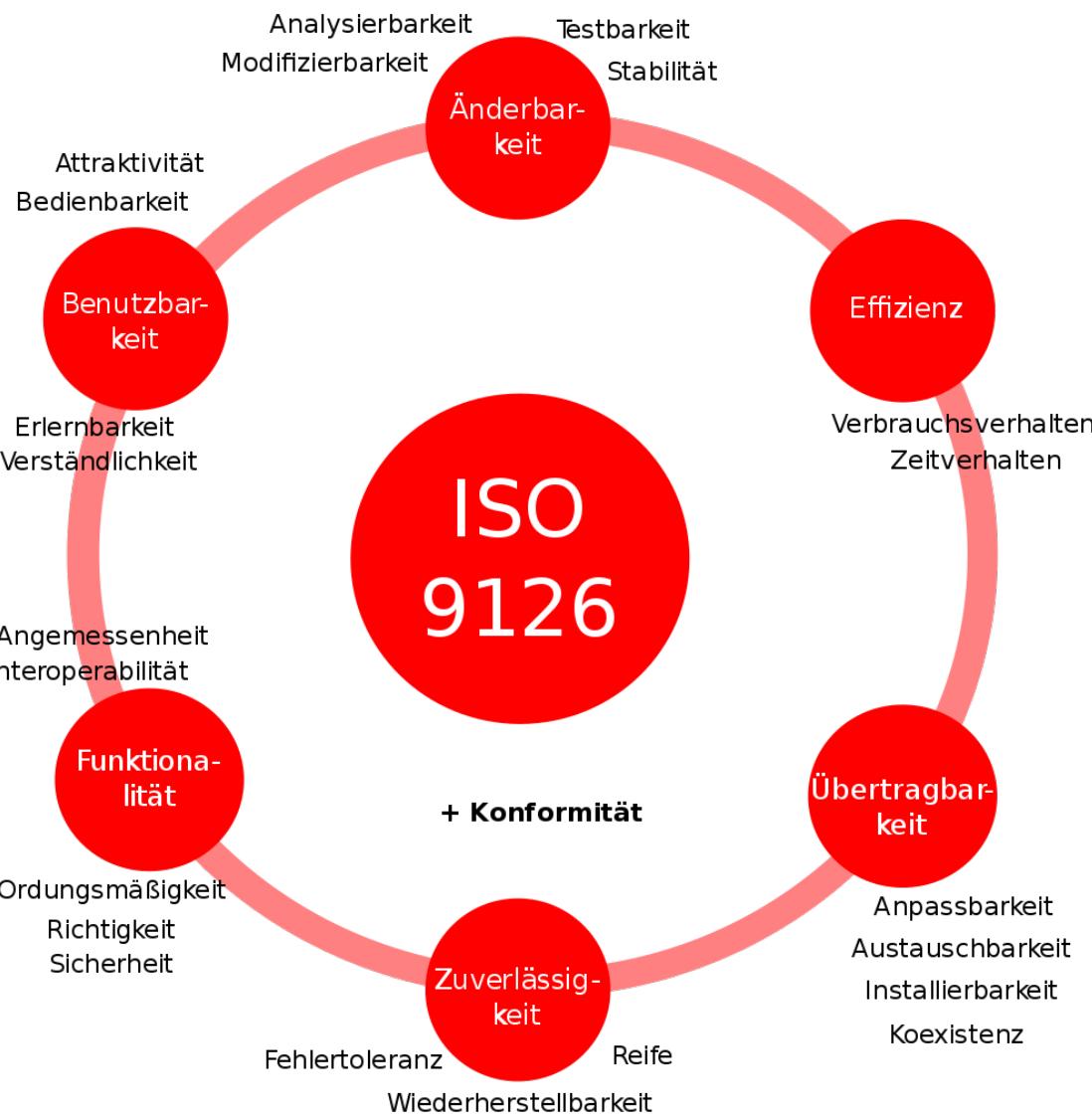
Eigenschaften einer soliden Teststrategie:

- **Fachgerecht:** Passende Testziele, Testmethoden, Testarten.
- **Risikoorientiert:** Nicht alle Systeme bzw. Systemkomponenten sind gleich kritisch. Abdeckung entsprechend anpassen.
- **Wirtschaftlich:** In der Regel beschränktes Testbudget.

unter Berücksichtigung mehrerer Dimensionen:

- Qualitätskriterien des Produkts.
- Modul-Hierarchieebenen.
- Automatisierung.

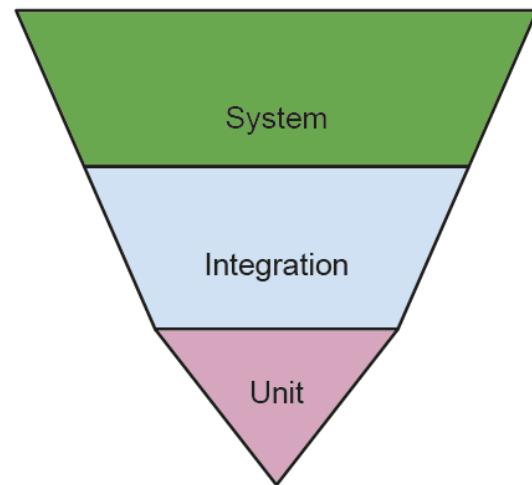
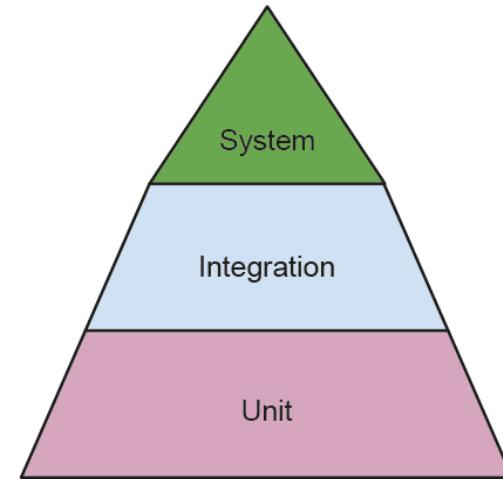
# Qualitätskriterien des Produkts



Teilweise unterschiedliche Kriterien pro Modul.

# Modul-Hierarchieebenen

- Klassische agile Testpyramide:
  - Idee: Integrations- und Systemtests sind aufwändig, da schwer automatisierbar.
  - Unittests stellen Basis da.
- Alternative Testpyramide:
  - Idee: Mittels Tools und Virtualisierung sind Integrations- und Systemtests gut automatisierbar.
  - Unitest nur bei besonders kritischen Funktionalitäten.



Quelle: Modulare Softwarearchitektur, Herbert Dowalil

# Automatisierung

- Was lässt sich automatisieren?
- Steht der Nutzen im Verhältnis zum Aufwand?



# Übung: Teststrategie für ein Doodle-Klon (Funktionsweise)

## Erstellen eines Accounts:

New Account Creation

Please complete the form below to create an account for the GroupVote application.

After account creation, an email will be sent to the address specified in the form. This email contains an account verification link that you must follow to complete account creation.

First Name: Martin  
Last Name: Bättig  
Email: martin.baettig@hslu.ch  
Password: \*\*\*\*

[Create Account](#)



Account Verification

Please check your email account for an email from GroupVote. This email contains the validation code needed to finish the account creation. Enter this code here, or follow the link in your email.

Email: bla@bla  
Validation Code:

[Validate Account](#)

## Erstellen einer Umfrage mit Einladung per Email:

Login

Email:   
Password:

[Login](#)  
No account? [Register a new account.](#)

Your Votes

[Create new Vote](#)

QM Workshop im April  
Projekt Kick-Off

Edit Show Delete  
Edit Show Delete

Create Poll

Poll Title: QM Meeting

Options:

Option 1: Monday, 28.03.2022 Remove  
Option 2: Tuesday, 29.03.2022 Remove

Add

Participants:

Participant 1: fritz.mueller@hslu.ch Remove  
Participant 2: verni.meier@gmail.com Remove

Add

[Create Poll and Send Invitations](#)



## Teilnahme an Umfrage



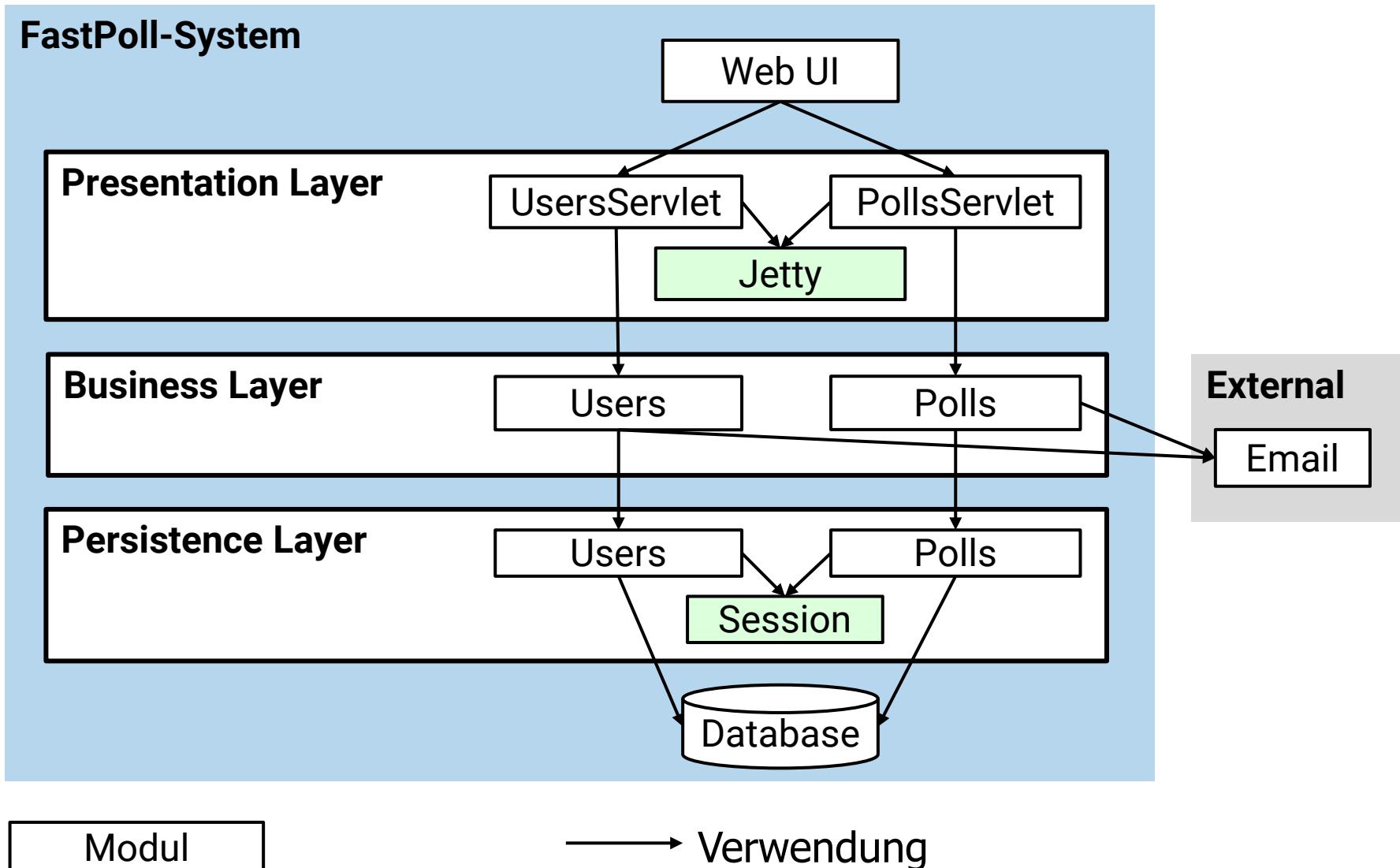
Participate in Poll

Poll Title: QM Workshop in April

	Freitag, 01.04.2022	Montag, 04.04.2022	Dienstag, 05.04.2022	Mittwoch, 06.04.2022
Fritz	yes	yes	no	yes
Verni	yes	no	no	yes
Heinz	<input checked="" type="checkbox"/>	<input type="checkbox"/>	<input checked="" type="checkbox"/>	<input type="checkbox"/>

[Send](#)

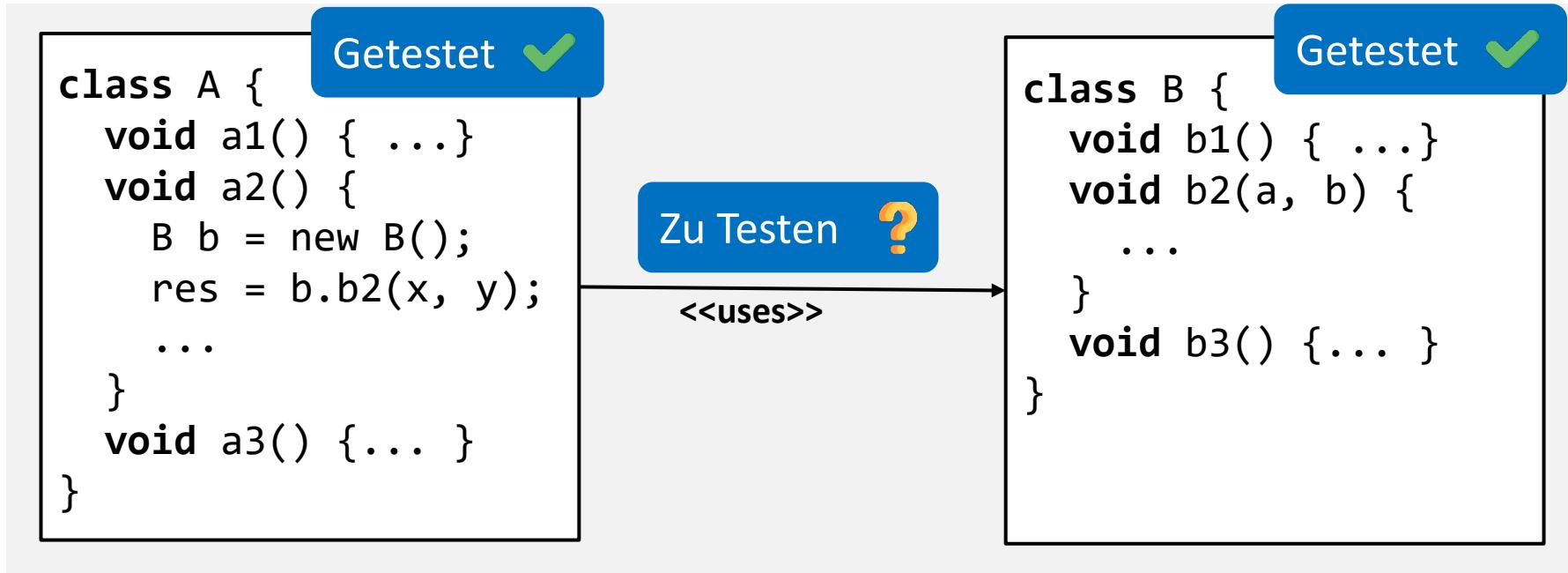
# Übung: Teststrategie für ein Doodle-Klon (Architektur)



# **Integrationstest**

# Integrationstest

Prüfe die Schnittstellen und das Zusammenspiel der Systemkomponenten (Unit, Komponente, etc.):



**Vorbedingung:** Alle zu integrierenden Module sind bereits (soweit möglich) erfolgreich getestet.

# Integrationstests testen folgendes:

- **Schnittstellen (direkte Abhängigkeiten)**
  - Objekt-Kompatibilität (Typen und Wertebereiche).
  - Aufruf-Sequenzen.
  - Wer validiert Inputs – der Aufrufende oder der Aufgerufene.
- **Datenabhängigkeiten (indirekte Abhängigkeiten)**
  - Für jede Komponente ermitteln, welche Datenabhängigkeiten bestehen und diese testen.
  - Gemeinsam genutzte Ressourcen (z.B. Dateien, Shared-Memory, Datenbanktabellen).
- **CallGraph-Abdeckung**
  - Bei Komponenten, die von verschiedenen Aufrufern genutzt werden, auch alle Aufrufvarianten testen.

# Schwierigkeiten: Gemeinsam genutzte Ressourcen

- **Reentrance:** Eine Komponente wird mehrfach gleichzeitig ausgeführt.
- **Interrupts:** Eine Komponente A wird von einer Komponente B unterbrochen. Komponente B führt auf einer gemeinsam genutzten Ressource eine Schreiboperation aus. Anschliessend muss Komponente A wieder das korrekte Ergebnis zurückliefern.
- **Race Conditions:** Zwei Komponenten greifen (concurrent) ohne ausreichende Synchronisierung auf die gleiche Ressource zu.

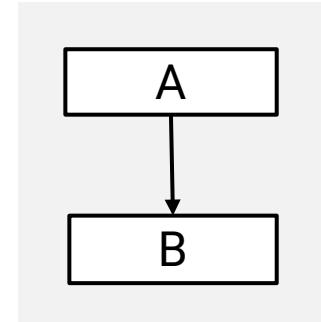
Code, der im statischen Testumfeld erfolgreich lief, kann in dynamischer Umgebung immer noch Fehler auslösen!

# Stellvertreter

- Um ein Testobjekt zu testen, müssten alle Komponenten, die das Testobjekt benötigt, in der Testumgebung installiert und im Testlauf mit verwendet werden können.
- Bei inkrementellen Entwicklung ist es eher die Regel als die Ausnahme, dass nicht alle benötigten Komponenten fertig bzw. vorhanden sind.
- **Möglichkeit fehlende Komponenten durch Stellvertreter (z.B. Mock-Objekte oder Simulatoren) zu ersetzen.**
- Der Einsatz von Stellvertretern erlaubt auch **selektivere Tests** zu erstellen.

# Integrationstestfälle entwerfen: Step-By-Step

1. Wechselwirkung analysieren: Welche Operationen von B ruft A auf? Diese Aufrufe sind zu testen.
2. Parametersätze der Aufrufe von B ermitteln (ggf. Äquivalenzklassenanalyse).
3. Testfallinput ermitteln: Welche Aufrufsequenzen von A sind notwendig um im Aufruf von B die in Schritt 2 ermittelten Parametersätze auszulösen?
4. Soll-Ist-Vergleich: Wie kann die Sollreaktion von B ermittelt werden? Direkt aus dem Output von A? Oder muss der Übertragungsweg mitgeschnitten werden?
5. Bei asynchroner Kommunikation: Zusätzlich Timing, Durchsatz, Kapazität, Performance sowie Datenübertragungsfehler prüfen.



# Fallbeispiel: Integrationstest für Accounterstellung

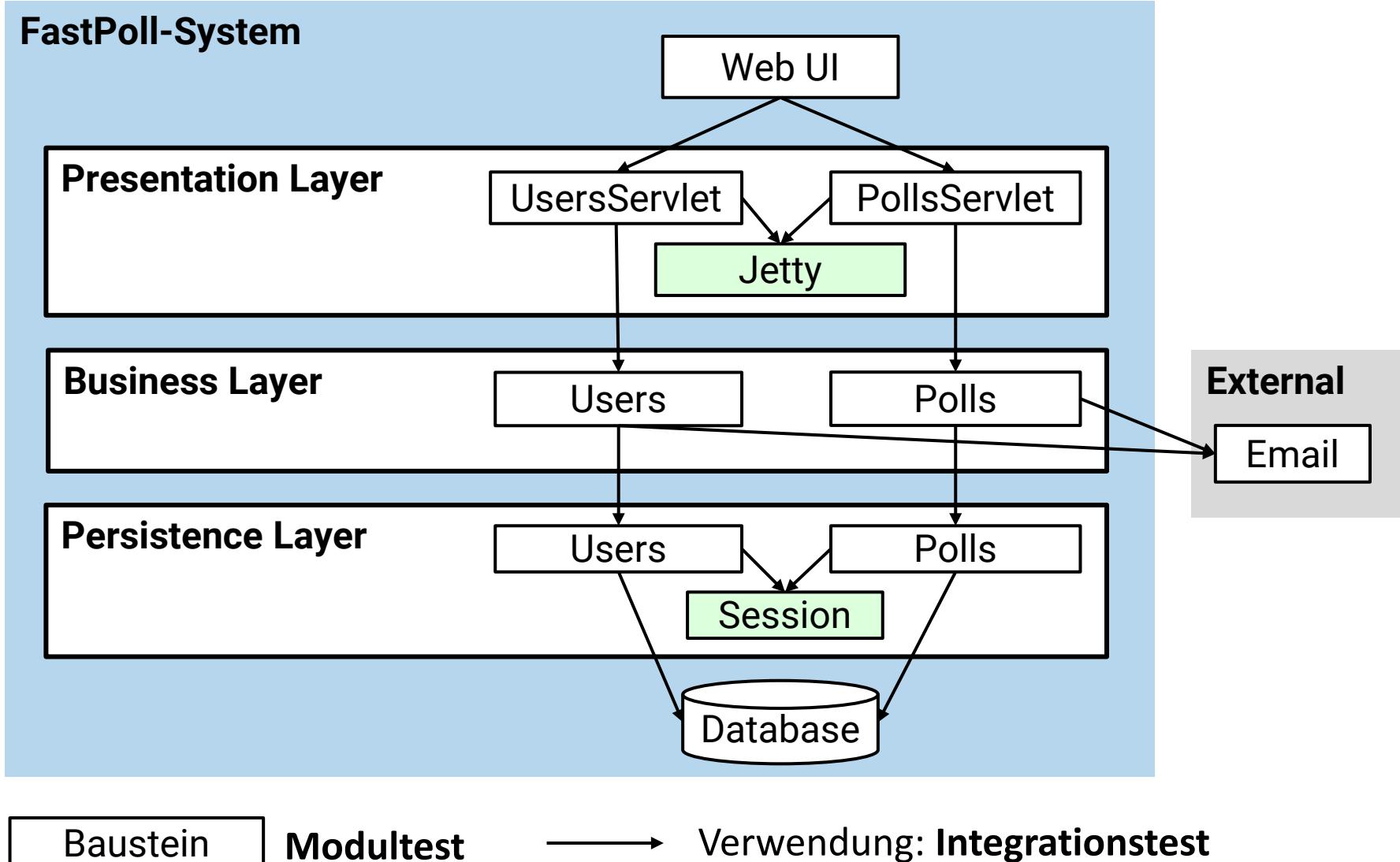
```
private static final String SENDER_ADDRESS = "noreply@fastpoll";
Properties emailProperties = ...; // email connection settings
AccountData accountData = ...; // data transfer object

public void createAccount() {
    accountData.store();
    Session session = Session.getInstance(emailProperties);
    try {
        Message message = new MimeMessage(session);
        message.setFrom(new InternetAddress(SENDER_ADDRESS));
        message.setRecipients(Message.RecipientType.TO,
            InternetAddress.parse(accountData.getEmail())));
        message.setSubject("FastPool: Account Verification");
        message.setText("Dear user, please use the code to...");
        Transport.send(message);
    } catch (MessagingException e) {
        throw new RuntimeException(e);
    }
}
```

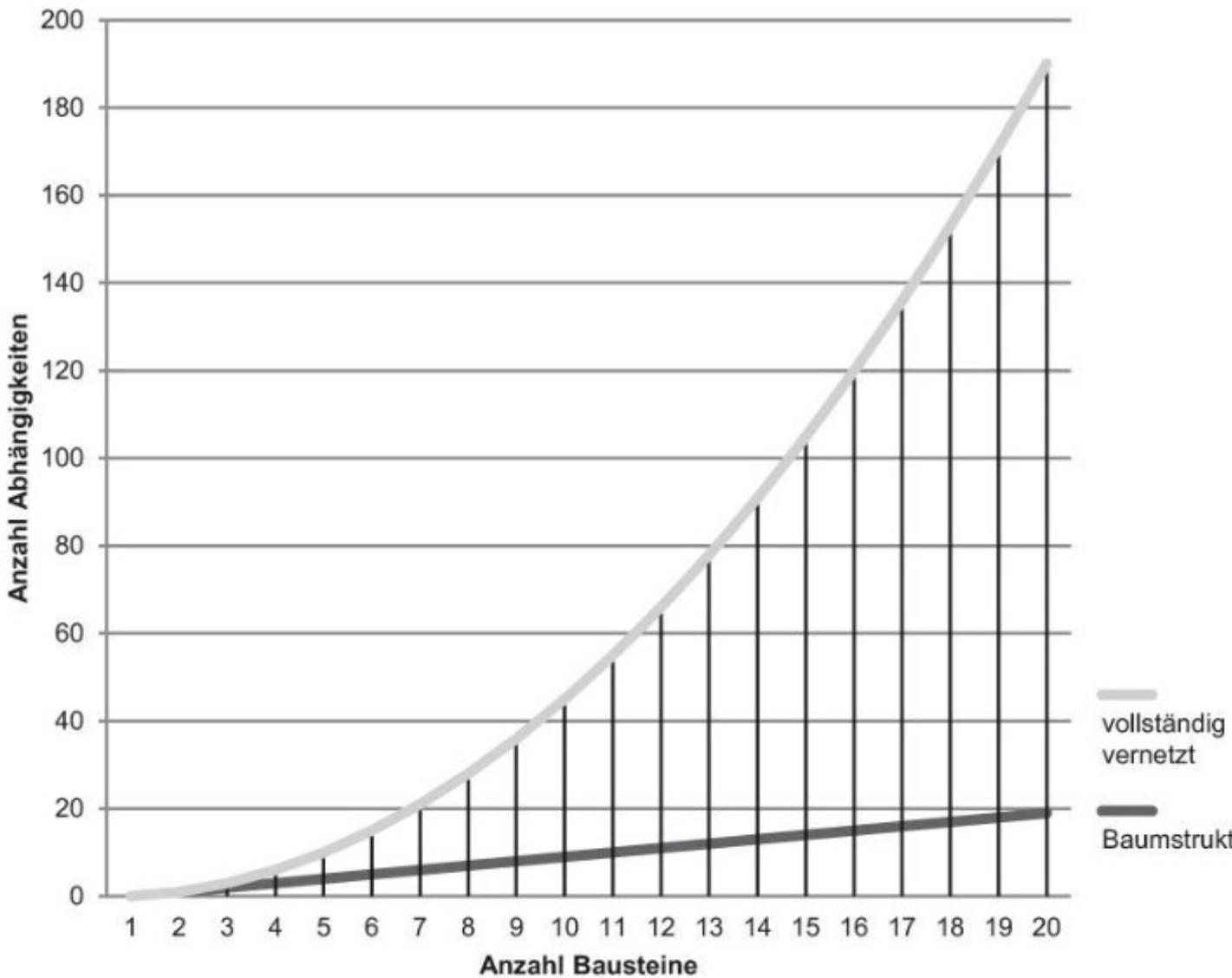
# Integrationsstrategien

- **Bottom-Up the Small:** Kleinere Teilsysteme lassen sich bottom-up integrieren.
- **Top-Down the Controls:** Bei aufwändigen Kontrollstrukturen mit Hilfe von Stubs top-down vorgehen und dann die richtigen Komponenten integrieren.
- **Big-Bang the Backbone:** Was für den weiteren Testablauf benötigt wird in einem Aufwisch zusammenführen.
- **Continuous Integration:** Bei iterativ-inkrementeller Entwicklung neu dazu gekommenes laufend integrieren und testen.

# Fallbeispiel: FastPoll-System ("Mini-Doodle")



# Auswirkung von Modulvernetzung auf Abhängigkeiten



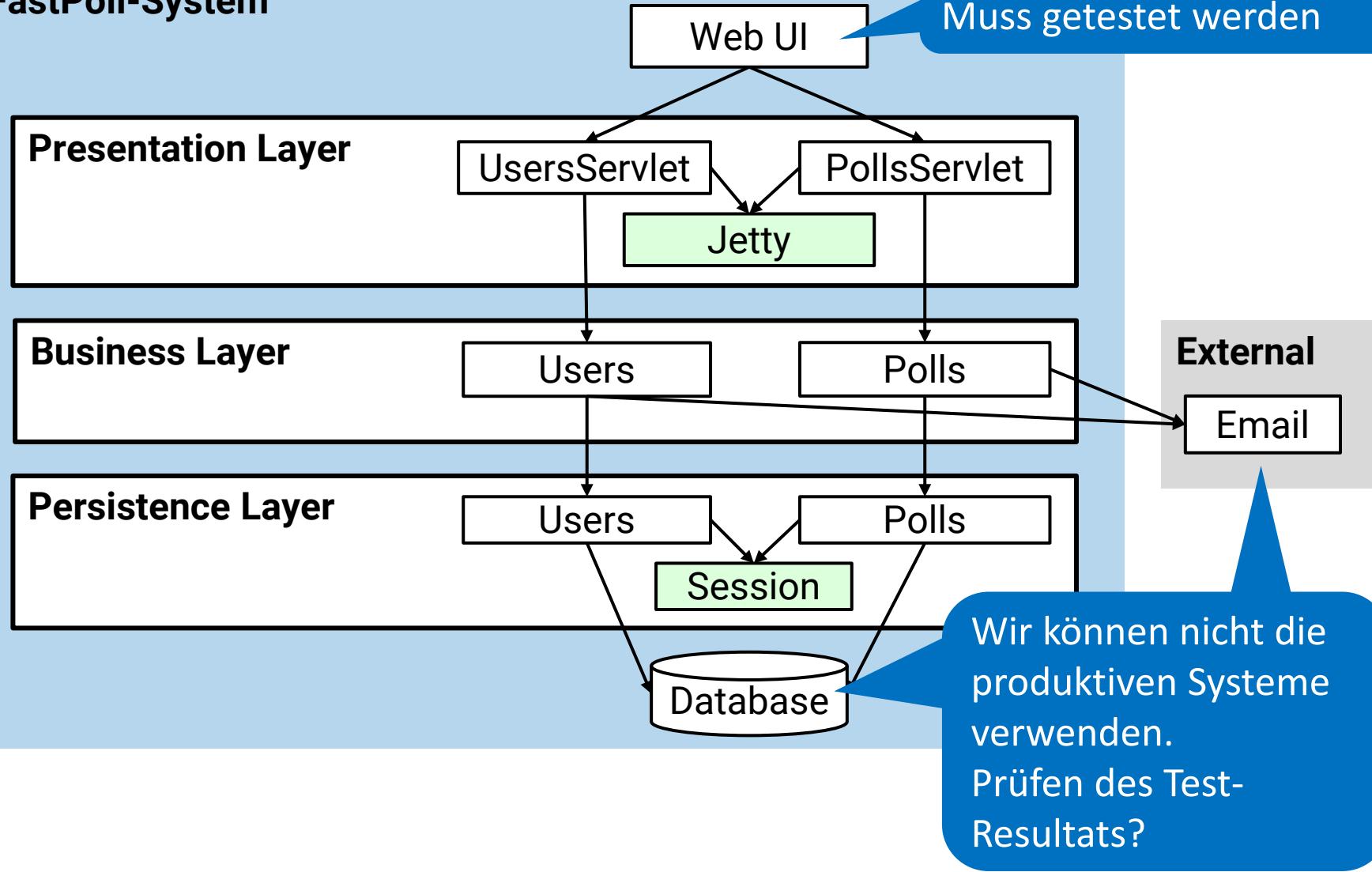
# **Systemtest**

# Systemtests

- Systemtests prüfen die **gesamte Wirkungskette im Softwareprodukt**, also Aspekte, die mit Unit-Tests und Integrationstests nicht abgedeckt werden.
- Ein potenziell lieferbares Softwareprodukt muss in der Regel:
  - ausserhalb der Entwicklungsumgebung lauffähig sein,
  - über eine Bedienschnittstelle verfügen
  - mit anderen Applikationen und Systemen interagieren.
- Es sind also Testfälle nötig, die in einer Testumgebung ablaufen, welche der **späteren Einsatzumgebung möglichst nahe** kommt.

# Fallbeispiel: FastPoll-System ("Mini-Doodle")

## FastPoll-System



# Herleitung der Systemtestfälle

- Systemtestfälle können grundsätzlich abgeleitet werden aus:
  - den Anforderungen.
  - zugehörigen, detaillierteren Use-Case-Beschreibungen.
  - den Akzeptanzkriterien.
- Nicht-funktionale Anforderungen werden oft wenig explizit festgehalten, entsprechend kommen auch nicht funktionale Tests zu kurz:  
Last- / Performance- / Stress- / Security- / Robustness-Tests sind ebenfalls wichtige Systemtests.
- Wie beim Test-First-Ansatz auf Unit-Test-Ebene fördert auch das Formulieren der Systemtests das Verständnis der Anforderungen und bringt Unklarheiten und Inkonsistenzen frühzeitig zu Tage.

# Test von Benutzerschnittstellen

moz-extension://efc0e4c7-be7c-40ae-bff9-f8fa2adf7a5d - Selenium IDE - Test\* - Mozilla Firefox

Project: Test\*

Executing ▾

Test\*

https://www.google.ch

Command	Target	Value
2	set window size	1044x557
3	click	name=q
4	type	name=q
5	check	name=q
6	run script	window.scrollTo(0,94)
7	click	css=.srg:nth-child(2) > .g:nth-child(2) .LC20lb

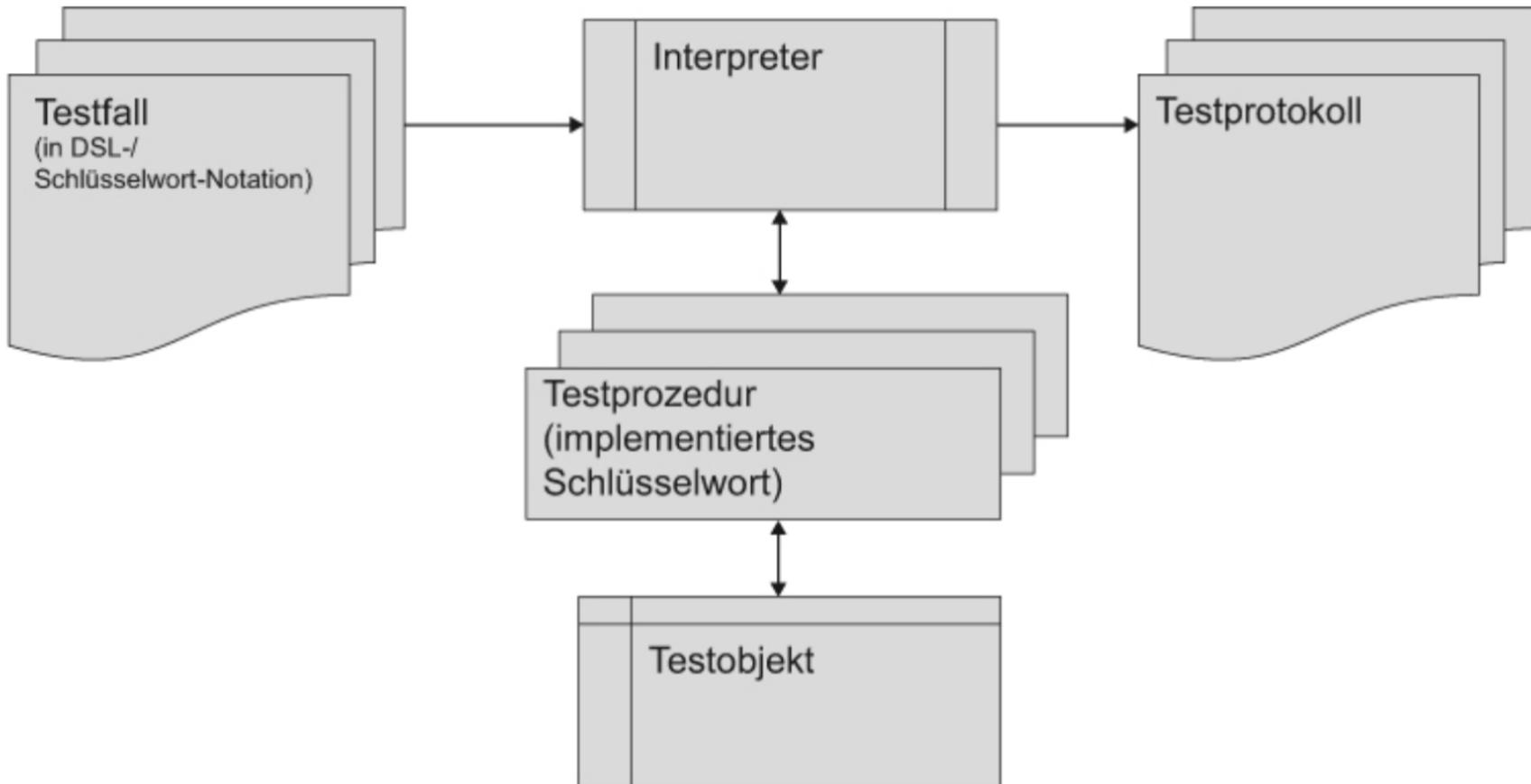
Runs: 1 Failures: 1

Anleitung wie für einen manuellen Test

Log Reference

# Entkopplung mittels dreischichtiger Testarchitektur

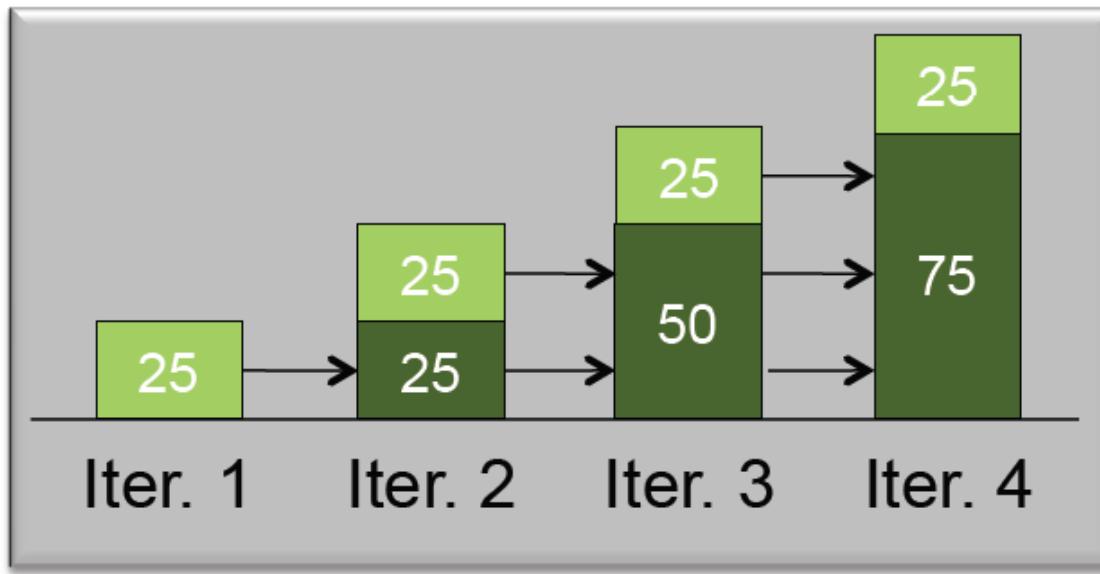
- Entkopplung der Testumgebung (z.B. Timeouts).
- Entkopplung der Testschnittstelle (verschiedene Oberflächen).



# Regressionstests

- Bereits realisierte und getestete Features müssen nach jeder Änderung / Erweiterung der Software erneut getestet werden  
(→ Regressionstest).

 Test neuer Funktionen  
 Test bestehender Funktionen (Regression)



Bildquelle: <https://www.slideshare.net/swissq/scrum-rocks-testing-sucks-de>

# Dokumentation

- Integrations- und Systemtests werden als **Regressionstest** auch in weiteren Entwicklungsschritten **immer wieder gebraucht** und genutzt.
- Damit Integrations- und Systemtests wiederholbar sind, müssen sie **nachvollziehbar dokumentiert** werden.
- Wichtige Bestandteile der Beschreibung eines Testfalls sind:
  - die Vorbedingungen für die Testausführung,
  - die Handlungen und Eingaben für die Durchführung des Tests,
  - die erwarteten Ergebnisse und Nachbedingungen.
- Durch Automatisierung wie bei Unitests sind auch Integrations- und Systemtests am besten dokumentiert.
- **(Teil-)Automatisierung lohnt sich!**

# **Testing in Scrum**

# Testplanung und -Organisation

## Sprintplanung

Zu Beginn jedes Sprints erarbeitet der ProductOwner mit dem Team auf Basis der Rahmenplanung und des ProductBacklogs das neue Sprint-Ziel.  
Im Sprint Planning werden die höchst priorisierten ProductBacklog-Items (Stories) so in das SprintBacklog übernommen, dass ein sinnvolles Sprintziel mit den verfügbaren Ressourcen in der vorgesehenen Timebox erreicht werden kann.

Der Sprintplan enthält:

1. Sprintziel
  - Aktualisierte Risikoliste
  - Was soll in diesem Sprint prinzipiell erreicht werden (allenfalls Bezug zu übergeordneten Projekt-Lieferobjekten, Meilensteinen etc.)
2. (Initial) Taskboard
  - Für diesen Sprint ausgewählte UserStories , inkl. Definition of «done»
  - Aufwand und Zuordnung (nötigenfalls UserStories in Tasks herunter brechen)
  - Abfolge im Sprint

Die Sprintpläne werden im PMP als Anhang abgelegt.



- Ein Sprintziel muss es sein, die Fertigstellung von Features zu mit Hilfe von Abnahmetests prüfen zu können.
- Konsequenz davon ist, dass im Taskboard zu den in diesem Sprint geplanten User-Stories auch die erforderlichen Integrationstests, Systemtests und Abnahmetests (-> Sprint-Reviews) eingeplant werden müssen.
- Natürlich müssen auch die jedes Mal zunehmenden Regressionstests eingeplant werden.

# Testaufgaben im Scrum-Team

- **Im Planning-Meeting:** Abschätzen wieviel Zeit zum Testen von User-Stories benötigt wird und dafür sorgen, dass diese bei der Aufwandschätzung berücksichtigt werden
- **Während dem Sprint:** Tests möglichst rasch durchführen, Anhäufung von pendenten Testfällen vermeiden
- **Product-Owner:** Nach Sprint aber vor Abnahme: Führt Akzeptanztests aus (Ist das Sprintziel erreicht?)
- **Sprint-Review:** getestete Features demonstrieren => inkrementelle Validierung
- **Retrospektive:** Wo waren die Stolpersteine aus Tester-Sicht, was lief besonders gut? Was kann man neu/anders machen.

# Zusammenfassung

- V-Modell - Verifizieren und valideren.
- Agile Testing: supporting the Team / critique the Product.
- Poking around is a waste of time.
- Grössere Testausbeute bei Verwendung verschiedener Testarten.
- Integrationstest: Fokus auf Schnittstellen und Zusammenspiel der Komponenten.
- Systemtest: Fokus auf gesamte Wirkungskette des Produkts.
- Regression Testing: nach jeder Änderung / Erweiterung die Software erneut testen: nur machbar wenn dokumentiert / automatisiert.

# Auftrag: Testing im Projekt "verteiltes Logging-System"

- Erstellen Sie eine Teststrategie für das Logger-Projekt. Diese ist im Kapitel 8 der Architekturbeschreibung festzuhalten. Umfang ca.  $\frac{1}{2}$  - 1 A4-Seite.
- Teil der Teststrategie sind mindestens drei manuelle Systemtests sowie automatisierte Unitests in geeignetem Umfang.
  - Ideen für Systemtests erhalten Sie aus den Akzeptanzkriterien der Userstories.
- Erstellen Sie für die drei manuellen Systemtests ein Drehbuch.
- [Ab Sprint 3] Separate Testberichte pro Sprint.
- Zeit für Testen (Regression!) in Sprintplanung mitplanen.

# Quellen / Literatur

- Testen in Scrum Projekten von Tilo Linz, dpunkt-Verlag.
- Agile Testing von Lisa Crispin & Janet Gregory, Addison-Wesley.
- Testing Object-Oriented Systems von Robert Binder, Addison-Wesley.
- Scrum Rocks, Testing Sucks?! von Adrian Stoll SwissQ, Testing Day 2011.

# Fragen?

Verteilte Systeme und Komponenten

# **Automatisiertes Testing**

**Wie man effektiv Software testet.**

Roland Gisler

# Inhalt

- Einführung und Motivation
- Test First Ansatz
- Abgrenzung zwischen Unit- und Integrationstests.
- Messen der Codeabdeckung von Tests
- Design: Dependency Injection
- Test Doubles (Mocking / Stellvertreter)
- Testen mit Hilfe von Containern (Integration)
- Zusammenfassung und Quellen

# Lernziele

- Sie kennen die verschiedenen Testarten und sind in der Lage gute Unit- und Integrationstests zu schreiben.
- Sie beherrschen die Entwicklung nach dem Test-First Prinzip.
- Sie nutzen Werkzeuge zur Messung der Codeabdeckung aktiv zu Verbesserung Ihres Codes und der Testfälle.
- Sie kennen das Prinzip der Dependency Injection (DI).
- Wie wissen was Test Doubles sind und können Mocking-Frameworks einsetzen.
- Sie sind in der Lage Container-Technologien für automatisierte Tests zu nutzen.

# **Einführung**

# Software Testing - Einführung

- Bei vielen Entwickler\*innen verpönt und unbeliebt. Warum?
  - «Ich kann programmieren, ich mache keine Fehler.»
  - «Ich will programmieren, nicht testen.»
  - «Ich muss den Termin einhalten, es muss schon fertig sein!»
  - «Wenn ich keinen Fehler finde, war das Testen verlorene Zeit!»
- Testen hat teilweise ein schlechtes Image und gilt als 'uncool'...
  - Einerseits in einzelnen Köpfen (Entwickler\*innen).
  - Andererseits aber auch in der Firmenkultur (Management).
- Aber: Tests gewährleisten, dass (speziell auch wichtige) Software möglichst fehlerfrei arbeitet!
  - z.B. medizinische Geräte, Verkehrsmittel, Atomkraftwerke etc.

→ Sie erinnern sich: Ariane 5 und andere Geschichten...

# Motivation für Testen – Erinnern Sie sich?

- Wir testen um **X**ehler zu finden? **Nein!**
- Besser:

**Wir testen kontinuierlich während der Implementation,  
um die Gewissheit zu haben dass es funktioniert!**

- Fehler finden bevor man Sie gemacht hat!
  - Fehler korrigieren bevor man Sie implementiert hat!
  - Oder mindestens Fehler schon im Ansatz (wenn es noch niemand anders gemerkt hat) finden!
- Wie macht man das?

**Test First!**

# Test First Methodik

- Entwickelt aus XP (extrem programming, u.a. von E. Gamma)
- Ganz einfacher Ansatz:  
**Vor** der Implementation immer **zuerst** die Testfälle schreiben!

Viele positive Effekte:

- Beim Schreiben der Testfälle denkt man auch an die konkrete Implementation des zu testenden Codes. Dabei **reift** diese buchstäblich **besser** heran!
- Dabei fallen einem viele **Ausnahmen** und **Sonderfälle** ein, welche man bei der Implementation wie **selbstverständlich** auch **berücksichtigt**.
- Ist die **Implementation** eines SW-Elementes **fertig**, kann dieses ohne aufwändige Integration **sofort getestet** werden!

# **Unit Tests**

# Unit Tests

- Werden häufig mit Komponenten-, Modul- und Entwicklertests gleich gesetzt.
- Sind funktionale Test von einzelnen, in sich abgeschlossenen Units (typisch Klasse, aber auch Komponente oder Modul).
- Ziele von guten Unit Tests:
  - **Schnell** und **einfach ausführbar, selbstvalidierend** (mit **assert\***-Methoden) und **automatisiert**.
  - Möglichst **ohne** Abhängigkeiten zu anderen Klassen, Komponenten oder Modulen (lose Kopplung).
  - Werden während der Entwicklung geschrieben und ausgeführt.
    - In der Entwicklungsumgebung (IDE).
    - Im automatisierten Buildprozesses (CI).
- Gute Unterstützung durch Frameworks (z.B. JUnit, TestNG etc.)

# Unit Tests: Nutzen

-  **Positiv:**
  - Neue oder veränderte Komponenten können sehr **schnell** getestet werden (regressiv).
  - Testen ist vollständig in die Implementationsphase integriert.
  - Test First Ansatz ist möglich.
  - Automatisiertes, übersichtliches Feedback / Reporting.
  - Messung von → Codeabdeckung kann integriert werden.
-  **Negativ:**
  - Für GUI(-Komponenten) etwas aufwändiger.
  - Qualität und Nachvollziehbarkeit der Testfälle muss im Auge behalten werden: Qualität vor Quantität!
  - In manchen Architekturen / Umgebungen schwierig umsetzbar.

# **Integrationstests mit JUnit**

# Integrationstest mit JUnit - Namenskonvention

- **Wichtig:** JUnit (und andere Frameworks) können zur Automatisierung (fast) **aller** Testarten verwendet werden!
- Für Integrationstests existiert für JUnit (und Apache Maven) eine eigene Namenskonvention:
  - Klassenname **XyzIT** für Integrationstests.
  - Werden auch unter **src/test/java** abgelegt.
- Die Unterscheidung ergibt sich «**nur**» durch den Zeitpunkt der Ausführung, und deren (Laufzeit-)Abhängigkeiten.
- Integrationstest können mit Apache Maven mit dem eigenen Stage **integration-test** bzw. **verify** ausgeführt werden.
  - Getrennte Plugins **surefire** und **failsafe** weisen die Testresultate auch getrennt (Unit und Integration) aus.

# Abgrenzung: Unit vs. Integrations Tests

- Wird häufig individuell festgelegt und kontrovers diskutiert.
- Unit Tests sind wirklich Unit Tests, wenn sie (unter anderem)...
  - auf einem beliebigen System und jederzeit lauffähig sind.
  - (bei Java) auch auf unterschiedlichen Betriebssystemen laufen.
- Konsequenz:
  - Testfälle welche z.B. mit dem **Dateisystem** interagieren, sind in strenger Sichtweise bereits **Integrationstests!**
  - Testfälle die z.B. Sockets verwenden (auch wenn nur auf «localhost») sind bereits Integrationstests!
- Unit Tests sollten somit **nie** aufgrund von «Fremdeinflüssen» fehlschlagen!
  - Beispiel: falscher Pfad ('\ vs. '/), Platz, Zugriffsrechte etc.

# Empfehlungen zu Unit und Integrations Tests



- Machen Sie eine bewusste Trennung zwischen den beiden Kategorien.
- Einigen Sie sich im Team / Organisation auf eine gemeinsame Philosophie wie sie die Kategorien exakt aufteilen.
  - Akademische Sturheit hilft nicht – pragmatische Einheitlichkeit hingegen schon!
  - Kann auch mal projektspezifisch sein!
- Je mehr als Unit Test realisiert werden kann, je besser!
- **Aber jeder automatisierte Test ist ein grosser Gewinn!**
- Nutzen Sie zusätzliche Hilfsmittel wie →Code Coverage oder →Test Doubles (Mocking), oder →Testcontainer um die Motivation zu steigern.

# **Messung der Code Coverage**

# Was ist Code Coverage?

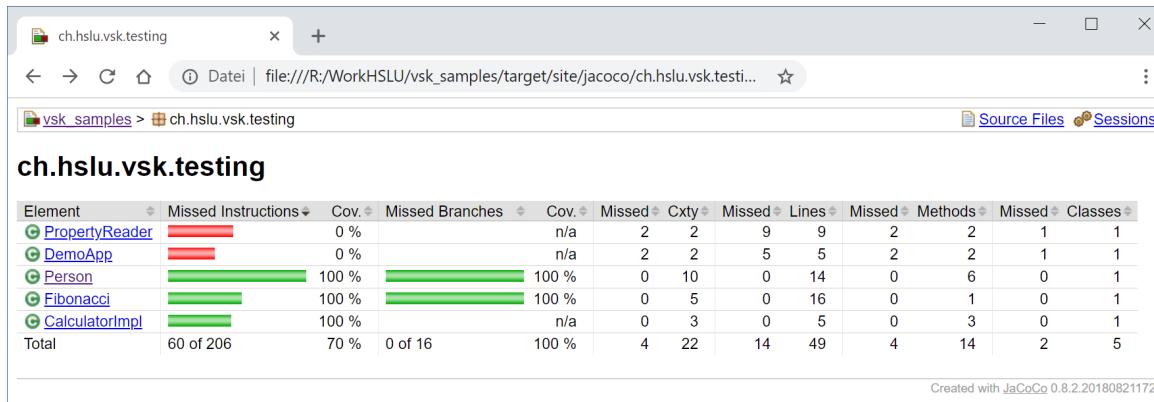
- Code Coverage ist eine Metrik welche zur Laufzeit misst, welche Quellcodezeilen ausgeführt wurden.
- Diese Messung erfolgt typisch während der Ausführung der Testfälle.
  - Kann auch zur 'normalen' Laufzeit erfolgen, dann z.B. zur Messung welche Funktionen tatsächlich genutzt werden!
- Somit kann eine Aussage gemacht werden, wie umfassend der Code tatsächlich genutzt bzw. getestet wurde!
- Umfangreiche, statistische Aufbereitung der Daten möglich:
  - Nach Testfall, Komponente, Package, Teilsystem etc.
  - Auf Buildserver z.B. auch historisiert, d.h. zeitliche Veränderung der Werte wird sichtbar gemacht.

The screenshot shows a Java code editor window with the file 'Person.java' open. The code defines a class Person with methods for getName, getVorname, equals, hashCode, and toString. Green diamond markers are placed at the start of each method body, indicating they have been executed during testing. The code is as follows:

```
45.  /*
46.  * @return the vorname.
47.  */
48. public String getVorname() {
49.     return vorname;
50. }
51.
52. @Override
53. public boolean equals(final Object obj) {
54.     if (obj == this) {
55.         return true;
56.     }
57.     if (!(obj instanceof Person)) {
58.         return false;
59.     }
60.     final Person other = (Person) obj;
61.     return Objects.equals(this.name, other.getName()) && Objects.equals(this.vorname, other.getVorname());
62. }
63.
64. @Override
65. public int hashCode() {
66.     return Objects.hash(this.name, this.vorname);
67. }
68.
69. /**
70. * @see java.lang.Object#toString()
71. */
72. @Override
73. public String toString() {
74.     return String.format("Person{name: %s, vorname: %s}", this.name, this.vorname);
75. }
76. }
```

# Coverage - Motivation für Einsatz

- Herausforderung: Wie implementiert man mit möglichst geringem Aufwand trotzdem möglichst umfassende Testfälle (Effizienz!)?
  - Wie kann man die Qualität von Testfällen beurteilen?
- Messen der durch die Testfälle erreichten Codeabdeckung!
  - Während der Ausführung der Tests wird gemessen, welche Statements/Codezeilen tatsächlich vom Test erfasst wurden.



- **Vorsicht:** Eine hohe Coverage ist **kein** Beweis für gute Testfälle oder gar die Fehlerfreiheit des Codes!

# Coverage - Was kann man messen?

- Man unterscheidet verschiedene Messtechniken und -werte:
  - Statement Coverage (Line Coverage)
  - Branch Coverage
  - Decision Coverage
  - Path Coverage
  - Function Coverage
  - Race Coverage
- Messwerte sind unterschiedlich Aufwändig und Aussagekräftig.

# Coverage - Was wird gemessen? (1)

- Statement Coverage 
  - Misst ob (und wie häufig) eine Codezeile durchlaufen wurde.
  - Problem: Handelt es sich bei der Zeile z.B. um einen logischen Vergleich/Ausdruck, ist ein einmaliger Durchlauf nicht repräsentativ.
- Branch Coverage
  - Prüft, dass alle Zweige einer bedingten Anweisung ausgeführt wurden.
- Decision Coverage 
  - Bei Fallunterscheidungen (**if**, **while** etc.) wird geprüft, dass alle Teilausdrücke in der Bedingung auf **true** und **false** aufgelöst wurden (strenger als Branch Coverage).

## Coverage - Was wird gemessen? (2)

- Path Coverage
  - Bei der Path Coverage wird gemessen, ob alle möglichen Kombinationen von Programmablaufpfäden durchlaufen wurden.
  - Problem: Die Anzahl der Möglichkeiten steigt exponentiell mit der Anzahl Entscheidungen → in der Praxis nicht durchführbar.
- Function Coverage
  - Misst auf der Basis der Funktionen ob sie aufgerufen wurden.
- Race Coverage
  - Konzentriert sich auf Codestellen die parallel ablaufen.

# Coverage - Technische Umsetzung

- Instrumentierung des Quellcodes 
  - Der Quellcode wird durch einen Preprocessor vor dem Compilieren mit Statements zur Coverage-Messung ergänzt.
  - Nachteil: Modifizierter Quellcode (man denke an Debugging).
- Instrumentierung des Bytecodes 
  - Der Bytecode wird bei/nach der Kompilierung mit Bytecode zur Coverage-Messung ergänzt.
  - Nachteil: **class**-Dateien müssen separiert werden (Deployment).
- Just-in-time Instrumentierung zur Laufzeit 
  - Instrumentiert den Bytecode direkt während des Classloadings.
  - Vorteile: Nur ein Binary, unabhängig von Compiler, jederzeit und überall ad-hoc aktivierbar!
  - Nachteil: Teilweise «Konkurrenz» bei Bytecode-Manipulation.

# Coverage - Wer misst und wann?

- Gemessen wird die Abdeckung bei der Ausführung des Codes durch den (modifizierten) Code selber.
  - Das Coverage-Werkzeug **instrumentiert** den Code.
  - Messung sehr häufig bei der Ausführung der Testfälle.
  - Daten werden typisch in eine spezifische Datei persistiert.
- Ein populäres Coverage Werkzeug für Java ist JaCoCo.
  - Bestandteil von EclEmma, welches zu Eclipse gehört.
  - Details siehe <https://www.eclemma.org/jacoco/index.html>
  - Ist in VSK-Projekten (Basis: oop\_maven\_template) integriert.
- Es ist ein Buildtool, die IDE oder der Buildserver werten die Resultate «nur» aus und stellen diese dann ansprechend dar.
  - Mit Apache Maven wird mit `mvn jacoco:report` ein HTML-Report erzeugt, siehe `./target/site/jacoco/index.html`

# Live Demo's / Screencast's

- Messung der Code Coverage mit JaCoCo / Console:

[E02 SC04 TestingCoverageConsole.mp4](#)

- Messung der Code Coverage in Eclipse:

[E02 SC05 TestingCoverageEclipse.mp4](#)

(Screencasts aus Modul OOP)



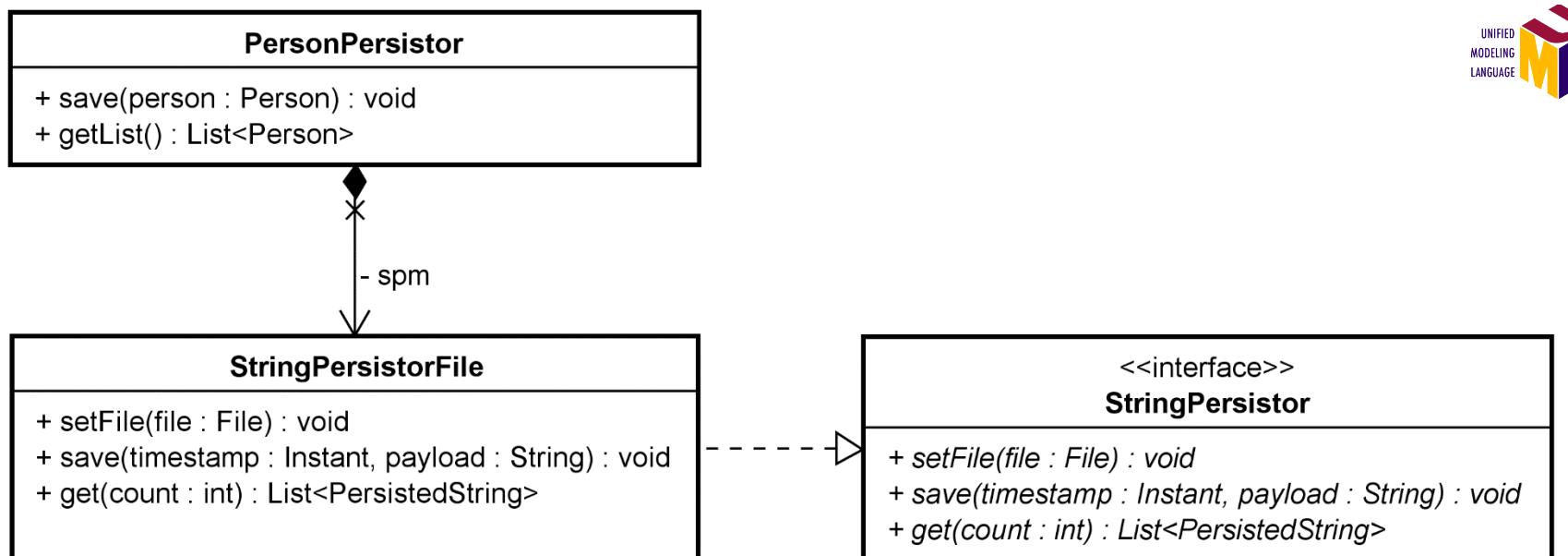
# **Dependency Injection**

# Schlechte Testbarkeit

- Oft stellen Entwickler\*innen fest, dass sich selbst einfache Klassen oder Komponenten «schlecht» testen lassen.
  - Das führt sehr schnell zum Verzicht auf (Unit-)Tests, *oder*:
  - Es werden wieder vermehrt Integrationstests implementiert.
- Bei näherer Betrachtung sind oft zu viele bzw. **zu stark gekoppelte** Abhängigkeiten die Ursache.
  - Deren negative Auswirkungen fallen bei der ersten Anwendung (und das sind die Testfälle!) sehr schnell auf.
- Die eigentliche Ursache ist somit schlicht **schlechtes Design!**
- Darum: Lässt sich eine Softwareeinheit nur «schlecht» (kompliziert und/oder aufwändig) testen, sollte man das **Design immer kritisch hinterfragen!**

# Beispiel: PersonPersistor

- Wir wollen eine Klasse **PersonPersistor** implementieren, welche **Person**-Objekte als **Strings** serialisiert in eine Datei speichert.
- Wir haben bereits eine erprobte und getestete Implementation eines universellen **StringPersistorFile** → Wiederverwenden!
- Ein erster, primitiver Lösungsansatz (dabei bleibt es leider zu oft!):



## Beispiel: PersonPersistor mit hoher Kopplung – Schlecht!

- PersonPersistor erzeugt sich seinen StringPersistor selber.

```
public final class PersonPersistor {  
  
    private final StringPersistorFile spm =  
        new StringPersistorFile();  
  
    ...  
}
```



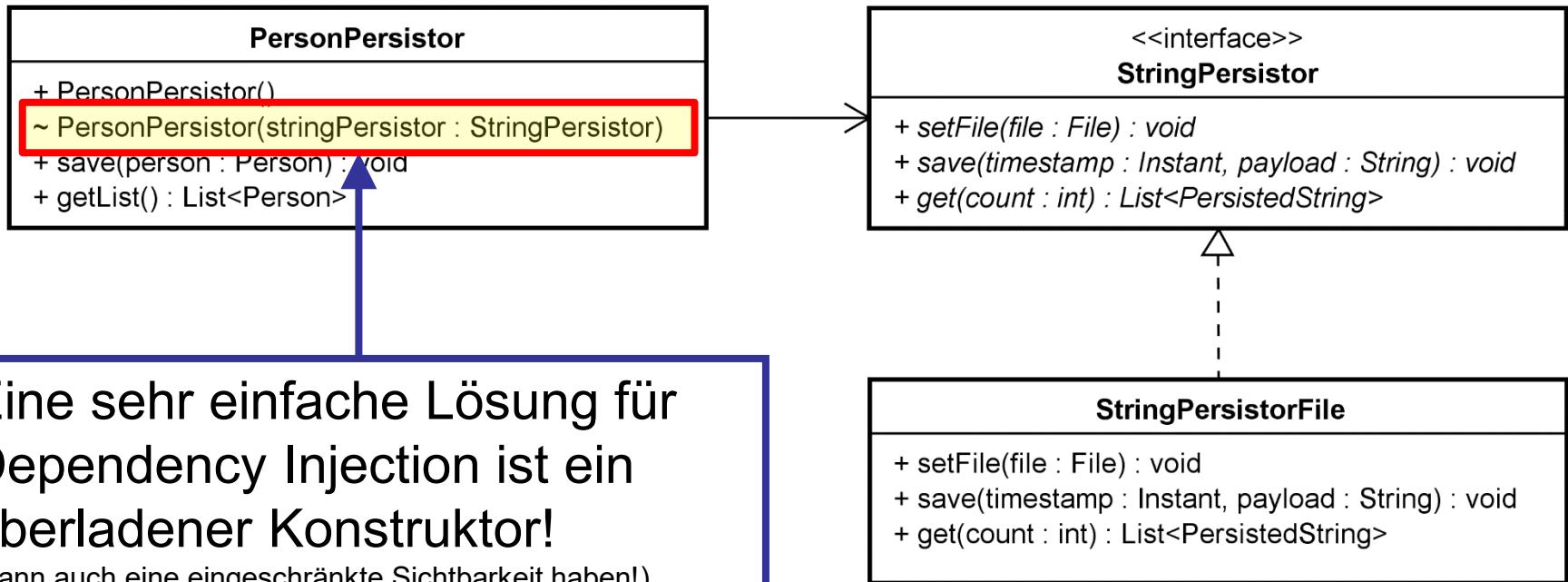
- In der Realisierung sehr einfach, aber das ist auch der einzige Vorteil dieser Lösung!
- Dagegen stehen einige negative Aspekte:
  - Typen und Implementation fest verdrahtet (starke Kopplung).
  - Abhängigkeit zu einer Implementationsklasse (obwohl ein Interface existieren würde!).
  - Unflexible Implementation, wie testet man diese?

# Probleme beim Testen des PersonPersistor

- Der **PersonPersistor** enthält die Logik um **Personen** in **Strings** zu serialisieren und wieder zu deserialisieren. Diese Funktionalität **wollen** wir testen.
- Da intern ein **StringPersistorFile** verwendet wird, ist das aber schwierig, weil dieser auf das Dateisystem zugreift, und wir damit sofort in der Kategorie →**Integrationstest** sind!
- Zudem testen wir beim Test des schlanken **PersonPersistors** den (hoffentlich schon getesteten) **StringPersistorFile** ungewollt nochmal mit. → Selektivität des Testfalles sinkt!
- Es wäre besser, wenn wir (mindestens für das Testen) die Dependency auf den intern verwendeten **StringPersistorFile** durch etwas anderes ersetzen könnten!

# Lösung: Dependency Injection (DI) – gut!

- Die Lösung: Wir verwenden Dependency Injection!
- Eine Klasse/Komponenten erzeugt ihre Abhängigkeiten nicht selber, sondern lässt sich diese (wahlweise) auch **von Aussen** übergeben.



# Vorteile beim Einsatz von Dependency Injection

- Man ersetzt den konkreten Typ durch ein **Interface**, womit die Kopplung stark abnimmt.
  - **DIP** – **Dependency Inversion Principle** (aus → S.O.L.I.D.)
  - Dadurch können auch verschiedene, alternative Implementationen genutzt werden.
  - Es resultiert eine bessere «**Separation of Concerns**» (SoC).
- Das Beste: Die Testbarkeit wird dadurch massiv vereinfacht! Man kann während der Tests eine alternative Implementation als Platzhalter →**Test Double** einfügen.
  - Integrationstests werden somit wieder zu Unit Tests!
  - Es resultieren schnellere und selektivere Tests!
- Was sind Test Doubles?

# Beispiel: Test mit einer Fake-Implementation

- Für die Testausführung wird die Fake-Implementation **StringPersistorMemory** (statt **File**) verwendet, welche die Strings nur im Memory «speichert»:

```
@Test  
public void testGetEmptyList() {  
    final PersonPersistor instance =  
        new PersonPersistor(new StringPersistorMemory());  
    assertThat(instance.getList()).isEmpty();  
}
```

- Somit resultiert für diesen Testfall:
  - Er hat **keine** Abhängigkeit zum Dateisystem mehr.
  - Er ist wieder ein **Unit Test** (statt Integrationstest).
  - Er testet viel weniger Dritt-Code und wird dadurch **selektiver!**

# **Effektives Testen mit Test Doubles**

# Nachteile von Fake-Implementationen beim Testen

- So elegant die Idee ist, so hat sie auch grosse Nachteile: Je besser und umfangreicher man testet, umso grösser wird die Anzahl von unterschiedlichen Fake-Implementationen.
  - Unterhalt der Fake-Implementationen erhöht aufwand.
  - Übersichtlichkeit leidet (was ist Echt und was ist Fake?).
- Die Lösung: Man kann während der Tests eine zur Laufzeit, dynamisch erstellte (!) Implementation als Platzhalter →**Test Double** einfügen.
  - Integrationstests werden somit wieder zu Unit Tests!
  - Es resultieren schnellere und selektivere Tests!
  - Wir haben nicht unzählige «Fake»-Implementation.
- Was sind Test Doubles? →siehe Input [EP 23 TestDoubles.pdf](#).

## Beispiel 2: Test mit einem Mock

- Für die Testausführung wird ein Mock (oder Spy) verwendet, welcher direkt im Testfall erzeugt und konfiguriert wird:

```
@Test
public void testGetEmptyList() {
    final StringPersistor mock =
        mock(StringPersistor.class);
    when(mock.get(0)).thenReturn(Collections.emptyList());
    final PersonPersistor instance =
        new PersonPersistor(mock);
    assertThat(instance.getList()).isEmpty();
}
```

- Der Testfall
  - Hat **keine** Abhängigkeit mehr zu einer Implementation.
  - Seine **Selektivität** ist somit **maximal!**

Hinweis: Das `mock`-Objekt ist ein Proxy (Proxy-Pattern nach GoF) und wird hier mit Mockito erzeugt.

# Live Demo's / Screencast's

- Unit Test mit (schlechter) Integration:

[EP 22 SC03 TestingBadIntegration.mp4](#)

- Unit Test mit einer Fake-Implementation (Test-Double):

[EP 22 SC04 TestingDoublesFake.mp4](#)

- Unit Test mit Mocking-Framework (Mockito):

[EP 22 SC05 TestingDoublesMock.mp4](#)



# **(Integrations-)Testen mit Containern**

# Das «Leiden» bei Integrations-Test

- Selbst wenn Integrations-Tests eigentlich automatisiert sind, stellen sie uns vor grosse Probleme: Die nötigen «Umsysteme» müssen meist in mühsamer Arbeit manuell installiert und konfiguriert werden.
  - z.B. Datenbankserver, Applikationsserver, Webserver etc.
  - viel Handarbeit, fragil, grosse Fehleranfälligkeit.
  - Testdatenmanagement: Bei manchen Projekten ein sehr wichtiges (und häufig unterschätztes) Thema!
  - Dokumentationsaufwand, Versionierung!
- Eine sehr potente Lösung für einige dieser Herausforderungen: Docker und Testcontainers - <https://www.testcontainers.org/>



# Integrations-Tests mit Docker-Containern

- Die Grundidee: Wir bauen für die Tests individuelle Container, welche schnell und einfach konfiguriert, hochgefahren und danach ohne Spuren wieder abgeräumt werden können.
  - Noch dazu: Das geht alles schnell! (vgl. virtuelle Maschine)
- **Testcontainers** ist ein JUnit ergänzendes Framework, welches:
  - Eine Java-API zu vielen Docker-Aufgaben anbietet.
  - Vorbereitete Images (als Klassen!) zur Verfügung stellt.
  - Das Hoch- und Runterfahren weitgehend automatisiert.
  - Sich automatisch um das Portmapping kümmert!
- Für maximale Flexibilität können eigene Images sogar innerhalb eines Testfalles (ad-hoc) mit einer sehr eleganten Fluent-API erstellt und konfiguriert werden.
  - Vergleiche: Dynamische Mock-Erstellung und Konfiguration.

# Beispiel 1: Echo Server – Testen des Startes – 1/2

- Testcontainers erlaubt uns Container zu definieren, welche dann vollautomatisch vor jedem (!) einzelnen Test hochgefahren werden.

```
@Testcontainers
class EchoServerDefaultPortIT {

    @Container
    GenericContainer<?> echoServer
        = new GenericContainer<>(DockerImageName.parse("repo/echoserver:latest"))
            .withStartupTimeout(Duration.ofSeconds(1))
            .withExposedPorts(EchoServer.DEFAULT_PORT);
    ...
}
```

- **@Testcontainer** markiert die Testklasse für das Framework.
- **@Container** definiert einen konkreten Container (als Objekt).
  - in max. 1s muss er oben sein (auf Port reagieren), sonst failure!
- Container erhält dynamische IP und Port, stellt sicher, dass es keine Kollisionen mit bereits laufenden Containern gibt!

## Beispiel 1: Echo Server – Testen des Startes – 2/2

- Der erste Testfall prüft, ob der Server korrekt hochfährt:

```
@Test  
void testServerStart() {  
    assertThat(server.getLogs())  
        .contains("started").contains("5555");  
}
```

- Mit `getLogs()` kann man auf die Ausgaben des Containers (Container-Log) zugreifen.
- Beispiel-Ausgabe des Servers beim Start:

```
2022-10-27 18:14:29,313 INFO - EchoServer on 'Oracle Corp.' ↵  
started - listening on tcp://*:5555
```

→ Somit wurde der Server offenbar korrekt gestartet.

## Beispiel 2: Test ob ein Echo kommt – 2/3

- Der zweite Testfall prüft, ob der Server ein Echo gibt (und loggt):

```
@Test
void testServerGetEchoAndLog() {
    final String url = String.format("tcp://%s:%s",
                                      echoServer.getHost(), echoServer.getFirstMappedPort());
    try (ZMQ.Socket socket = new ZContext().createSocket(SocketType.REQ)) {
        socket.connect(url);
        socket.send("Hallo!".getBytes(ZMQ.CHARSET));
        assertThat(new String(socket.recv(), ZMQ.CHARSET)).isEqualTo("Hallo!");
    }
    assertThat(echoServer.getLogs()).contains("Hallo!");
}
```

- `getHost()` und `getFirstMappedPort()` liefern die dem Container dynamisch zugewiesene IP bzw. Port.
- Beispiel mit ZeroMQ, prüft auf die korrekte (Echo-)Antwort, und ob das im Server auch geloggt wird.

## Beispiel 3: Individuelles Test-Image bauen(ad-hoc)

- Wir können mit Testcontainers sogar ad-hoc Images bauen, dazu steht uns eine elegante Fluent-API zur Verfügung:

```
@Container
GenericContainer<?> echoServer
    = new GenericContainer<�新器>(new ImageFromDockerfile("ad-hoc/echoserver-temurin", false)
        .withFileFromFile("./target/echoserver.jar", new File("./target/echoserver.jar"))
        .withDockerfileFromBuilder(builder -> builder
            .from("eclipse-temurin:17.0.4.1_1-jre-alpine")
            .expose(5555)
            .workDir("/app")
            .copy("./target/echoserver.jar", "/app/")
            .cmd("java", "-jar", "echoserver.jar")
            .build()))
        .withStartupTimeout(Duration.ofSeconds(1))
        .withExposedPorts(5555);
```

- **withFileFromFile()** – Beispiel für Zugriff auf Build-Kontext
- **withDockerfileFromBuilder()** – FluentAPI mit Builder-Pattern, welches quasi die Syntax des **Dockerfile** nachbildet.
  - Ist das Elegant? ☺ Ginge natürlich auch mit einem **Dockerfile**.

# Live Demo's / Screencast's

- Integrationstests mit Testcontainers (Docker Container):

[EP 22 SC06 Testcontainers.mp4](#)



# Herausforderungen von Testcontainern

- Das Konzept ist faszinierend und hat eine grosse Mächtigkeit!
  - Es stellt aber auch grosse Ansprüche an die Infrastruktur:
    - Images müssen gespeichert werden, bzw. sollten gezielt verwaltet (und auch gelöscht!) werden.
    - Eigene Registries und Repositories werden unverzichtbar.
    - Was vorher «individuell» auf einem Rechner installiert wurde, zentralisiert sich jetzt z.B. auf einer Buildinfrastruktur.
  - Docker-in-Docker («dind») ist möglich, muss aber explizit berücksichtigt und vorgesehen werden.
- Aus diesen Gründen beschränken wird uns in VSK vorerst auf eine rein «lokale» Anwendung!
- Persönlicher Docker-Account, Ausführung «nur» auf eigenem (Entwickler\*innen-)Rechner.

# Auftrag für das Projekt

- Testen Sie!
  - Primärer Fokus: **Automatisierte** Tests!
- Versuchen Sie möglichst nach Test-First Methodik zu entwickeln.
  - Beginnen Sie bei einfachen, kleinen Klassen mit **Unit Tests!**
- Erfahren und erkennen Sie die Abgrenzung zu **Integrationstests**
  - Sie können auch Integrationstest automatisieren!
  - Dazu verwenden Sie JUnit und ggf. Testcontainer.
  - Klassen der Integrationstest enden auf **\*IT**.
  - Ausführen (nur lokal) mit dem Target: **mvn integration-test**
  - Optional: Coverage-Report erstellen mit **mvn jacoco:report**
- Optional: Lokale Versuche mit Testcontainern.

# Zusammenfassung

- Nicht nachträgliches Testen zur Fehlersuche, sondern kontinuierliches Testen zur Bestätigung, dass es funktioniert!
- Test First - Ansatz als sehr attraktive Methode aus dem XP-Ansatz.
- Messung der Code Coverage als Motivationsfaktor.
- Designregeln für gute Testbarkeit: SRP, SoC, Dependency Injection.
- Test Doubles (Mocking) um Testfälle noch stärker zu entkoppeln und mehr Unit Tests (statt Integration) machen zu können.
- Integrations-Tests mit Container-Technologien machen das (Test-)Leben noch viel spannender.



**Testen macht noch mehr Spass!**

**Fragen?**

# Quellen 1

- JUnit Testframework, <https://junit.org/junit5/>
- AssertJ, <https://assertj.github.io/doc/>
- Code Coverage Messung:
  - EclEmma, JaCoCo, <http://www.eclemma.org/jacoco/>
  - Clover, <http://www.atlassian.com/software/clover/> (komerz.)
- Mocking Frameworks, Open Source (Beispiele):
  - Mockito, <https://site.mockito.org/> - Empfehlung
  - EasyMock, <http://easymock.org/>
  - MockRunner, <http://mockrunner.github.io/>
- Docker Container:
  - Testcontainers, <https://www.testcontainers.org/>

Verteilte Systeme und Komponenten

# **Testing: Test Doubles**

**Warum nicht alles ein Mock ist.**

Roland Gisler

# Inhalt

- Test-Doubles – warum nicht alle Mocks oder Stubs sind
- Anforderungen an das Design
- Test-Doubles:
  - Dummy, Stub, Spy, Mock und Fake
- Empfehlungen
- Quellen

# Lernziele

- Sie verstehen was Test Doubles sind und können sie erklären.
- Sie kennen die verschiedenen Arten von Test Doubles und können diese adäquat einsetzen.
- Sie kennen exemplarische Mocking-Frameworks und können diese nutzen.

# Was sind «Test Doubles»?

- Ein «Double» ist ein Platzhalter für die echte, produktive Implementation während der Tests.
  - In Anlehnung an «stunt doubles» für Schauspieler\*innen beim Film.

Jennifer Lopez (rechts)  
mit ihrem (männlichen)  
«stunt double» (links).



© xposurephotos.com

- Häufig spricht man unpräzis nur von Mocks und Mocking.
  - siehe <http://blog.8thlight.com/uncle-bob/2014/05/14/TheLittleMocker.html>
- Der korrekte Oberbegriff ist «**Test Double**», davon gibt es dann verschiedene, interessante Spezialisierungen.

# Warum Test Doubles?

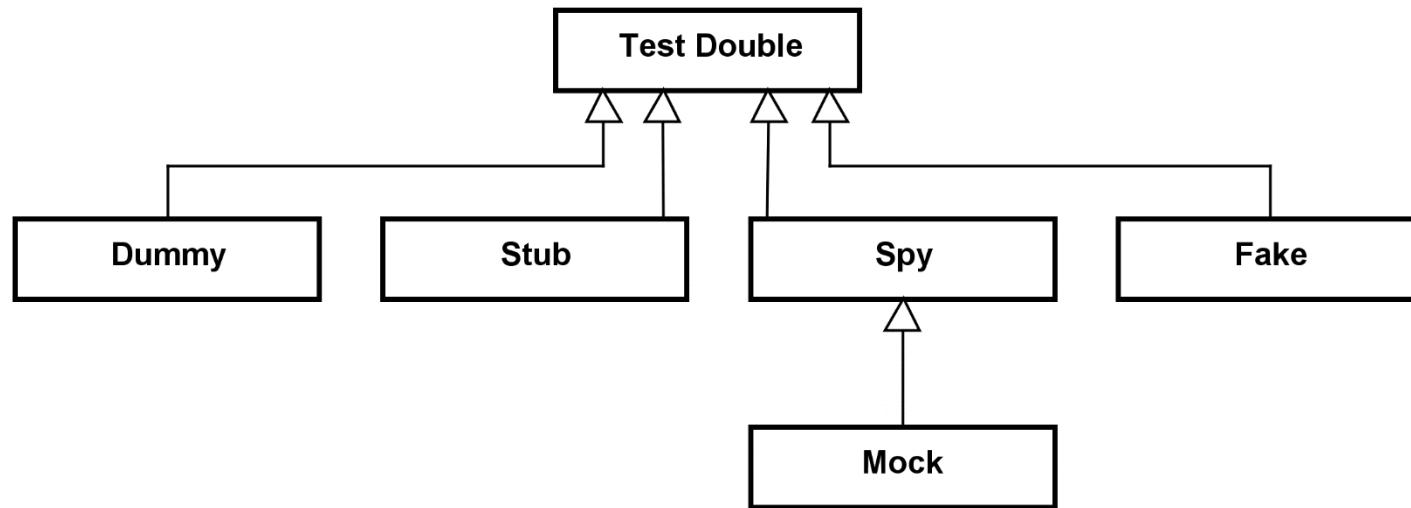
- Test Doubles dienen hauptsächlich dazu den Aufwand für Integrationstests zu reduzieren, indem man stattdessen mehr Testfälle als einfachere Unit Tests realisieren kann.
- Man will so viel wie möglich mit **Unit Tests** prüfen, weil:
  - Erste Teststufe, direkt bei der Entwickler\*in.
  - Schnell, häufig, überall lauffähig, vollständig automatisiert.
  - Hohe Selektivität der Testfälle.
- Test Doubles können aber auch innerhalb von Integrationstests sehr nützlich sein.
  - Gezielte Isolation der Tests von einzelnen Integrationen (Abhängigkeiten von anderen Systemen).

# Anforderungen für Testen mit Test Doubles

- Damit das Testen mit Test Doubles gelingt, muss ein entsprechend gutes Design vorliegen!
  - Gutes Testen und gutes Design unterstützen sich **gegenseitig!**
- Der Einsatz von **Interfaces** lohnt sich fast immer!
  - Eine Schnittstelle lässt verschiedene Implementationen zu.
  - Minimal: **Echte** Implementation und **Test**-Implementation
- Wahl der gewünschten Implementationen muss zur (Test-)Laufzeit beeinflusst werden können.
  - Per →**Dependency Injection** (manuell oder per Framework).
- Achtung: Es ist eine «Sicherung» nötig, dass das **nicht** in der Produktion passiert!

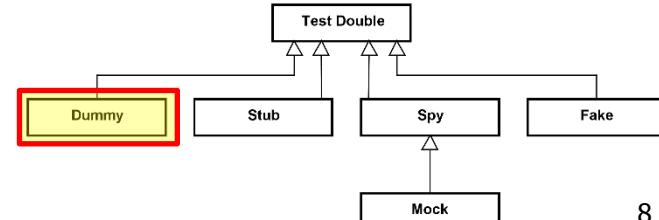
# Test-Doubles - Übersicht

- Es gibt **Dummy**, **Stub**, **Spy**, **Mock** und **Fake**



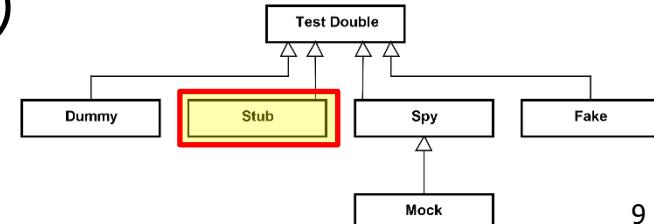
# Test-Doubles: Dummy

- **Dummy** (engl. *dummy* «unecht», «funktionslos», «Attrappe»)
- Sehr primitive und (häufig) **leere** Ersatz-Implementation, die als aktueller Parameter an Methoden übergeben wird.
  - Aktueller Parameter ist für den Test zwar notwendig, dessen Nutzung und Implementation (für den Test) aber irrelevant.
- Dummy dient zur (meist) **funktionslosen** Entkopplung der beim Test unerwünschten Abhängigkeiten.
- Beispiel:  
Einem Objekt muss z.B. ein Logger übergeben werden, der soll aber einfach **nichts** machen, weil das Loggen ist nicht das eigentliche Testziel.



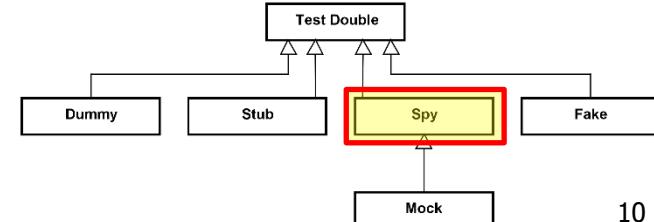
# Test-Doubles: Stub

- Stub (engl. *stub* «Stummel», «Stumpf», «Platzhalter»)
- Einfache Implementation, welche mit möglichst **geringem** Aufwand **sinnvolle, vordefinierte** Werte (z.B. Konstanten) zurückliefert.
- Erlaubt ein sogenanntes «**State**»-Testing.
  - State (Zustand) wird durch Daten repräsentiert.
  - State ist bei Stubs in der Regel **konstant**.
- Für die unterschiedlichen Testziele werden ggf. auch mehrere unterschiedliche Stubs (Implementationen) erstellt.
- Beispiel: Klasse für Authentifikation, welche...
  - Beliebige Benutzer / Passwörter akzeptiert (`login = true`)
  - Niemanden akzeptiert (`login = false`)



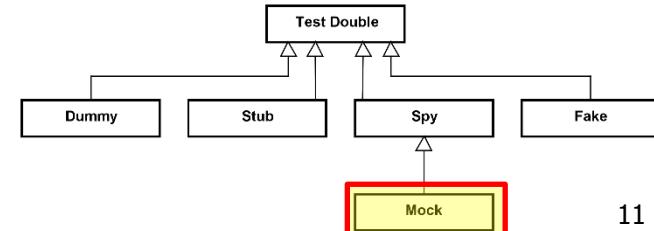
# Test-Doubles: Spy

- Spy (engl. *spy* «spionieren», «ausspähen», «Spion»)
- Alternative Implementation, welche **dynamische** Werte zurückliefern kann. Gleichzeitig **merkt** sich der Spy exakt die **Aufrufe** der Methoden!
  - Anzahl/Häufigkeit, Parameter, Zeitpunkt, Exceptions etc.
- **Nach** der Interaktion können die aufgezeichneten Ereignisse für die **Verifikation** des Testfalls genutzt werden.
- Erlaubt ein so genanntes «Behavior»-Testing (Verhalten).
- Beispiel: Wurde auf dem im Testkandidaten registrierten **ActionListener(-Spy)** die Methode **actionPerformed(...)** auch tatsächlich aufgerufen?



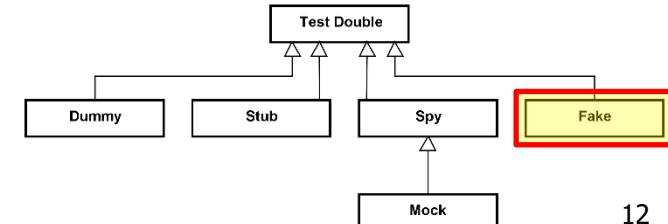
# Test-Doubles: Mock

- **Mockup** (engl. *mock*[ -*up*] «vorgetäuscht», «Simulation»)
- **Spezialisierung** des Spy, welche dynamische Werte zurückliefern kann und die korrekte Interaktion **selber** (→ Abgrenzung zum Spy) verifizieren kann.
- Mocks werden typisch mit Hilfe von speziellen **Mock-Frameworks** zur Laufzeit für **jeden** Testfall als **individuelle** Mock-Objekte (Proxy-Pattern, GoF) erstellt!
  - Verhalten wird dynamisch (für jeden einzelnen Testfall!) und somit **programmatisch** konfiguriert.
- Ein Mock ist dem Spy sehr ähnlich. Einziger Unterschied ist der **Ort der Verifikation**, Mocks sind dadurch spezifischer.



# Test-Doubles: Fake

- **Fake** (engl. *fake* «gefälscht», «künstlich», «nachgemacht»)
- **Alternative** Implementation, welche eine Komponente mit vernünftigem Aufwand **vollständig** (!) ersetzen kann.
- Ermöglicht die vollständige Entkopplung von einer Abhängigkeit.
  - Trade-off: Aufwand dessen Implementation muss in einem vernünftigem Verhältnis zum Nutzen sein (Unit vs. Integration).
- Beispiel: Abhängigkeit von Webservices wird durch eine lokale (Fake-)Implementation ersetzt.
  - Kommunikation fällt weg → schneller.
  - Implementation ist trotzdem vorhanden (wenn nicht zu komplex), idealerweise sogar wiederverwendet.



# **Empfehlungen**

# Empfehlung - Wann setzt man nun was ein?



## ▪ **Dummy und Stub:**

- Einfache Ersatzimplementationen um eine bessere Testisolation zu erreichen. Mit geringem Aufwand erreicht man eine höhere Selektivität und Stabilität der Testfälle. **Einfach!**

## ▪ **Spy und Mock:**

- «Universalwaffen» für Behavior-Testing mit Hilfe von Mocking-Frameworks. Diese können auch zur Realisierung von Stubs und Dummies genutzt werden. **Komplexer.**

## ▪ **Fake:**

- Eher aufwändige Implementation, zur vollständigen Entkopplung vom Original. Aufwand muss sich lohnen! **Aufwändig.**
- Was, wenn der Fake besser als das Original ist...?

# Beispiel für Java - Mocking mit Mockito

- Bewährtes Mocking-Framework für Java.
- Maven Dependency (natürlich im Scope **test**):



```
<dependency>
    <groupId>org.mockito</groupId>
    <artifactId>mockito-core</artifactId>
    <version>4.8.1</version>
    <scope>test</scope>
</dependency>
```

- Viele statische Funktionen auf der Klasse **org.mockito.Mockito**
- Am einfachsten ist ein statischer Import aller Methoden:  
**import static org.mockito.Mockito.\*;**
- Dokumentation siehe <http://site.mockito.org/>

# Empfehlungen



- Mocking-Frameworks sind kein goldenere Hammer!
  - Es gibt Klassen die sind **zu aufwändig** für Mocking.
  - Es gibt Klassen die sind **zu einfach (!)** für Mocking!
  - Einsatz mit gesundem Augenmass.
- Die Verständlichkeit des Testcodes steht an hoher (erster) Stelle:  
Wenn ein Testfall durch Mocking so kompliziert wird, dass man ihn nicht mehr versteht, oder sich nicht mehr getraut ihn anzufassen, hat man verloren.
- Überlegen Sie immer gut, ob es sich lohnt. Steigen Sie langsam in die Technik ein, sie ist absolut faszinierend!

# Kritik an Mocking Frameworks

- Nur für **Dummies** und **Stubs** braucht man vielfach **kein** Mocking-Framework. **Keep it simple, stupid!** (KISS)
  - Auch hier ein Trade-off: Man kann auch zu viel mocken.
  - Ein direkt implementierter Dummy (oder Stub) ist nicht selten einfacher zu verstehen.
- Das Design entscheidet!
  - Verwende so oft wie möglich Interfaces!
  - Damit kann sehr schnell alternative Implementation integrieren.
- Uncle Bob sagt sinngemäss:  
*«Ich verwende selten Mocking, die sind mir zu kompliziert. Es ist häufig einfacher es selber zu machen.»*

Uncle Bob (rechts) mit  
einem anonymen Clean  
Coder (links)



© Ruedi Arnold

# Quellen 1

- Martin Fowler: Mocks Aren't Stubs -  
<http://martinfowler.com/articles/mocksArentStubs.html>
- Robert C. Martin (Uncle Bob): The Little Mocker –  
<http://blog.8thlight.com/uncle-bob/2014/05/14/TheLittleMocker.html>
- xUnit Patterns - <http://xunitpatterns.com/>
- Mocking-Frameworks:
  - Mockito - <http://site.mockito.org/> (Empfehlung!)
  - EasyMock Framework - <http://easymock.org/>
  - jMock Framework - <http://jmock.org/> (alt)

**Fragen?**

Verteilte Systeme und Komponenten

# **Continuous Integration (CI)**

Roland Gisler

# Inhalt

- Was ist Continuous Integration?
- Die zehn Praktiken der CI
- Technische Umsetzung

# Lernziele

- Sie können die Ziele der **Continuous Integration (CI)** erklären
- Sie kennen die zehn wesentlichen Praktiken der CI.

# Continuous Integration

- Continuous Integration ist  
der Ausweg aus der «Integrationshölle»!



# Continuous Integration (CI)

Hauptziele bei der Entwicklung von Software nach CI:

- Immer ein **lauffähiges** Produkt (Buildresultat) zu haben.
  - Es kann somit kontinuierlich getestet werden.
- Bei Fehlern jeder Art möglichst schnell ein Feedback zu erhalten.
  - Primär durch automatisierte Unit- und Integrationstests.
  - Aber auch durch Kompiler, Classpath, statische Codeprüfung etc.
- Im Team parallel entwickeln zu können und dennoch...
  - den Überblick nicht zu verlieren.
  - den Integrationsaufwand zu minimieren.
  - über den aktuellen Zustand auf dem Laufenden zu sein.
- Moderne, **zeitgemäße**, agile Software-Entwicklung!
- Grundidee: Kent Beck (XP, JUnit) und Martin Fowler (Refactoring)

# **Die 10 Praktiken der CI**

# Übersicht: Die 10 Praktiken der CI – nach Martin Fowler

1. Einsatz eines (zentralen) Versionskontrollsysteem.
2. Automatisierter Buildprozess.
3. Automatisierte Testfälle.
4. Alle Ändern den Quellcode auf dem Hauptzweig\*.
5. Bei einer Änderung wird automatisch ein Build durchgeführt.
6. Der Buildprozess muss schnell sein.
7. Auf/mit Kopien der produktiven Umgebung testen.
8. Einfacher Zugriff auf aktuelle Buildartefakte.
9. Offensive Information über den aktuellen Zustand.
10. Automatisches Deployment\*.

# 1 - Versionskontrollsystem

- Grundlagen: siehe Input → [EP 11 Versionskontrolle](#).
- Sämtliche Quellen-Artefakte welche für den vollständigen Build einer Software benötigt werden unterliegen der Versionskontrolle.  
→ «Alles was für den Build benötigt wird,  
aber nichts was erneut gebaut (build) werden kann.»
- Nutzen Sie die Fähigkeiten des Versionskontrollsysteems:
  1. Sinnvolle Commit-Kommentare vergeben.
    - Ideal: Mit Hinweis auf **Issue#** eines Issue Tracking Systems.
  2. Tagging – markieren von bestimmten Versionen.
    - Für die einfache, eindeutige Identifikation von Releases.
  3. Branches – Zweige für parallele Entwicklung.
    - Ermöglicht z.B. die parallele Weiterentwicklung, einfaches Bugfixing, (möglichst kurzlebige!) Feature-Banches etc.

## 2 - Automatisierter Buildprozess

- Grundlagen: siehe Input → [EP 12 Buildautomatisation](#)
- Build auf einer kontrollierten, «sauberen» Maschine durchführen.
  - Einsicht: Entwickler\*innen-Maschinen sind **nie** sauber... ☺
- Ausschliesslich auf der Basis der aktuellen Quellen aus dem Versionskontrollsystem (VCS).
  - Dadurch stellt man z.B. sehr schnell und einfach fest, ob man vergessen hat etwas wesentliches einzuchecken.
- Inklusive Ausführung der Testfälle und QS-Metriken.
  - Laufen die Unit-Tests tatsächlich immer und überall?
  - Gibt es keine Seiteneffekte durch parallele Codeänderungen?
  - Ist die Codequalität gut genug?

### 3 - Automatisierte Testfälle

- Grundlagen: siehe Input → [EP 22 AutomatisiertesTesting](#)
- Möglichst viel durch automatisierte Testfälle abdecken.
  - Primär Unit Tests, weil einfach und überall lauffähig.
  - Sekundär auch Integrationstests, z.B. Abhängig von Datenbank.
- Fehlerhafte oder nicht vollständige Implementationen sollen so schnell wie möglich aufdecken werden.
  - Bei Integrationstests auch häufig unerwartete Nebeneffekte.
- Bewährt haben sich auch Performance-Tests.
  - Wirkt sich ein neues Feature negativ auf die Performance aus?
- Primäres Ziel: Tests müssen immer laufen bzw. im Fehlerfall so schnell wie möglich wieder gefixt werden!
  - Gemeinsames Ziel für das **ganze Team**.

## 4 - Änderungen auf dem Hauptzweig des VCS

- Ursprünglich wurde gefordert, dass alle Entwickler\*innen auf dem einzigen Hauptzweig (HEAD, trunk, master, main etc.) arbeiten.
  - Die Motivation ist, dass sämtliche Änderungen möglichst schnell (und kontinuierlich) in die **einige** Codebasis integriert werden.
    - Bei **grossen** Teams (welche entsprechend viele Änderungen produzieren) führt das aber zu sehr vielen «merges».
  - Mit moderneren VCS, welche «billige» Branches anbieten, begann man darum leichtfertig für jede Entwickler\*in einen eigenen Branch zu eröffnen, um wieder autonomer arbeiten zu können.
- Das führt aber wieder zu einem sehr aufwändigen «BigBang», wenn diese Branches dann vor dem Release in den Hauptzweig gemerged werden (müssen)!!
- Man fällt wieder in die «Integrationshölle» zurück!
- 

## 4.1 - Änderungen auf dem Hauptzweig – Ergänzung

- Es wurden verschiedene Branch-Konzepte entwickelt, die einen vernünftigen Kompromiss zwischen autonomer Arbeit, und trotzdem genügend häufiger Integration ermöglichen.
- Das populärste und bekannteste Prinzip: **GitFlow**
  - <https://nvie.com/posts/a-successful-git-branching-model/> Mittlerweile von zahlreichen Tools integriert, um die (nicht immer trivialen) Abläufe zu automatisieren und vereinfachen.
- Empfehlung: **GitFlow** ist sehr gut, es lohnt sich aber oft, sich ein (ggf. vereinfachtes, siehe z.B. [GitHub-Flow](#)) und an das konkrete Projekt bzw. Team adaptiertes Konzept zu überlegen!
  - Teamgrösse, Projektgrösse, Modularisierung, Dynamik etc.
- Ziel: Häufige Integration, bei möglichst wenig Overhead!

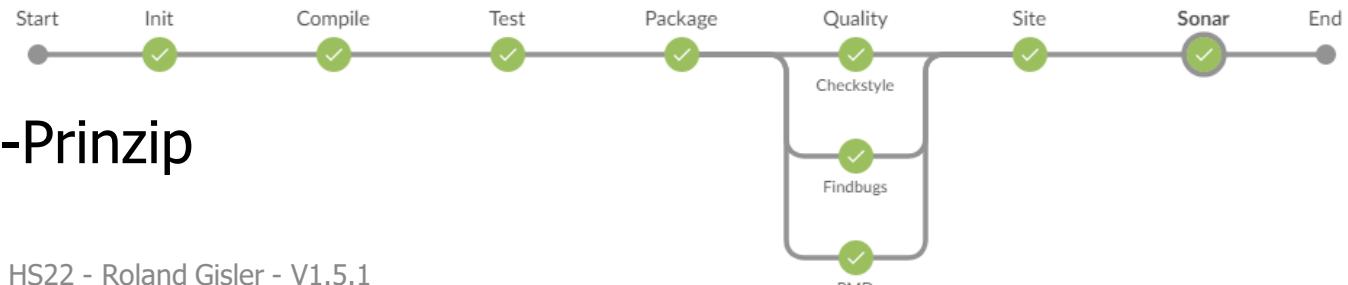
## 5 - Automatischer Build bei Änderungen



- Buildserver, siehe Input → [EP 14 Buildserver](#).
  - Voraussetzung: siehe Input → [EP 12 Buildautomatisation](#).
- Buildserver prüft regelmässig auf Veränderungen im Versionskontrollsystem (poll), bzw. wird vom SCM aktiv informiert (push).
  - Stellt er solche fest, werden die Quellen ausgecheckt und ein neuer Build gestartet.
- Alle Resultate des Build werden offensiv kommuniziert.
  - Erfolg, Testfälle, Laufzeit, Codechecks, Metriken – einfach alles!
- Ziel für das Team: Bricht ein Build (aus welchen Gründen auch immer), konzentriert man sich in erster Priorität und gemeinsam darauf, den Build wieder «**grün**» zu kriegen!
  - Proaktive Notifikation an alle beteiligten Entwickler\*innen.

## 6 - Buildprozess muss schnell sein

- Je schneller die Entwickler\*innen ein Feedback bekommen, dass etwas nicht mehr läuft, je besser!
- Natürlich muss ein Kompromiss gefunden werden – manche Tests benötigen mehr Zeit.
  - Werden somit kaum lokal durchgeführt.
  - Umso wichtiger ist es, diese im zentralen Build laufen zu lassen!
- Alternative: Zwei (oder mehr) gestaffelte Builds durchführen.
  - Schneller «continuous build für Feedback an Entwickler\*innen.
  - Langsamer (weil umfangreicherer) «nightly build» über Nacht.
- Noch flexibler mit Build-Chains oder Pipelines: Sequenz von Builds.



## 7 - Auf realer Umgebung testen

- Die zentrale Build- und Testumgebung sollte möglichst ähnlich zur produktiven Umgebung sein.
  - Ideal: Kopie der produktiven Umgebung → Teuer!
  - Alternative: Leichtgewichtige Virtualisierung z.B. mit Docker.
- Das umfasst z.B. folgende Punkte:
  - Hardwareausstattung, Betriebssystem, Laufzeitumgebung (Java etc.), Netzwerkzugriff (Kommunikation mit Drittsystemen)
  - Datenqualität und **Datenmenge**
  - Technologien wie Docker helfen uns hier extrem!
- In der Realität bei kleinen Systemen gut zu erreichen, bei grossen Systemen aber häufig viel zu teuer.
- Weitere Herausforderung: Datenschutz!

## 8 - Einfacher Zugriff auf Buildartefakte

- Sämtliche Buildresultate sollten jederzeit für eine weitere Nutzung zur Verfügung stehen.
- So dass einfach und schnell die neusten Versionen z.B. weiterführend getestet werden können.
  - Hat wiederum ein schnelleres Feedback zur Folge.
- Wird typisch über Buildserver erreicht, welche die Buildresultate selber archivieren können.
  - Achtung: Grosse Datenmengen möglich!
- Zusätzlich können die binären (ausführbaren) Artefakte zusätzlich noch in ein binäres Repository deployed.
  - z.B. Maven Repository, Sonar Nexus, Artifactory etc.
  - siehe Input → [EP 13 DependencyManagement](#)

## 9 - Offensive und offene Information

- Es gibt keine Geheimnisse!
- Für jede Entwickler\*in ist jederzeit einsehbar welche Änderungen...
  - ...von wem und wann eingecHECKT wurden.
  - ...von welchem Build erstmals erfasst wurden.
  - ...zu welchen Ergebnissen geführt haben (Build, Tests).
  - ...zu welchem Issue gehören.
  - ...welche Massnahmen getroffen wurden.
- Transparenz und Nachvollziehbarkeit: Offene Information nicht zur Kontrolle, sondern zur gemeinsamen Unterstützung!
  - Echtes «**collective code ownership**» als Ziel.
- Das Team hat ein **gemeinsames** Ziel:  
Erfolgreiche Builds und eine möglichst fehlerfreie Software!

# 10 - Automatisches Deployment

- Wenn immer möglich sollte das Buildergebnis auch automatisch verteilt und installiert werden.
    - Installation auf einem repräsentativen Zielsystem.
  - Oder zumindest regelmässig automatisch aktualisieren.
    - z.B. Täglich, über Nacht etc.
    - Es macht wenig Sinn alle fünf Minuten eine Server-Applikation zu installieren, wenn ein manueller Test eine Stunde dauert...
  - Damit steht die aktuelle Software sofort wieder für weiterführende, (z.B. manuelle) Systemtests bereit.
    - Hat wiederum ein schnelleres Feedback zur Folge.
    - Vermeidet Meldung von Fehlern, die schon behoben sind.
- **DevOps**-Methoden, infrastructure-as-code, Container, Cloud...

# **CI im Modul VSK**

# CI im Modul Verteile Systeme und Komponenten (VSK)

-  1. Einsatz eines (zentralen) Versionskontrollsystems.
-  2. Automatisierter Buildprozess.
-  3. Automatisierte Testfälle.
-  4. Alle Ändern den Quellcode auf dem Hauptzweig.
-  5. Bei einer Änderung wird automatisch ein Build durchgeführt.
-  6. Der Buildprozess muss schnell sein.
-  7. Auf/mit Kopien der produktiven Umgebung testen.
-  8. Einfacher Zugriff auf aktuelle Buildartefakte.
-  9. Offensive Information über den aktuellen Zustand.
-  10. Automatisches Deployment.

# Technische Umsetzung von CI

- ✓ ▪ Moderne, zeitgemäße Entwicklungsumgebung (nicht nur IDE!)
- ✓ ▪ Automatisierter, reproduzierbarer Buildprozess.
- ✓ ▪ Buildserver-Technologie.
- Ideal: Gegenseitige Vernetzung und Integration aller Tools.
- ✓ ▪ Offene Information an alle Beteiligten.
- Positiver Teamgeist, gemeinsames Ziel: Lauffähige Software!

# Logger-Projekte: Jetzt aber!

- Projekte sollten immer «grün» (ohne Fehler baubar) sein.
- Zwischen- und Schlussabgabe: **GRÜN** ist Pflicht und Ehre!

S	W	Name	Last Version	# QG	# CS	# PMD	# SB	Zeilenabdeckung	Branchabdeckung	Letzte Status	Letzte Dauer	Cause
✓	✗	g00-loggerinterface	1.0.0-SNAPSHOT	6	1	0	0	0.0%	100.0%	14 Tage	1 Minute 3 Sekunden	++ □ ▷
✓	✗	g01-demoapp	1.0.0-SNAPSHOT	4	3	0	1	0.0%	100.0%	5 Tage > 1 Monat	33 Sekunden	↑ □ ▷
✓	✗	g01-logger	1.0.0-SNAPSHOT	151	87	12	14	57.75%	62.5%	5 Tage > 1.3 Monate	1 Minute 36 Sekunden	↑ □ ▷
✓	✗	g01-stringpersistor	1.0.0-SNAPSHOT	60	50	6	2	84.62%	94.44%	5 Tage > 1.3 Monate	54 Sekunden	++ □ ▷
✓	✗	g02-demoapp	1.0.0-SNAPSHOT	0	0	0	0	85.71%	100.0%	2.2 Tage > 1.3 Monate	12 Minuten	↑ □ ▷
✓	✗	g02-logger	1.0.0-SNAPSHOT	959	535	272	18	0.11%	0.0%	2.2 Tage > 5.8 Tage	14 Minuten	++ □ ▷
⚠	✗	g02-stringpersistor	1.0.0-SNAPSHOT	50	35	1	8	78.79%	87.5%	5.8 Tage > 15 Tage	47 Sekunden	++ □ ▷
✓	✗	g03-demoapp	1.0.0-SNAPSHOT	0	0	0	0	85.71%	100.0%	1.9 Stunden > 1.3 Monate	46 Sekunden	↑ □ ▷
✓	✗	g03-logger	1.0.0-SNAPSHOT	25	17	0	1	2.38%	100.0%	1.9 Stunden > 17 Tage	1 Minute 41 Sekunden	++ □ ▷
✓	✗	g03-stringpersistor	1.0.0-SNAPSHOT	31	26	0	4	0.0%	0.0%	16 Tage > 1.3 Monate	53 Sekunden	++ □ ▷
✓	✗	g04-demoapp	1.0.0-SNAPSHOT	4	2	0	1	64.86%	100.0%	4.9 Tage > 5 Tage	36 Sekunden	↑ □ ▷
✓	✗	g04-logger	1.0.0-SNAPSHOT	56	40	5	1	0.0%	0.0%	4.9 Tage > 1.3 Monate	1 Minute 31 Sekunden	++ □ ▷
✓	✗	g04-stringpersistor	1.0.0-SNAPSHOT	0	0	0	0	100.0%	100.0%	13 Tage > 1.3 Monate	1 Minute 14 Sekunden	++ □ ▷
✗	✗	g05-demoapp	1.0.0-SNAPSHOT	-	-	-	-	82.76%	100.0%	1.9 Tage > 12 Tage	37 Sekunden	↑ □ ▷
✓	✗	g05-logger	1.0.0-SNAPSHOT	85	46	5	4	58.62%	31.25%	1.9 Tage > 4.1 Tage	11 Minuten	++ □ ▷
✓	✗	g05-stringpersistor	1.0.0-SNAPSHOT	56	43	3	4	86.05%	87.5%	11 Tage > 1.3 Monate	1 Minute 4 Sekunden	++ □ ▷
✓	✗	g06-demoapp	1.0.0-SNAPSHOT	0	0	0	0	85.71%	100.0%	13 Tage > 1.3 Monate	45 Sekunden	↑ □ ▷
✓	✗	g06-logger	1.0.0-SNAPSHOT	11	5	1	1	5.13%	100.0%	13 Tage > 1.3 Monate	1 Minute 42 Sekunden	++ □ ▷
✓	✗	g06-stringpersistor	1.0.0-SNAPSHOT	13	9	0	3	0.0%	0.0%	14 Tage > 1.3 Monate	51 Sekunden	++ □ ▷
✓	✗	g07-demoapp	1.0.0-SNAPSHOT	0	0	0	0	85.71%	100.0%	5.9 Tage > 1.3 Monate	54 Sekunden	↑ □ ▷
✓	✗	g07-logger	1.0.0-SNAPSHOT	458	36	314	9	0.14%	0.0%	5.9 Tage > 1.3 Monate	2 Minuten 19 Sekunden	++ □ ▷
✓	✗	g07-stringpersistor	1.0.0-SNAPSHOT	27	21	0	1	80.3%	76.67%	12 Tage > 1.3 Monate	1 Minute 10 Sekunden	++ □ ▷
✓	✗	g08-demoapp	1.0.0-SNAPSHOT	0	0	0	0	85.71%	100.0%	22 Stunden > 1.3 Monate	44 Sekunden	↑ □ ▷
✓	✗	g08-logger	1.0.0-SNAPSHOT	85	52	3	2	3.75%	0.0%	22 Stunden > 20 Tage	1 Minute 46 Sekunden	++ □ ▷
✓	✗	g08-stringpersistor	1.0.0-SNAPSHOT	58	39	1	8	76.47%	78.57%	13 Tage > 13 Tage	47 Sekunden	++ □ ▷
✓	✗	g09-demoapp	1.0.0-SNAPSHOT	0	0	0	0	85.71%	100.0%	2.3 Stunden > 1.3 Monate	44 Sekunden	↑ □ ▷
✓	✗	g09-logger	1.0.0-SNAPSHOT	48	28	3	10	0.97%	0.0%	2.3 Stunden > 2.5 Tage	1 Minute 39 Sekunden	++ □ ▷
✓	✗	g09-stringpersistor	1.0.0-SNAPSHOT	107	88	4	10	16.83%	10.0%	7.2 Tage > 1.3 Monate	1 Minute 2 Sekunden	++ □ ▷
✓	✗	g10-demoapp	1.0.0-SNAPSHOT	1	0	0	1	0.0%	0.0%	7.1 Tage > 19 Tage	42 Sekunden	↑ □ ▷
✗	✗	g10-logger	1.0.0-SNAPSHOT	-	-	-	-	62.03%	60.0%	3.8 Stunden > 7.1 Tage	2 Minuten 3 Sekunden	++ □ ▷
✓	✗	g10-stringpersistor	1.0.0-SNAPSHOT	13	11	1	1	88.57%	100.0%	13 Tage > 19 Tage	1 Minute 5 Sekunden	++ □ ▷
✓	✗	g11-demoapp	1.0.0-SNAPSHOT	2	2	0	0	57.14%	100.0%	4.8 Tage > 1.3 Monate	37 Sekunden	↑ □ ▷
✓	✗	g11-logger	1.0.0-SNAPSHOT	82	60	1	1	43.94%	25.0%	4.8 Tage > 1.3 Monate	1 Minute 34 Sekunden	++ □ ▷
✓	✗	g11-stringpersistor	1.0.0-SNAPSHOT	29	26	0	2	80.0%	83.33%	15 Tage > 1.3 Monate	1 Minute 7 Sekunden	++ □ ▷

- Stichprobe: Mittwoch, 2. November 2022 – **ok!**

# Zusammenfassung

- Continuous Integration nach Martin Fowler:  
Anwendung von zehn konkreten Praktiken.
- CI darf/soll niemals Selbstzweck sein!
- Pragmatische Anpassung an die Bedürfnisse des jeweiligen Projektes und Teams.
- Gewinn und Nutzen von CI steht im Vordergrund:  
Kollektive Bearbeitung des Quellcodes, immer ein lauffähiges Produkt, Nachvollziehbarkeit der Entwicklung.
- Grundlage für zeitgemäße iterative und inkrementelle Entwicklung von Software.

**Fragen?**

Verteilte Systeme und Komponenten

# Fehlertoleranz und Resilienz

Martin Bättig



# Inhalt

- Definitionen und Begriffe
- Fehlertolerante Systeme
- Resilienz durch Wiederherstellbarkeit
- Verteilte Transaktionen

# Lernziele

- Sie kennen allgemeine Begriffe und Definitionen im Zusammenhang mit Fehlertoleranz und Resilienz.
- Sie wissen, an welchen Stellen beim Senden und Empfangen einer Message Fehler auftreten können.
- Sie wissen, was Idempotenz ist und wie Sie diese bei der Auslieferung von Messages ausnutzen können.
- Sie kennen die verschiedenen Ausliefergarantien für Messages.
- Sie kennen die Möglichkeiten um eine exactly-once Auslieferung zu realisieren.
- Sie verstehen das Prinzip der verteilten Transaktionen und ihre Anwendung bei der messageorientierten Kommunikation.

# **Definitionen und Begriffe**

# Definitionen

- **Verfügbarkeit:** System für sofortigen Einsatz bereit. Beschreibt einen Zeitpunkt. Ausgefallene oder ausgeschaltete Systeme sind nicht verfügbar.
- **Hochverfügbarkeit:** System ist mit sehr hoher Wahrscheinlichkeit für Einsatz bereit.
- **Zuverlässigkeit:** Zeitintervall innerhalb dessen ein System verfügbar ist.
- **Betriebssicherheit:** Gibt an, ob ein Ausfall eines Systems zu katastrophalen Ereignissen führen kann.
- **Wartbarkeit:** Wie schnell kann ein ausgefallenes System wieder hochgefahren werden.

# Definitionen

**Fehler:** Ursache einer Funktionsstörung.

**Fehlertoleranz:** System kann trotz Vorkommnissen von Fehlern seine Dienste anbieten.

**Resilienz:** Widerstandsfähigkeit gegenüber vorhersehbaren bekannten Fehlern.

**Resilienz durch Redundanz:**

- ⇒ Schutz gegen Systemausfall: Systeme oder Daten mehrfach vorhanden.  
(wird im Teil «Konsistenz und Replikation» behandelt).
- ⇒ Schutz gegen byzantinische Fehler (korrupte Daten) -> erfordert Consensus-Protokolle (komplexes Thema, nur kurze Einführung).

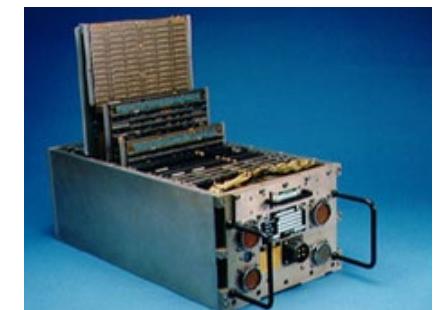
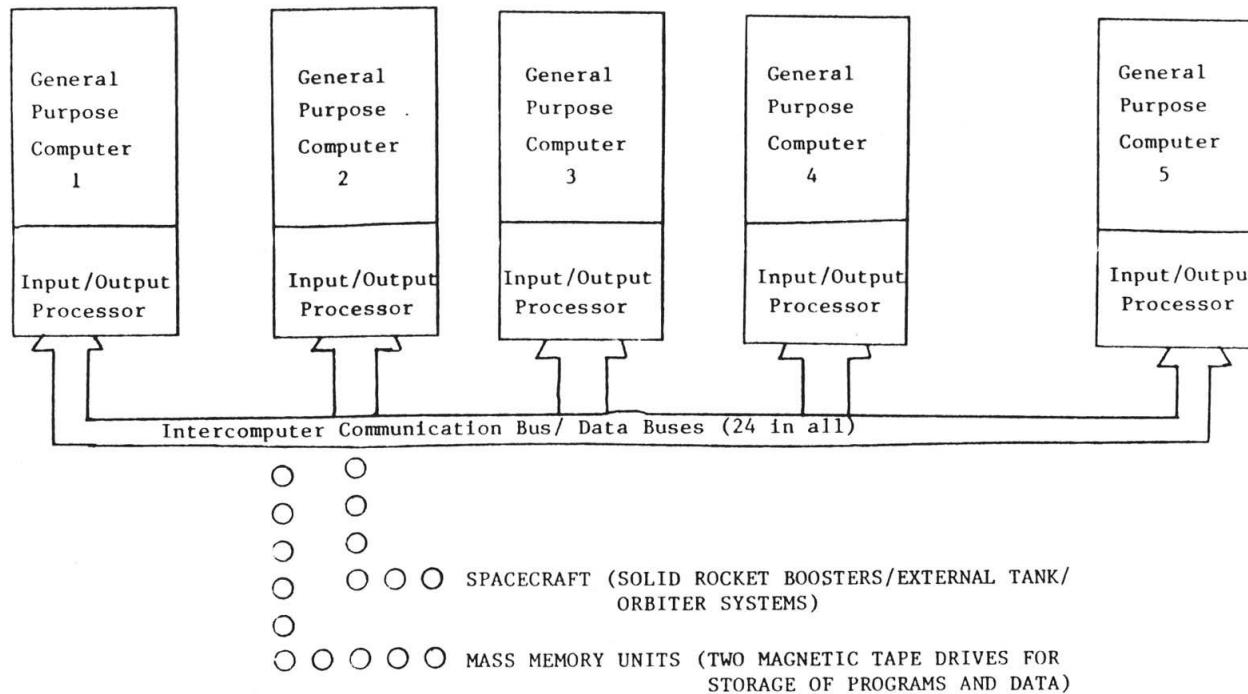
**Resilienz durch Wiederherstellbarkeit:**

- ⇒ System kann nach Ausfall inert kurzer Zeit wiederhergestellt werden und operiert exakt wie vor dem Ausfall (-> Wartbarkeit).

# Consensus-Protokolle

**Consensus (dt. Konsens):** Bei unterschiedlichen Ausgaben redundanter Systeme muss festgelegt werden, welche Ausgabe verwendet wird.

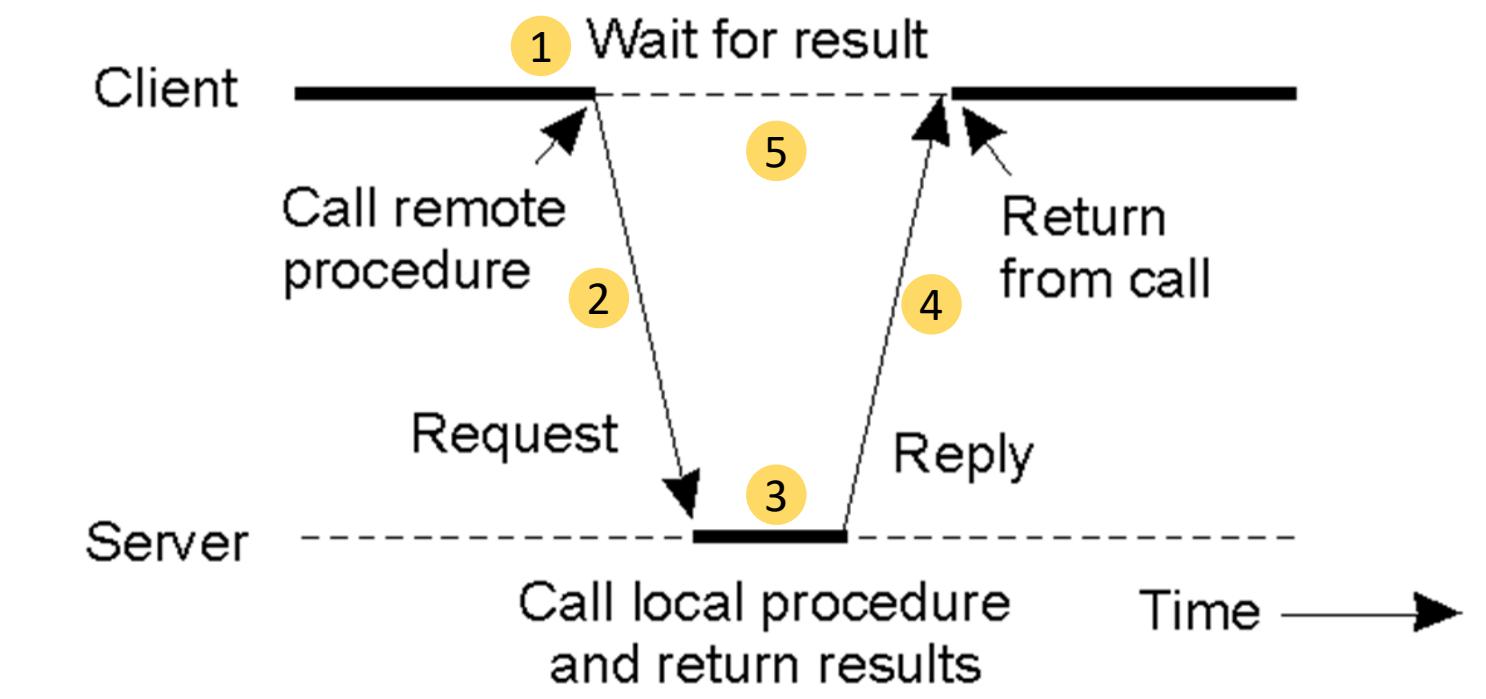
**Beispiel:** Vernetzte Steuerungscomputer des Space-Shuttles: Vier identische Computer und einer mit unterschiedlicher Programmierung.



# Fehlerarten bei verteilten System

- **Server nicht erreichbar:**
  - Daten / Messages können nicht verschickt werden.
- **Auslassungsfehler: Falsche Reihenfolge der Daten / Messages:**
  - Von TCP zuverlässig verhindert (mit welchem Mechanismus?).
- **Abrupter Verbindungsabbruch durch Netzwerkausfall:**
  - Folge: Verlorene Daten (z.B. Messages).
- **Abrupter Verbindungsabbruch durch Systemausfall:**
  - Folge: Verlorene Daten (z.B. Messages) oder inkonsistente Daten.

# Potentielle Fehler bei einem synchronen Aufruf (z.B. RPC)

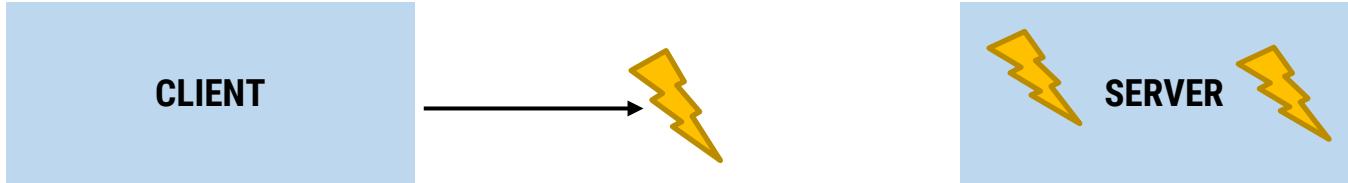


1. Keine Verbindung
2. Request geht verloren
3. Server crasht
4. Reply geht verloren
5. Client crasht

# **Fehlertolerante Systeme**

# Fehlersituation: Server nicht erreichbar (ohne Redundanz)

- Die Gegenstelle ist nicht erreichbar **VOR** Absenden einer Message:



Entweder Verbindung unterbrochen oder Server läuft nicht (identischer Effekt für Client).

**Erkennung:** Aufbau der Verbindung schlägt fehl.

## Massnahmen:

- Kein Betrieb möglich (z.B. Terminal-Server) => User informieren.
- Reduzierter Betrieb, falls Server nur Zusatzfunktionalität anbietet.
- Verwendung eines lokalen Caches (Zwischenspeicher).

# Verwendung eines lokalen Caches

- **Beim Lesen:** Beziehe Information aus vorangehender Kommunikation.
  - **Beim Schreiben:** Führe Schreib-Operationen lokal durch, sende an Server sobald verfügbar.
- ⇒ Achtung «Merge-Konflikte» analog z.B. VCS, falls mehrere Teilnehmer oder Geräte Änderungen vornehmen können.

# EchoClient mit lokalem Cache: CachingEchoHandler als Thread

```
public static class CachingEchoHandler implements Runnable {  
    Queue<String> queue = new ConcurrentLinkedQueue<>();  
    ZContext context;  
    ZMQ.Socket socket;  
  
    public CachingEchoHandler() {  
        context = new ZContext();  
        socket = context.createSocket(SocketType.REQ);  
        socket.connect(ADDRESS);  
    }  
  
    public void run() {  
        try {  
            while (true) {  
                sendAndReceiveMessage(waitForNextMessage());  
            }  
        } catch (InterruptedException e) {  
            throw new RuntimeException(e);  
        }  
    }  
}
```

Erstelle lokale Queue

ZeroMQ-Socket erstellen

Warte auf Message in lokaler Queue.

# EchoClient mit lokalem Cache: Queue and Sending

```
private void sendAndReceiveMessage(String message) {  
    socket.send(message.getBytes(ZMQ.CHARSET));  
    byte[] bytes = socket.recv();  
    System.out.println(new String(bytes, ZMQ.CHARSET));  
}
```

Message senden  
und empfangen  
(blockierend)

```
public void addMessageToQueue(String s) {  
    queue.offer(s);  
    synchronized (this) { this.notify(); }  
}
```

Message zu lokaler Queue  
hinzufügen

```
private String waitForNextMessage() throws InterruptedException {  
    String message;  
    while ((message = queue.poll()) == null) {  
        synchronized (this) { this.wait(); }  
    }  
    return message;  
}
```

Warte bis neue  
Message in lokaler  
Queue eintrifft.

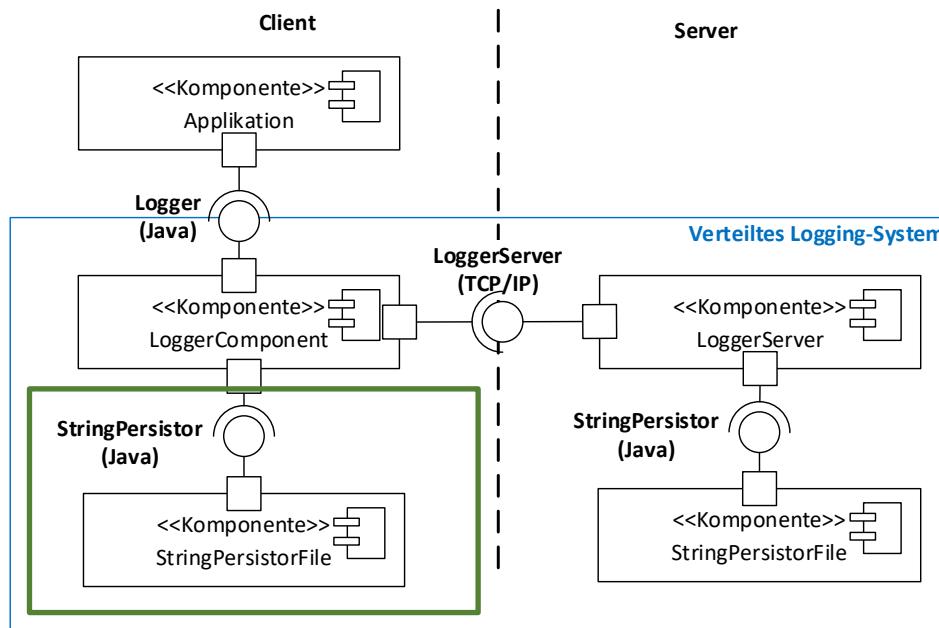
# CachingEchoClient - Main

```
public static void main(String[] args) throws IOException {  
    CachingClientHandler cachingEchoHandler = new CachingClientHandler();  
  
    Starte ClientEchoHandler als Thread, da Empfang von  
    Messages blockieren wird, falls EchoServer nicht verfügbar.  
    new Thread(cachingEchoHandler).start();  
  
    LOG.info("CachingEchoClient running on " + ADDRESS);  
    BufferedReader userIn =  
        new BufferedReader(new InputStreamReader(System.in));  
  
    while(true) {  
        String input = userIn.readLine();  
        cachingClientHandler.addMessageToQueue(input);  
    }  
}
```

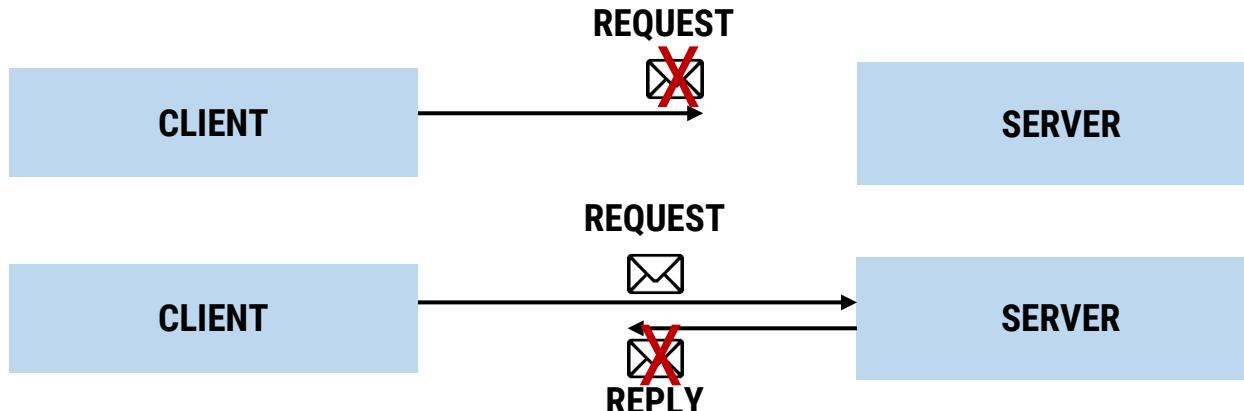
# Übung: Lokaler Cache im verteilten Logger-System

*Diskutieren Sie in Ihrem Team, wie Sie mittels einem lokalen Cache mit folgenden Situationen möglichst transparent umgehen können:*

- 1) Fehlgeschlagener initialer Verbindungsaufbau, nach ca. 10 Sekunden steht die Verbindung.
- 2) Fehlgeschlagener initialer Verbindungsaufbau, Server kommt während des ganzen Logging-Vorgangs nicht mehr online.
- 3) Abrupter Verbindungsabbruch, mit Wiederherstellung der Verbindung nach ca. 10 Sekunden. Messages können verloren gehen.



# Fehlersituation: Verlorene Message (Request oder Reply)



## Erkennung:

- Client startet Timer bei Absenden eines Requests.
- Ist nach Ablauf des Timers noch keine Antwort eingetroffen, gilt die Message als verloren.
- Unterscheidung eines **verlorenen Requests** von einem **verlorenen Reply**?

## Massnahmen:

- Benutzer Informieren / Fragen z.B. bei interaktiven Verbindungen.
- Falls sinnvoll, Request nochmals senden => ggf. Duplikatscheck

# Auslieferungsgarantien

- **At-least-once:** Message wird sofort gesendet bis eine Antwort eintrifft.
  - Problem: Aktion wird möglicherweise doppelt ausgeführt.
- **At-most-once:** Message wird höchstens einmal gesendet.
  - Problem: Aktion wird möglicherweise nicht ausgeführt.
- **Exactly-once:** Message wird exakt einmal gesendet.
  - Gewünscht, aber möglicherweise zu teuer oder unnötig.

# Transparenz mittels Idempotenz

## Idempotente Funktion (math.):

Funktion, welche auf sich selbst angewandt das identisches Resultat ergibt:

$$f(x) == f(f(x))$$

## Beispiel für eine idempotente Funktion:

- Rückgabe des absoluten Werts:  $f(x) = |x|$

## Beispiel für eine nicht-idempotente Funktion:

- Rückgabe der Negation:  $f(x) = -x$

# **Idempotente Anfragen**

- Anfragen sind idempotent, wenn die Funktion auf der Gegenseite idempotent ist.
- Wichtig: eine idempotente Anfrage darf keine Nebeneffekte wirken.
- Idempotente Anfragen können i.d.R. ohne Probleme wiederholt werden!

# Beispiele für idempotente Anfragen

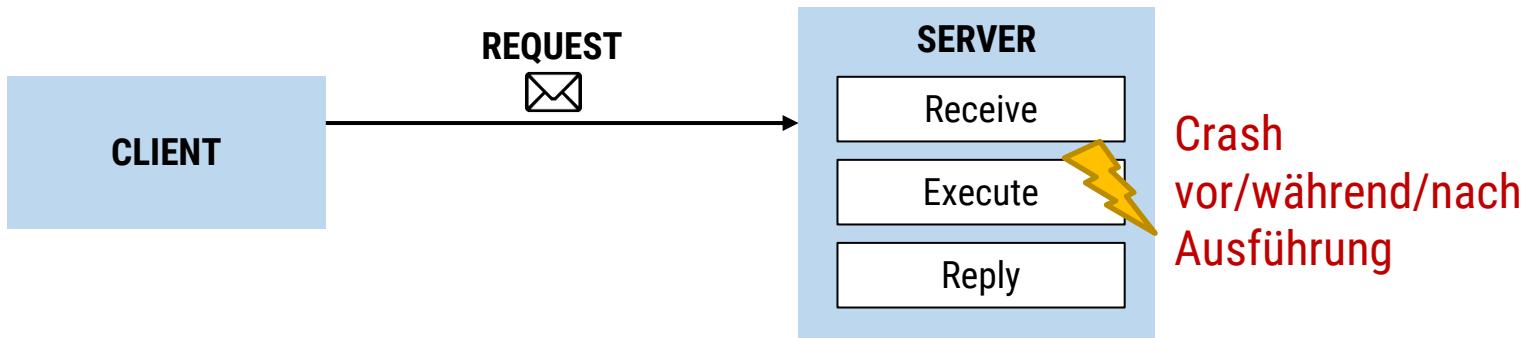
- Read-Requests: z.B. Abfrage nach einem Fahrplan.
  - Achtung: Falls Lese-Zugriffe protokolliert werden (z.B. bei Banken).
- Ändere Adresse von Ort A zu Ort B (ohne weitere Nebeneffekte).
  - Achtung bei parallelen Anpassungen, ggf. Zeit mitsenden.

## Nicht idempotent:

- Bestellung / Reservation / etc. (typisch: "erstelle neues Element").
- Lösche File F, wenn F in Zwischenzeit wieder erstellt werden kann.

# **Resilienz durch Wiederherstellbarkeit**

# Fehlersituation: Server-Crash während Requestverarbeitung



## Erkennung:

- Client: Keine Unterscheidung gegenüber verlorener Message möglich.
- Server kann über Aktionen ein Log führen und dieses beim Restart abarbeiten (zusätzliche Kosten durch Diskzugriff).

## Massnahme:

- Falls vor Ausführung (== verlorener Request): Client kann Message erneut senden.
- Falls während Ausführung: Konsistenz wieder herstellen.
- Falls nach Ausführung: Reply-Senden.
  - **Achtung:** Client könnte erneute Anfrage gesendet haben => Duplikat.

# Fehlersituation: Client-Crash während Warten auf Antwort



Entweder Verbindung unterbrochen oder Client läuft nicht (identischer Effekt).

## Erkennung:

- Server: Keine Verbindung zu Client möglich.

## Massnahme:

- Antwort speichern, falls Client identifizierbar (=> z.B. mittels Token).
  - **Achtung:** Client könnte erneuten Request stellen => Duplikatserkennung.
- Antwort verwerfen, falls Client nicht identifizierbar.
- Problematisch, z.B. falls ein Client Ressourcen blockieren kann.  
**Beispiel:** File-Locking im NFS (Network File System).

# Herausforderung: Erkennung von Duplikaten

Entweder durch:

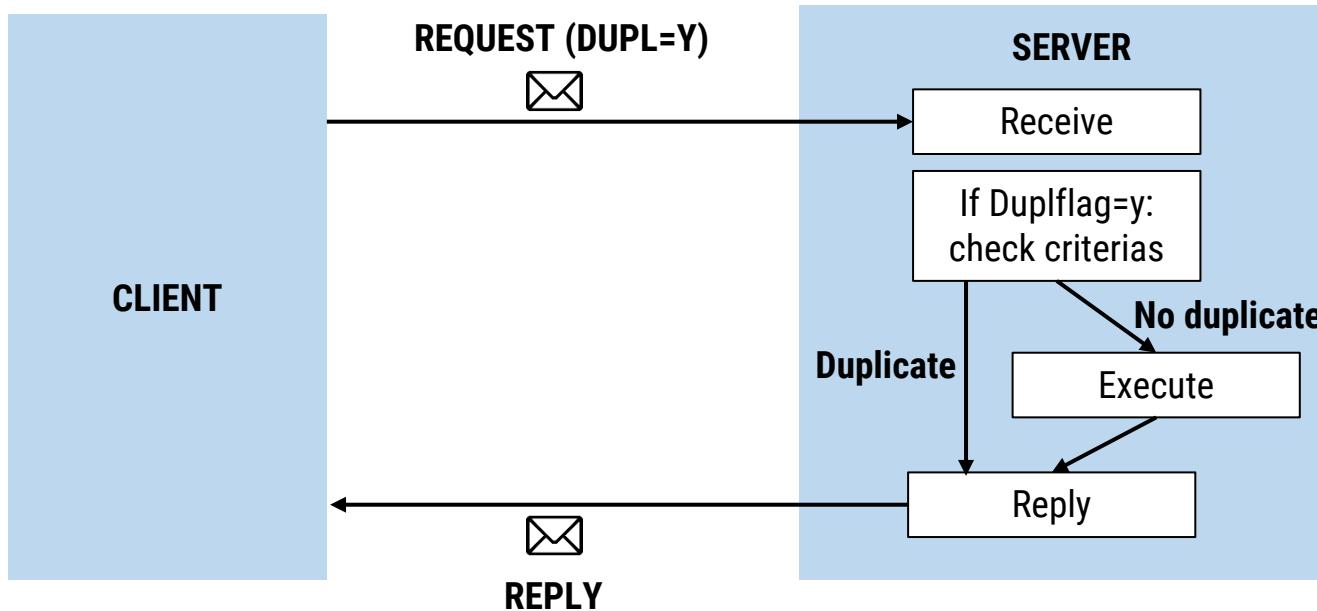
- **Heuristik (\*)**: falls eine gewisse Fehlerwahrscheinlichkeit tolerierbar ist.
- **Sequenznummer**: falls alle Duplikate erkannt werden sollen.

(\*) Heuristik: Methode um mittels **unvollständigem Wissen** trotzdem zu praktikablen Ergebnissen zu kommen.

# Erkennung von Duplikaten mittels Heuristik

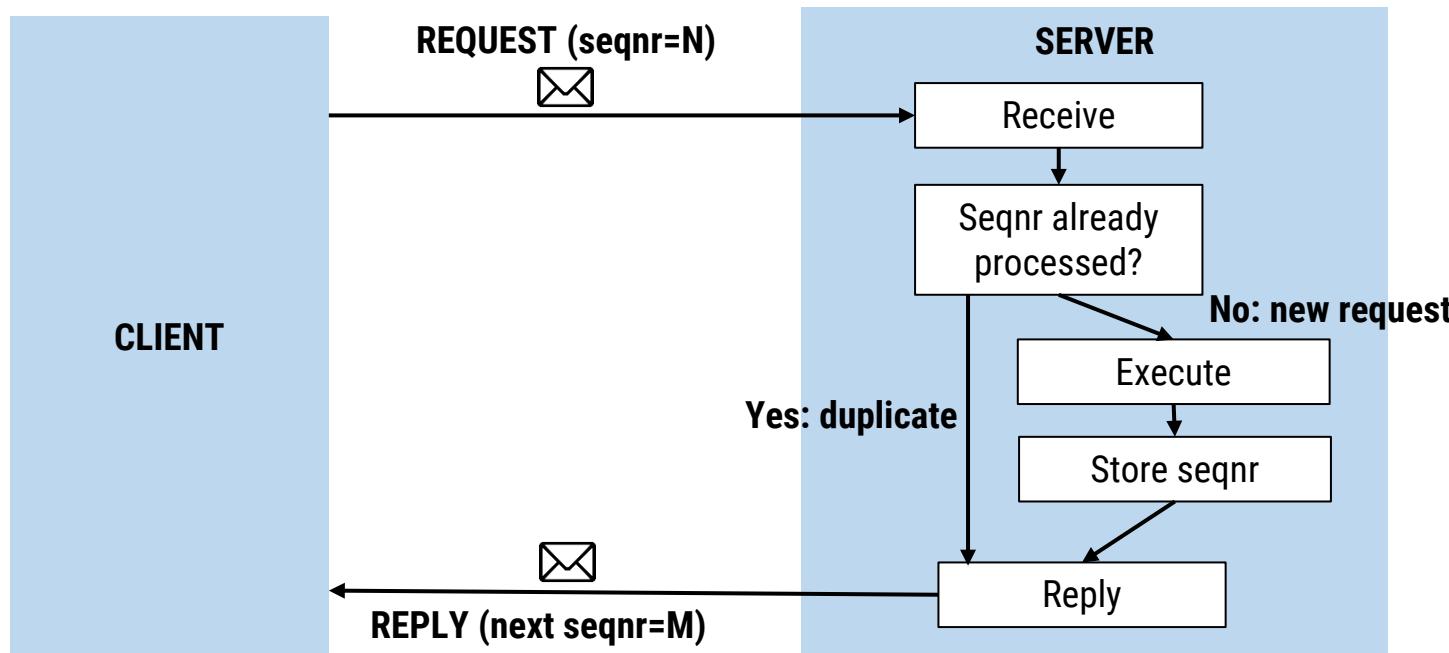
Mit gewisser Wahrscheinlichkeit durch Heuristiken:

- Duplikatsflag: Client gibt an, dass ein Request wiederholt wird.
- Verwendung diverser Kriterien (Kundennummer, Betrag, Ort, usw.).



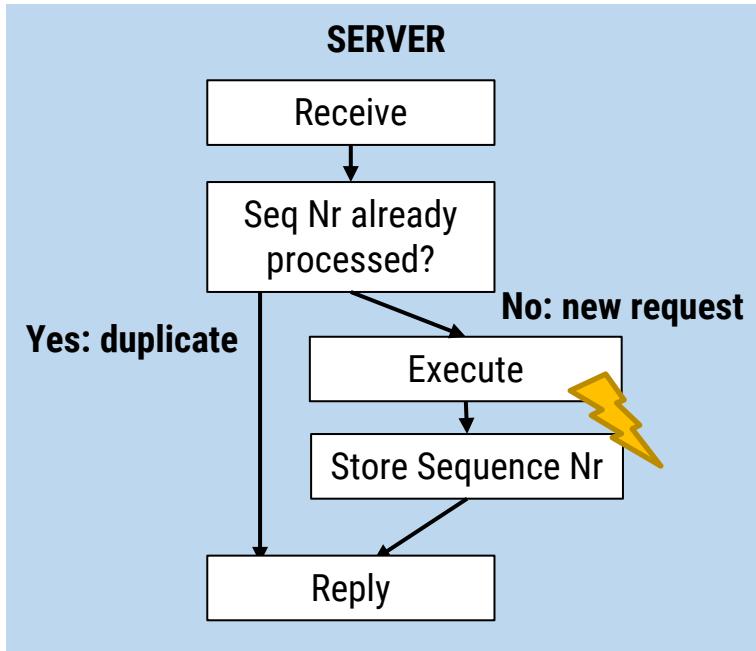
# Exakte Erkennung von Duplikaten mittels Sequenznummern

- Jeder Request erhält eine Sequenznummer.
- Server muss sich nach oder vor Ausführung die Sequenznummer (seqnr) merken (Kosten: Zugriff auf persistenten Speicher).
- I.d.R. grosse Nummer -> erhöht Grösse der Messages.
- **Mögliches Vorgehen:** Server wird bei jeder Antwort die nächste Sequenznummer vorgeben (Request-Response).

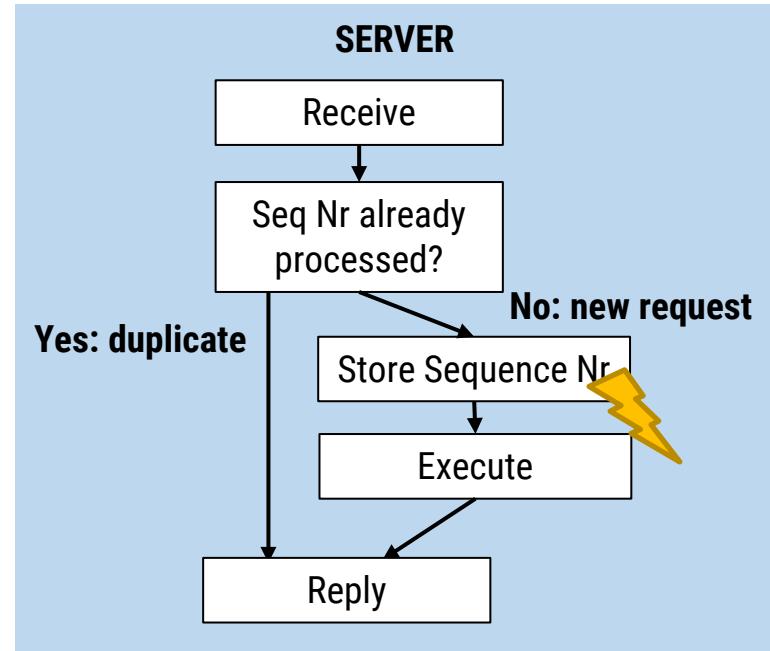


# Server-Crash: Fehlende Konsistenz

- Wie sicherstellen, dass Sequenznummer N nur als verarbeitet markiert wird, wenn der Request ausgeführt wurde?
- Andere Reihenfolge hilft nicht:



Crash vor "Store Sequence Nr":  
Aktion ausgeführt, aber nicht als  
ausgeführt markiert.  
=> Doppelte Ausführung



Crash vor Execute:  
Nicht ausgeführt, aber als  
ausgeführt markiert.  
=> Keine Ausführung

# Transaktionen zur Lösung des Konsistenzproblems

## Transaktion:

- Atomare Einheit der Ausführung: Entweder alles ausgeführt oder nichts.
- Typischerweise innerhalb von Datenbanken angewendet und unterstützt (Oracle / Postgres / MariaDB / H2 / DB2 / MongoDB / etc.).

## Ablauf:

- Transaktion starten.
- Mehrere zusammengehörige Operationen innerhalb der Transaktion durchführen (Daten abfragen, einfügen, modifizieren, löschen).
- Transaktion entweder:
  - **erfolgreich abschliessen (commit):** Alle Operationen werden ausgeführt  
**ODER**
  - **abbrechen (rollback):** Keine Operation wird ausgeführt.

# Transaktionen zur Lösung des Konsistenzproblems (forts.)

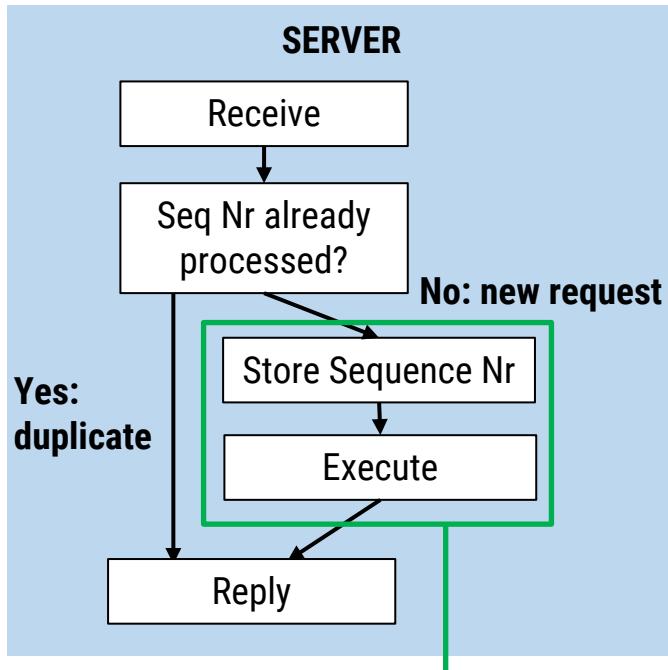
**Beispiel:** Geldtransfer von 100 CHF von Konto A zu Konto B.

1. Transaktion T starten
2. Account A um 100 CHF reduzieren
3. Account B um 100 CHF erhöhen
4. commit von T

Falls Crash vor Schritt 4 => Keine Aktion ausgeführt.

# Lösung des Konsistenzproblems mit Transaktionen

**Vorgehen:** Verwendung einer Datenbank D, welche Transaktionen unterstützt.



## Vorgehen (Skizze):

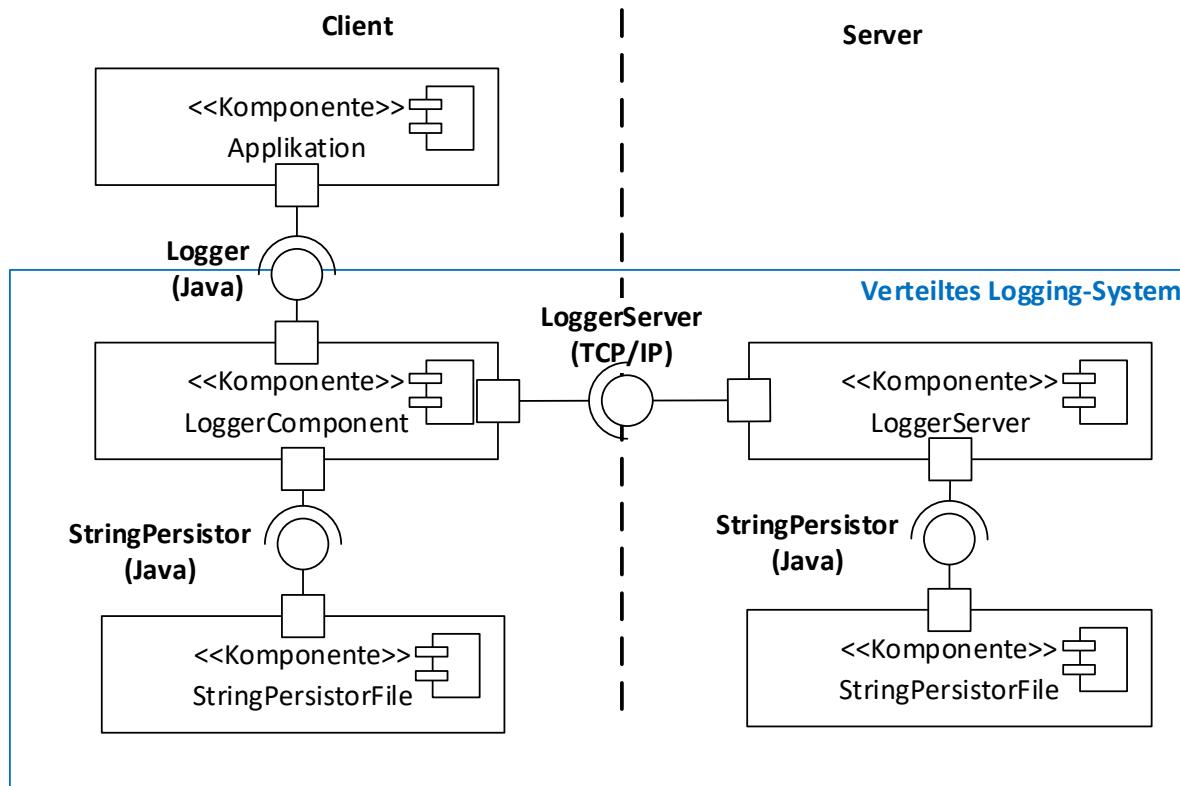
1. Starte Transaktion T.
2. Verarbeitung speichert alle Änderungen in Datenbank D.
3. Speichere Sequenznummer in Datenbank D.
4. Schliesse Transaktion T ab.

Ausführung als Transaktion:  
Entweder beide Aktionen  
ausgeführt oder keine.

# Übung: Exactly-once Auslieferungsgarantie im verteilten Logger

Diskutieren Sie in Ihrem Team, wie Sie eine Exactly-once Auslieferungsgarantie im verteilten Logger-System realisieren könnten (Hypothetisch, keine Anforderung):

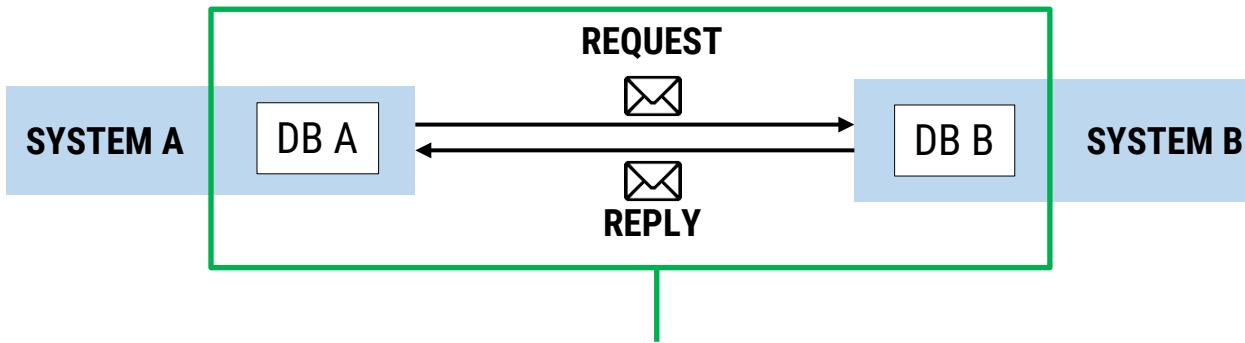
- 1) Wie müssten Sie das Protokoll ändern?
- 2) Hätte dies Einfluss auf die Komponentenaufteilung oder Schnittstellen?



# **Verteilte Transaktionen**

# Verteilte Transaktionen

- **Ziel:** Kombination von Transaktionen über zwei oder mehr Systeme hinweg.  
Alle Transaktionen werden entweder ausgeführt oder nicht.
- **Voraussetzung:** Jedes an der verteilten Transaktion teilnehmende System verfügt über einen Transaktions-Mechanismus, z.B. eine Datenbank (DB).



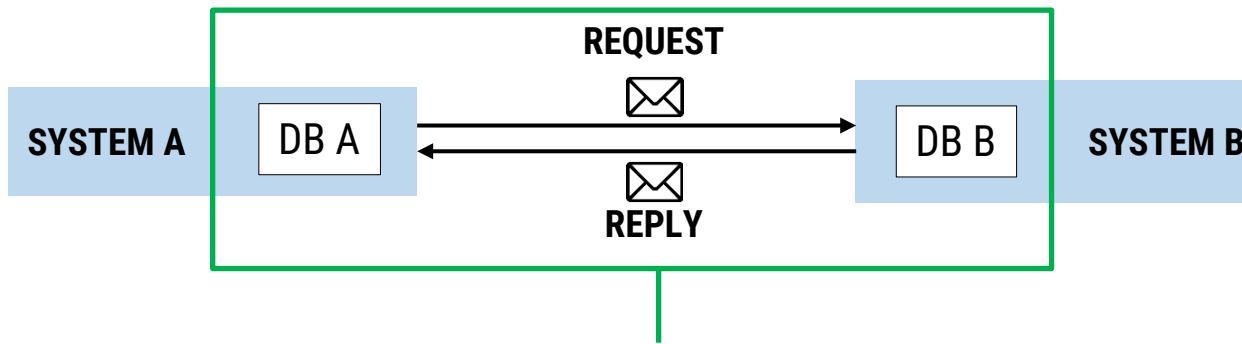
Verteilte Transaktion: Entweder  
beides ausgeführt oder nichts.

## Weiterführende Literatur:

Van Steen/Tannenbaum, Distributed Systems, Kapitel 8.5

# Kommunikation mit verteilten Transaktionen

- **Vorteil:** Einsatz von Sequenznummern usw. ist unnötig.
- ⇒ **Nachteil:** Höherer Ressourcenbedarf und Administrationsaufwand.
- ⇒ Achtung: Man muss transaktional programmieren (kein autocommit!).
- ⇒ Achtung: Bei System-Crashes können beide Systeme blockiert werden.



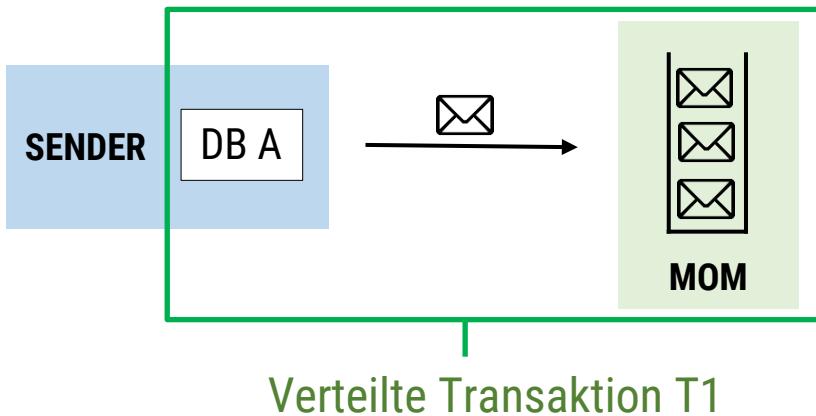
Verteilte Transaktion: Entweder beides ausgeführt oder nichts.

## Skizze:

System A: delete msg from outgoing where id = X  
System B: insert (msg) into incoming

# Verteilte Transaktionen mit persistenter Kommunikation

- Message-orientierte Middleware (MOM) unterstützt oft verteilte Transaktionen.
- **Ziel:** Zeitliche Entkopplung mit **exactly-once** Auslieferungsgarantie.

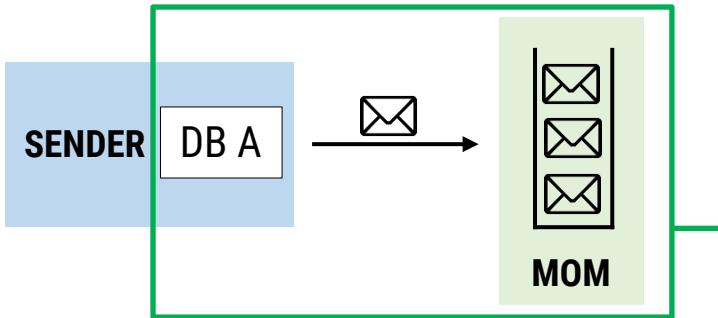


## Skizze der Transaktion:

- DB A: delete msg from outgoing where id = X
- MOM: Insert MSG

# Beispiel (Skizze): Exactly-once Auslieferung mittels MOM

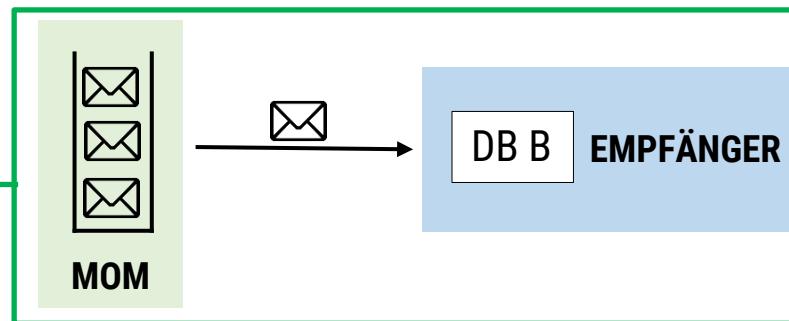
- Zuerst wird T1 ausgeführt, dann (ggf. Minuten später) T2:



Verteilte Transaktion T1

**Skizze der Transaktion:**

- DB A: delete msg from outgoing where id = X
- MOM: Insert MSG



Verteilte Transaktion T2

**Skizze der Transaktion:**

- MOM: Remove MSG
- DB B: insert (msg) into incoming

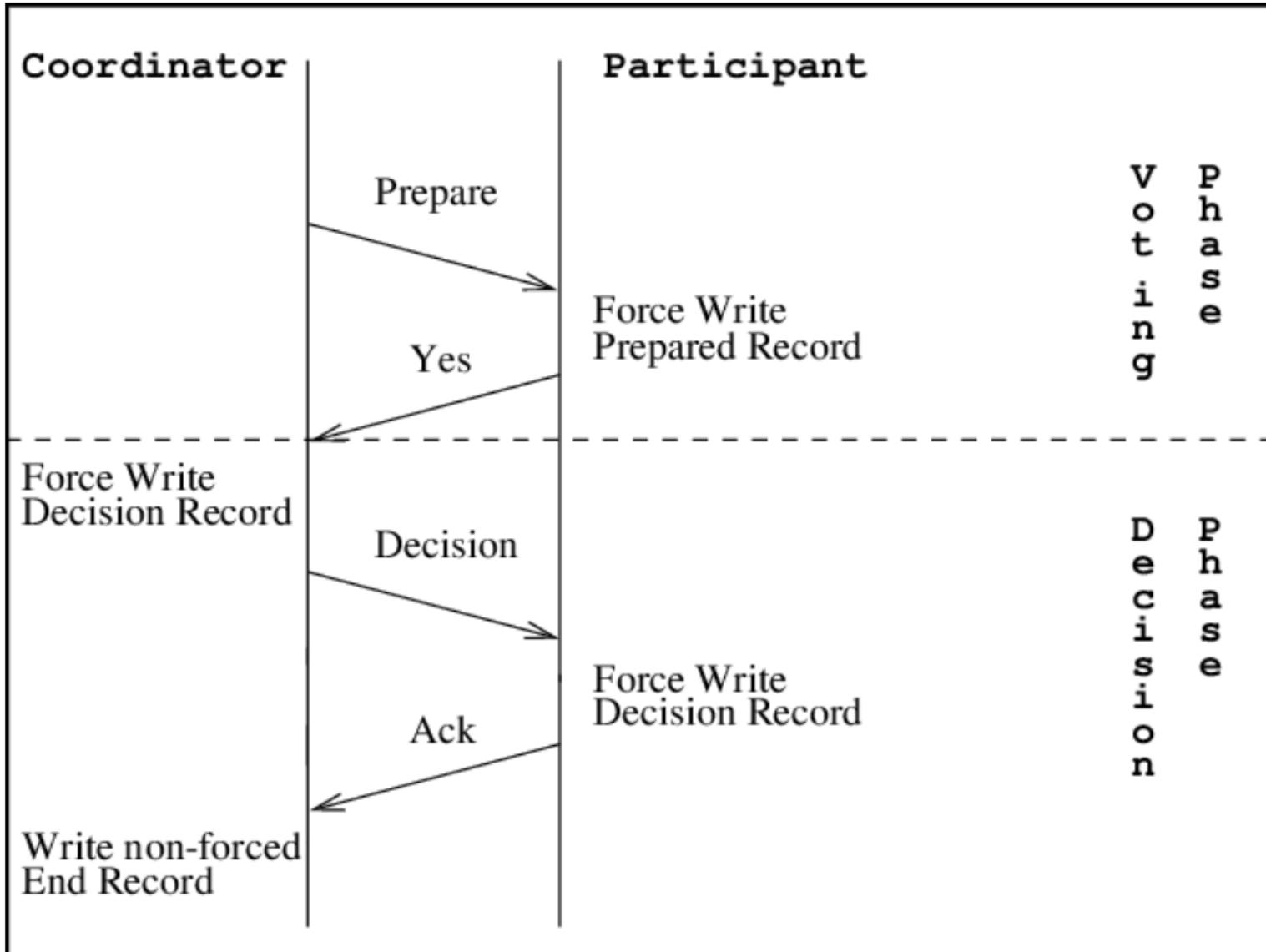
t

# Funktionsweise verteilter Transaktionen

**Two-Phase-Commit (2PC):** Typsicher Algorithmus für verteilte Transaktionen.

- Koordiniert wird 2PC vom einem Teilnehmer der verteilten Transaktion, z.B. der ersten Transaktion, welche ein Commit ausführt.
- **Erste Phase:** Koordinator fragt alle beteiligten Systeme, ob Sie die Transaktion erfolgreich abschliessen können (YES) oder nicht (NO).
- **Zweite Phase:** Koordinator trifft Entscheidung: Entweder alle commiten (nur YES erhalten), oder alle brechen die Transaktion ab (mindestens ein NO erhalten).

# Two-Phase-Commit: Ablauf



# Two-Phase Commit im Detail

## Voting Phase:

- Koordinator sendet eine PREPARE-Anfrage an alle Teilnehmer.
- Falls ein Teilnehmer eine PREPARE-Anfrage erhält, antwortet er entweder mit YES, falls er seine Transaktion abschliessen kann und will, ansonsten antwortet er mit NO.

## Decision Phase:

- Koordinator sammelt alle Antworten der Teilnehmer.
  - Falls alle Teilnehmer mit YES geantwortet haben, sendet er eine COMMIT-Anfrage an alle Teilnehmer.
  - Falls mindestens ein Teilnehmer mit NO geantwortet hat, sendet er eine ABORT-Anfrage an alle Teilnehmer.
- Jeder Teilnehmer, welcher mit YES geantwortet hat, wartet für auf die Anweisung des Koordinators.
  - Falls er eine COMMIT-Anfrage erhält, schliesst er die Transaktion erfolgreich ab.
  - Falls er eine ABORT-Anfrage erhält, mache Transaktion rückgängig.
- Falls Koordinator crasht, können andere Teilnehmer angefragt werden.

# Zusammenfassung

- Fehlertoleranz: System kann bestimmte Fehler tolerieren.
- Beim Senden einer Message können eine Vielzahl an Fehlern auftreten.
- Idempotente Messages können erneut gesendet werden.
- Auslieferungsgarantien je nach Anwendungsfall wählen: Exactly-once ist schwierig.
- Erkennung von Duplikaten für exactly-once Auslieferungsgarantie mittels Sequenznummer.
- Verteilte Transaktionen ermöglichen eine exactly-once Auslieferungsgarantie der Messagekommunikation und können mit Message-Oriented-Middleware kombiniert werden.
- Basis der verteilten Transaktionen ist der Two-Phase-Commit.

# Literatur

- Distributed Systems (3rd Edition), Maarten van Steen, Andrew S. Tanenbaum, Verleger: Maarten van Steen (ehemals Pearson Education Inc.), 2017.

# **Fragen?**

Verteilte Systeme und Komponenten

# **Entwurfsmuster**

Roland Gisler

# Inhalt

- Einführung - Was sind Entwurfsmuster?
- Gliederung der Entwurfsmuster in Kategorien
- Ausgewählte, einfache Beispiele, Teil 1
- Einsatz von Entwurfsmustern
- Ausgewählte, einfache Beispiele, Teil 2
- Ergänzende Hinweise zu Entwurfsmustern
- Zusammenfassung und Quellen

# Lernziele

- Sie verstehen die Vorteile beim Einsatz von Entwurfsmustern.
- Sie kennen verschiedene, ausgewählte Entwurfsmuster.
- Sie können konkrete Entwurfsmuster auswählen und gezielt einsetzen.

# **Einführung**

# Entwurfsmuster

- "Elemente wiederverwendbarer, objektorientierter Software."
- *oder*
- "Bewährte objektorientierte Entwürfe (Schablonen) für ein wiederkehrendes Entwurfsproblem."
- Massgeblich entwickelt und popularisiert von Erich Gamma, Richard Helm, Ralph Johnson und John Vlissides†, auch bekannt als die „**Gang of Four**“ (GoF).
- Resultate
  - Rund 20 dokumentierte Entwurfsmuster.
  - Buch „Entwurfsmuster“ (1995), ein echter Klassiker in der Informatik-Literatur.



# Erich Gamma

- Promovierte an der Universität Zürich
- Mitautor von Entwurfsmuster, Mitglied der GoF
- Mitentwickler von JUnit (mit Kent Beck)
- Aktuell:
  - Bei Microsoft als Distinguished Engineer tätig:  
Weiterentwicklung von Microsoft Visual Studio Code.
  - Lange als Distinguished Engineer bei Rational Software  
(Abteilung der IBM Software Group, mit Sitz in Zürich) tätig.
  - Leitete lange die Entwicklung der Eclipse Platform, auf der auch  
die populäre Eclipse JDT (IDE) basiert.



# **Wiederverwendung**

# **Wiederverwendung in der SW-Entwicklung**

- Wiederverwendung von bewährten Entwurfsmustern als Ziel.
- Verschiedene Arten von Wiederverwendung in der Softwareentwicklung:
  - Objekte zur Laufzeit wiederverwenden.
  - Wiederverwendung von Quellcode / Klassen.
  - Wiederverwendung von einzelnen Komponenten.
  - Einsatz von Klassen-Bibliotheken / Frameworks.
  - Wiederverwendung von Konzepten, z.B:
    - Entwurfs-, Architektur- oder Kommunikationsmuster...

# Wiederverwendung von Objekten

- Wiederverwendung von Objekten in einer Software während der Laufzeit.
- Beispiele:
  - Threads in einem **Executor-Pool**.
  - DB-Connection in einem Connection-Pool.
-  Effekt:
  - Bessere Performance, höhere Effizienz.
  - Geringerer Ressourcenbedarf.
- Herausforderung:
  - Effiziente Verwaltung der Objekte.
  - Hier können Entwurfsmuster bereits konkret helfen!

# Wiederverwendung von Quellcode/Klassen

- Wiederverwendung durch
  - Copy&Paste → schlecht! (vgl. Clean Code – [DRY](#) Prinzip)
  - Vererbung → häufig schlecht (!).
  - Aggregation und Komposition → Gut! (vgl. Clean Code – [FCoI](#))
-  Effekte:
  - Geringerer Entwicklungsaufwand.
  - Geringere Fehlerrate (Klassen sind bereits umfangreich getestet).
- Herausforderungen:
  - Schnittstellen der Klassen eher fremdbestimmt.
  - Auswahl der geeigneten Klassen/Bibliotheken.
  - Lernaufwand, Wartung und Weiterentwicklung.

# Wiederverwendung von Komponenten

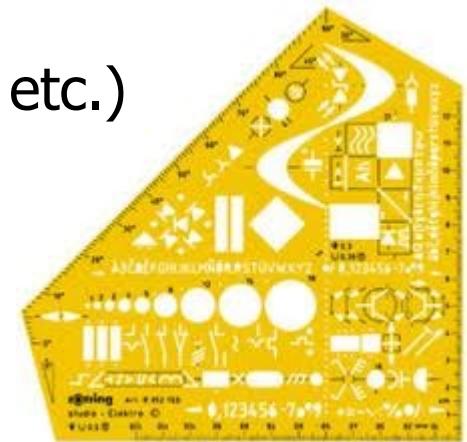
- Beispiele:
  - Logging-Komponente, Jakarta EE-Beans, Corba-Komponenten etc.
-  Effekte:
  - Geringerer Entwicklungsaufwand, weniger Fehler.
  - Ganzheitlicherer Ansatz, Blackbox, Abstraktion.
- Herausforderungen:
  - Anforderungen an die Umgebung / Kontext.
  - Eventuell inkompatible Schnittstellen → Entwurfsmuster!
  - Verwaltung der Komponenten (Konfigurationsmanagement).
  - Abhängigkeit vom Lieferanten.
  - Wartung und Weiterentwicklung.

# Herausforderung der (Quellcode-)Wiederverwendung

- Wiederverwendung ist sehr gut, bringt aber auch ein paar Herausforderungen mit sich:
  - Unterschiedliche Kontexte / Fachverständnisse.
  - Unterschiedliche Technologien / Lösungsansätze.
  - Einfache Weiterentwicklung und Wartung.
  - Aufwändiges Konfigurationsmanagement.
  - Verschiedene, inkonsistente Designkonzepte.
  - Zusätzliche Abhängigkeit von Dritten.
- **Wiederverwendung von Quellcode ist und bleibt eine grosse Herausforderung!**

# Alternative: Wiederverwendung von Konzepten

- Konzepte bleiben relativ konstant und stabil.
- Zusätzlich relativ breit abgestützt und erprobt, weil weitgehend Sprach- und Implementationsunabhängig!
- **Die Wiederverwendung von bewährten Entwurfsmustern ist eine sehr elegante, wirkungsvolle, unproblematische und kostensparende Form von Wiederverwendung!**
- Vergleiche:
  - Kommunikationsmuster (z.B. Handshaking)
  - Architekturmuster (z.B. C/S, Schichtung, MVC etc.)



# **Entwurfsmuster - Klassifikation**

# Klassifikation von Entwurfsmustern

- Entwurfsmuster werden primär nach Ihrem Zweck klassifiziert.  
Daraus sind drei Gruppen entstanden:
  - **Erzeugungsmuster** (Creational Patterns)
  - **Strukturmuster** (Structural Patterns)
  - **Verhaltensmuster** (Behavioral Patterns)
- Sekundäre Unterteilung:
  - **Klassenmuster**
    - Legen Beziehungen bereits zum Kompilierzeitpunkt fest.
  - **Objektmuster**
    - Beziehungen sind zur Laufzeit dynamisch veränderbar.

# Kategorie 1: Erzeugungsmuster

- Abstrahieren die Erzeugung von Objekten.
  - Entscheidung welcher (dynamische) Typ verwendet wird.
  - Entscheid über den Zeitpunkt der Erzeugung (z.B. lazy).
  - Entscheid auf welche Art das Objekt konfiguriert wird (Kontext, Initial-Konfiguration etc.).
- Delegation der Erzeugung an ein anderes Objekt.
  - 'Fabrik'-Konzept: Man fordert einfach ein Objekt an, die Details der Instanziierung und der Konfiguration interessieren (die Nutzer\*in) aber nicht.

# **Erzeugungsmuster - Übersicht**

- Abstrakte Fabrik (Abstract Factory, Kit)
- Erbauer (Builder)
- Fabrikmethode (Factory Method, Virtual Constructor)\*
- Prototyp (Prototype)\*
- Einzelstück (Singleton)\*

## Kategorie 2: Strukturmuster

- Fassen Objekte (oder Klassen) zu grösseren oder veränderten Strukturen zusammen.

*oder*

- Erlauben unterschiedliche Strukturen einander anzupassen und miteinander zu verbinden.

# **Strukturmuster - Übersicht**

- Adapter (Adapter, Wrapper)
- Brücke (Bridge, Handle/Body)
- Dekorierer (Decorator, Wrapper)
- Fassade (Facade)
- Fliegengewicht (Flyweight)
- Kompositum (Composite)\*
- Stellvertreter (Proxy, Surrogate)\*

## **Kategorie 3: Verhaltensmuster**

- Beschreiben die Interaktionen zwischen Objekten.
- Legen die Kontrollflüsse zwischen den Objekten fest.
- Zuständigkeit und/oder Kontrolle delegieren.

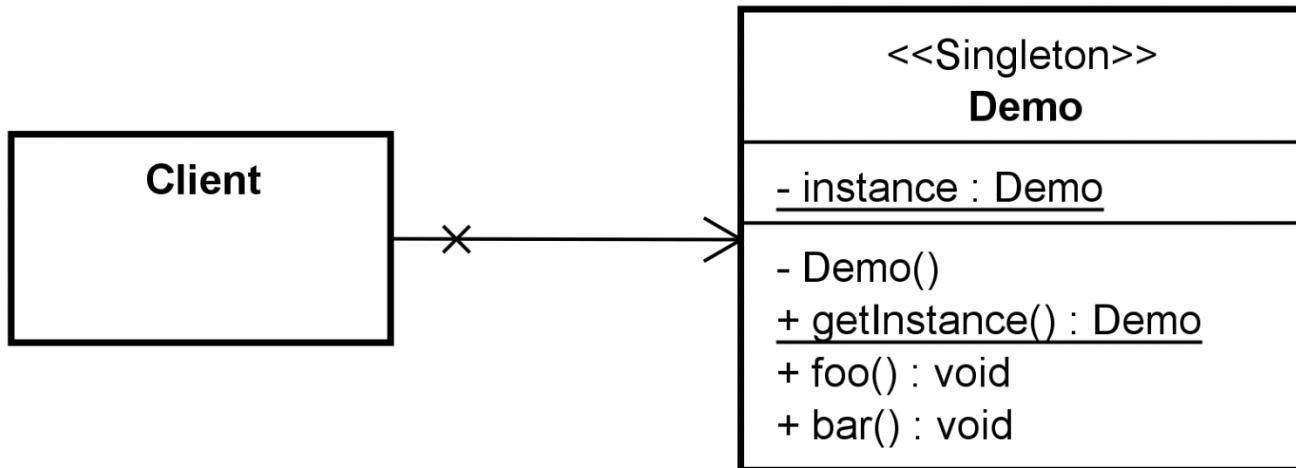
# Verhaltensmuster - Übersicht

- Befehl (Kommando, Command, Action, Transaction)
- Beobachter (Observer, Dependents, Publish/Subscribe, Listener)\*
- Besucher (Visitor)
- Interpreter (Interpreter)
- Iterator (Iterator, Cursor)\*
- Memento (Memento, Token)
- Schablonenmethode (Template Method)
- Strategie (Strategy, Policy)
- Vermittler (Mediator)
- Zustand (State, Objects for States)\*
- Zuständigkeitskette (Chain of Responsibility)

# **Singleton**

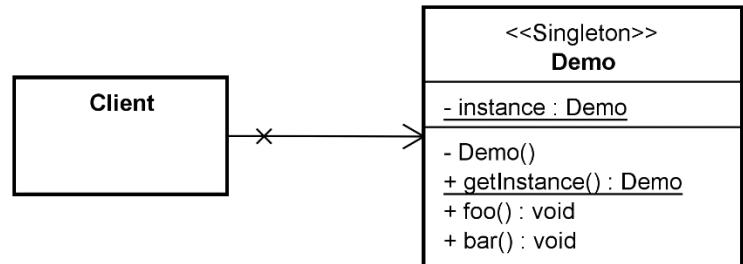
# Beispiel 1: Singleton

- Gewährleistet, dass von einer Klasse genau nur **eine einzige** Instanz (Objekt) erzeugt wird, und stellt für diese einen Zugriffspunkt zur Verfügung.



# Beispiel 1: Singleton

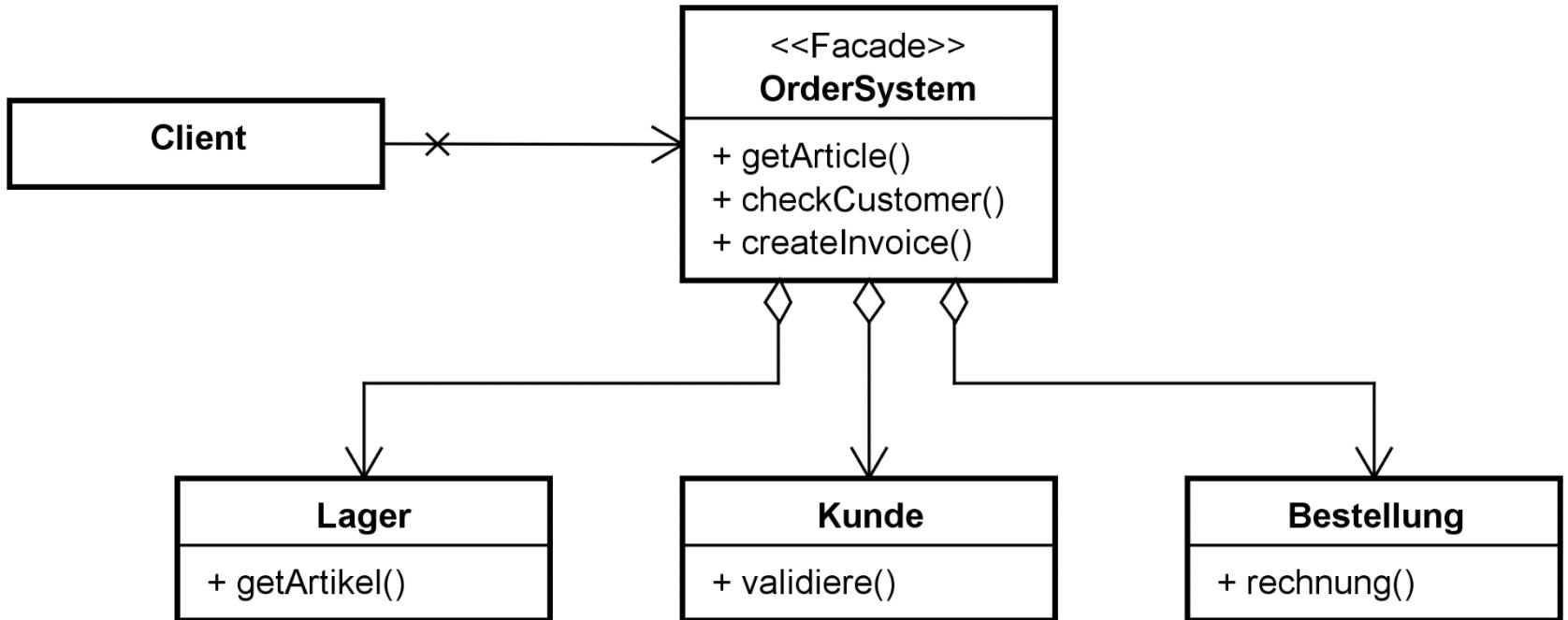
- Erzeugungsmuster, objektbasiert
- Wichtigste Eigenschaften der Implementierung:
  - Privates, statisches Attribut für Objektinstanz.
  - Öffentliche, statische Methode für Zugriff auf Objekt.
  - Privater Konstruktor (verhindert externe Instanziierung).
- Singleton hat mittlerweile einen schlechten Ruf:
  - Gamma bedauert, dieses Pattern propagiert zu haben.
  - Hauptkritik: Das Singleton führt zu einer starken Kopplung, ein späterer Austausch ist nur mit grossem Aufwand möglich.
- Empfehlung: Sehr zurückhaltend und gezielt einsetzen. Singleton **niemals** als universellen, globalen Zugriffspunkt verwenden.



# **Fassade**

## Beispiel 2: Fassade

- Stellt eine einheitliche, zusammengefasste Schnittstelle zu einer Menge von Schnittstellen mehrerer Subsysteme zur Verfügung.



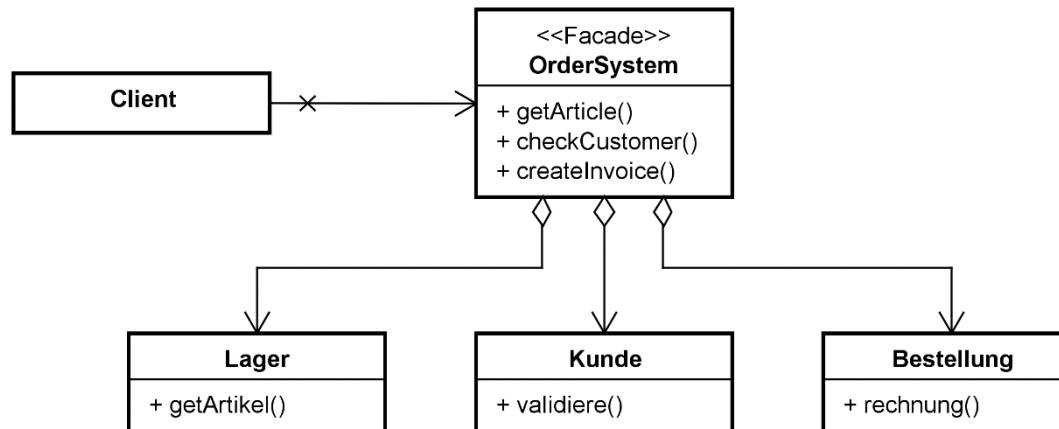
## Beispiel 2: Fassade - Teilnehmer

- Fassade – Beispiel: **OrderSystem**

- Weiss welche Subklassen für eine Anfrage zuständig sind und delegiert die Anfragen entsprechend weiter.
- Sorgt für eine konsistente Namensgebung der Methoden.
- Enthält ansonsten keine weitere Funktionalität!

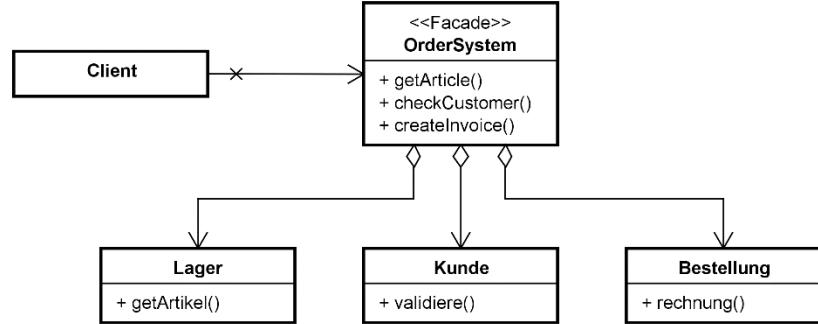
- Subsystemklassen – Beispiel: **Lager**, **Kunde**, **Bestellung**

- Implementieren die eigentliche Funktion.
- Wissen nichts von der Fassade (keine Referenz).



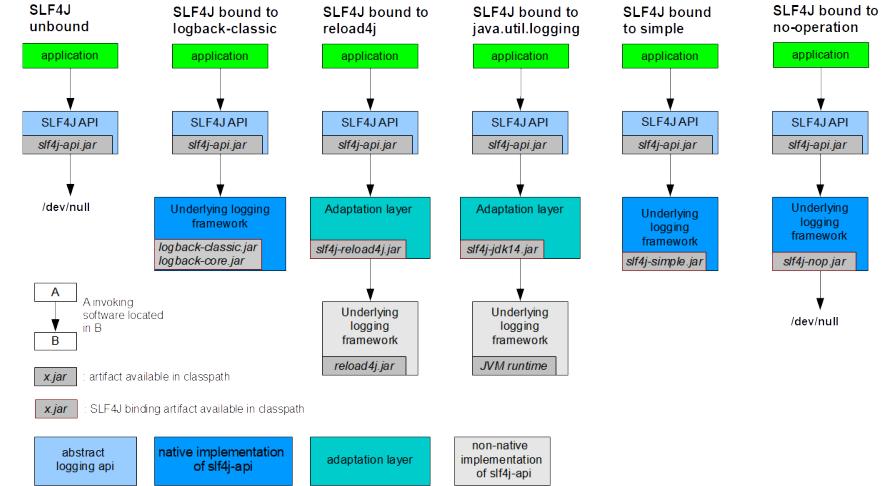
## Beispiel 2: Fassade

- Strukturmuster, objektbasiert
- Motivation für Einsatz
  - Vereinfacht die Anwendung mehrerer Subsysteme.
  - Minimiert die Abhängigkeiten zu den Subsystemen.  
→ Kopplung minimieren!
  - Einfache Austauschbarkeit eines Subsystems ermöglichen.
- Gefahr: Verkommt zum reinen Durchlauferhitzer.
- Empfehlung: Mit einer Fassade lässt sich sehr gut und einfach entkoppeln. Darauf achten, dass die Fassade nicht plötzlich wesentliche Funktionalität enthält, sie **delegiert** ausschliesslich!



# Beispiel 2: Simple Logging Facade for Java (SLF4J)

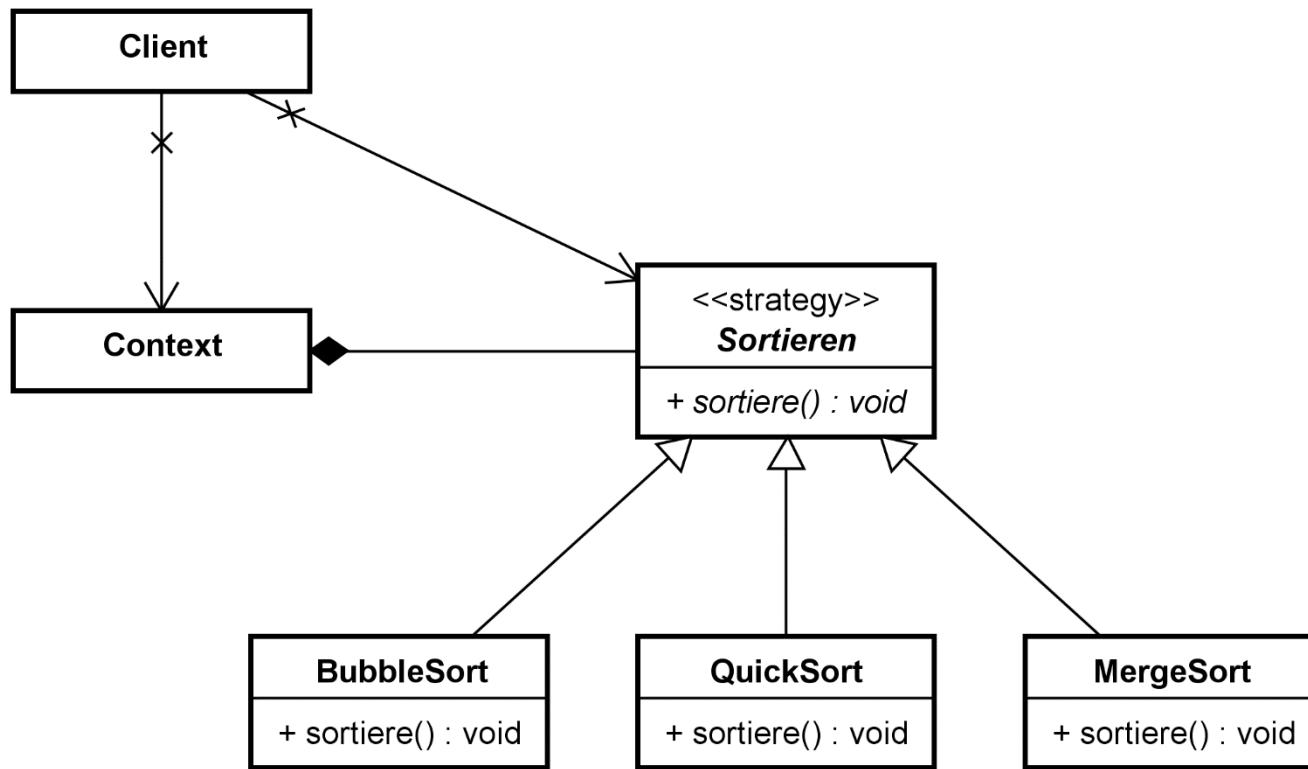
- Beispiel für Einsatz des Fassaden-Patterns. <https://slf4j.org>
- Die Fassade bleibt einheitlich, «darunter» können sich verschiedene Logging-Frameworks verstecken.
  - LogBack implementiert SLF4J beispielsweise direkt,
  - alle anderen Frameworks benötigen →**Adapter**-Libraries, wobei das korrekterweise eigentlich Bridges wären.
- Das konkret genutztes Logging Framework wird ausschliesslich durch entsprechendes Deployment bzw. über Classpath gesteuert!
- Ideal für Library- oder Framework-Entwicklung: Logging-Framework kann flexibel von Nutzer\*in ausgewählt werden.



# **Strategie**

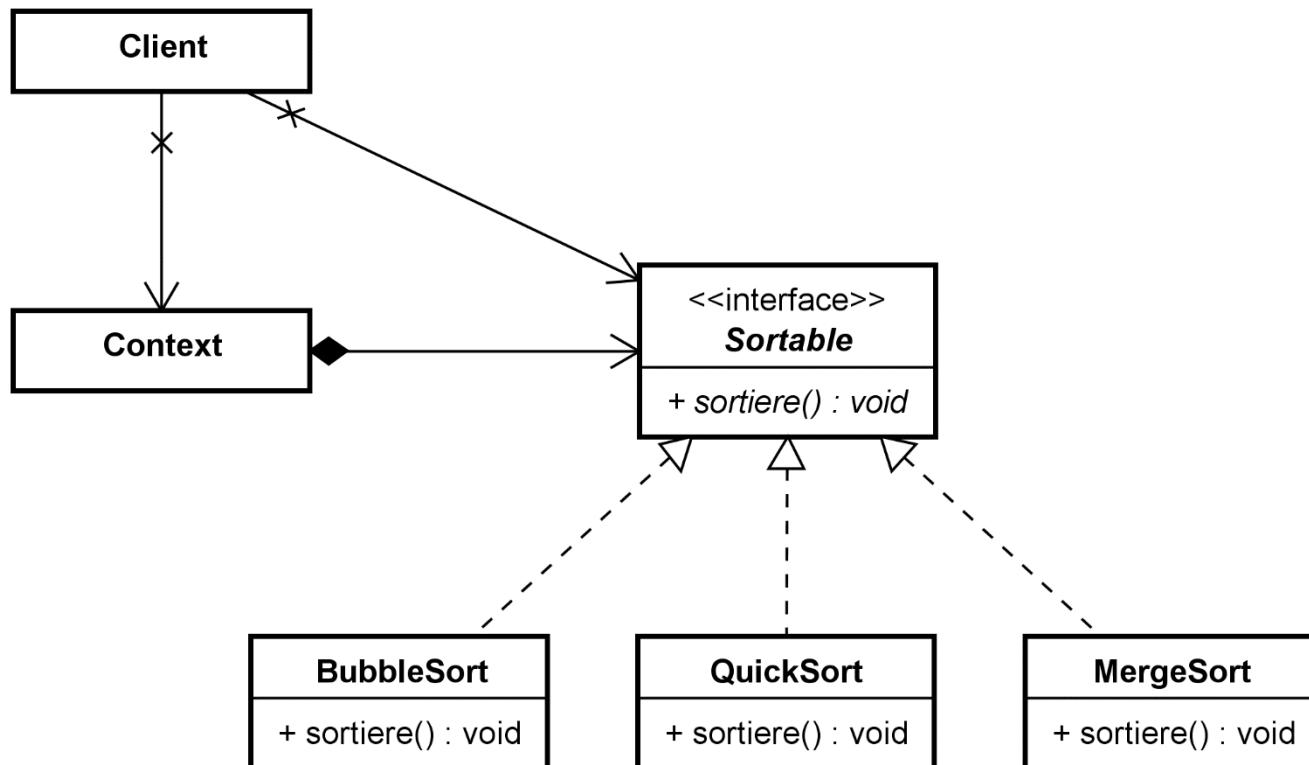
# Beispiel 3: Strategie mit abstrakter Basisklasse

- Definiere eine Familie von Algorithmen, kapsle jeden Einzelnen und mache sie austauschbar. Somit ist es möglich den Algorithmus unabhängig vom ihn nutzenden Klienten zu variieren.



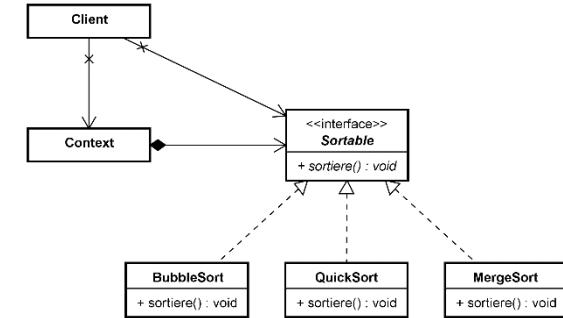
# Beispiel 3: Strategie mit Interface

- Für Sprachen welche keine Mehrfachvererbung, stattdessen aber Interfaces anbieten (z.B. Java), ersetzt man die (voll-)abstrakte Basisklasse gerne durch ein Interface.



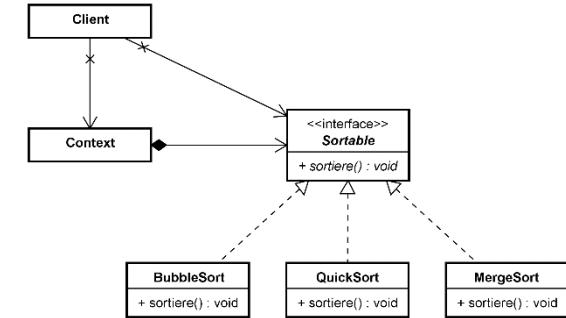
# Beispiel 3: Strategie - Teilnehmer

- Strategie – Beispiel: **Sortable**
  - Vollabstrakte Klasse oder Interface, definiert die Schnittstelle.
- Kontext – Beispiel: **Context**
  - Ist Optional, kann auch direkt durch Client erledigt werden.
  - Besitzt eine Referenz auf die konkrete Strategie, erstellt diese ggf. auch gleich selber.
  - Stellt ggf. eine Datenschnittstelle für die Strategien zur Verfügung.
- Konkrete Strategien – Beispiel: **BubbleSort, MergeSort etc.)**
  - Implementieren einen konkreten Algorithmus.
  - Greifen evt. auf den Kontext zu (für die Daten).



## Beispiel 3: Strategie

- Verhaltensmuster, objektbasiert
- Motivation für Einsatz:
  - Anbieten von unterschiedlichen Varianten/Implementationen von Algorithmen (z.B. Zeit vs. Speicher, Aufwand).
  - Eng verwandte Klassen, die sich nur im Verhalten unterscheiden, zusammenfassen.
  - Wenn der Bedarf nach unterschiedlichem Verhalten viele Bedingungsanweisungen zur Folge hätte.
- Empfehlung: Ein Pattern, das man leicht unterschätzt, und das sich auch bei sehr kleinen Methoden schon lohnen kann. Außerdem lassen sich damit z.B. grosse und hässliche **switch**-Statements wunderbar eliminieren.



# **Empfehlungen zu Entwurfsmustern**

# **Empfehlungen - Einsatz von Entwurfsmustern**



- Voraussetzungen
  - Man muss die Entwurfsmuster kennen und verstehen!
  - Quellen: Literatur oder Internet.
- Sinnvolle Auswahl und überlegter Einsatz
  - Entwurfsmuster sind keine ultimative Lösung für Alles!
  - Erfahrung sammeln, Erfahrung notwendig.
  - Besser kein Muster einsetzen, als das Falsche!

# Empfehlungen - Auswahl von Entwurfsmustern



- Es ist nicht immer einfach, das passende Muster (wenn überhaupt) auszuwählen!
- Sinnvolles Vorgehen:
  - Geht es um Erzeugung, Struktur oder Verhalten?
  - Passende Muster mit gleicher Aufgabe vorselektieren.
  - Welche Vor- und Nachteile bieten die Muster?
    - 'Bestes' Muster auswählen (am meisten Vorteile, grösste Vereinfachung etc.)
  - Wenn unentschieden: Wo haben Sie am meisten Freiheiten?
    - Muster mit grösster Flexibilität auswählen (grösstes Potential bei Erweiterungen / Wartung).

# Empfehlungen - Verifikation des Entscheides



- Nachdem man sich für ein Muster entschieden hat, sollte man unbedingt anhand von realen oder fiktiven Beispielen **verifizieren**, ob die erhofften/erwünschten positiven Aspekte tatsächlich vorhanden sind!
- Beispiel anhand des Strategiemusters:
  - Sind weitere, sinnvolle Algorithmen in Form von Strategien implementierbar/vorstellbar?
  - Haben diese adäquaten Zugriff auf alle notwendigen Daten?
  - Lässt sich die konkrete Strategie sinnvoll bestimmen bzw. konfigurieren?
  - Wird das Resultat letztlich einfacher oder komplizierter?

# Empfehlungen - Variieren von Entwurfsmustern



- Auch wenn Entwurfsmuster wohlüberlegt und vielfältig erprobt sind heisst das nicht, dass man sich stur daran halten muss!
- Entwurfsmuster sind 'nur' ein Konzept, welches auch wohlüberlegt verändert und optimiert werden darf.
  - Sturheit in der Implementation kann ein Design auch komplizierter oder aufwändiger machen.
  - Eingesetzte Sprache erlaubt evt. eine vereinfachte Umsetzung.
  - Im Gegenzug kann eine unbedachte Veränderung die spätere Entwicklung empfindlich behindern, oder gar die zentrale Idee eines Musters zerstören.
- Erfahrung und gesundes Augenmass notwendig!  
**→ Wer hat behauptet OO-Design sei einfach? ☺**  
Aber spannend auf jeden Fall!

## **Empfehlungen - Kombination von Entwurfsmuster**

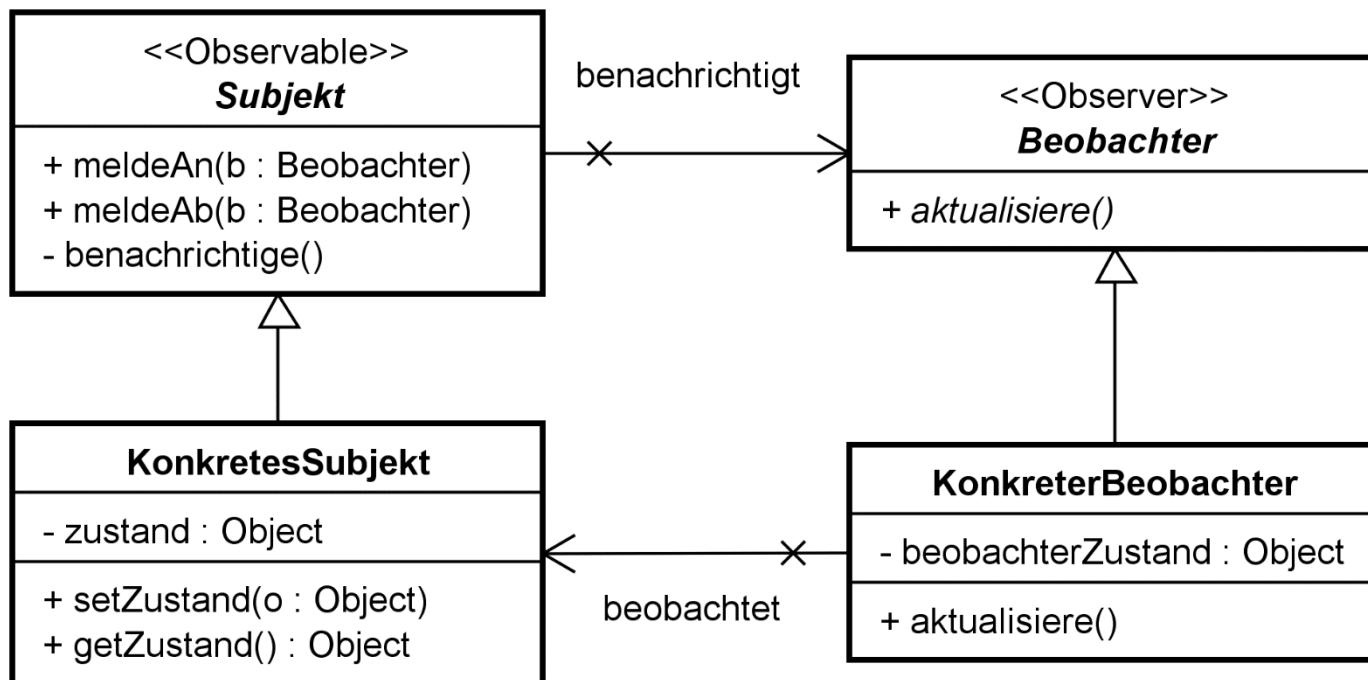


- Passiert relativ häufig, wird in der Literatur aber kaum erwähnt.
- Eine wirklich effiziente Lösung lässt sich manchmal nur durch eine enge Kombination von Mustern erreichen.
  - Komplexität wird dadurch aber grösser.
  - Muster treten nicht mehr in 'Reinform' auf, und sind darum ggf. schwieriger als solche zu erkennen.
- Beispiele:
  - Fabrik für Zustände.
  - Fassade für Fabriken von dekorierten Strategien.

# **Beobachter**

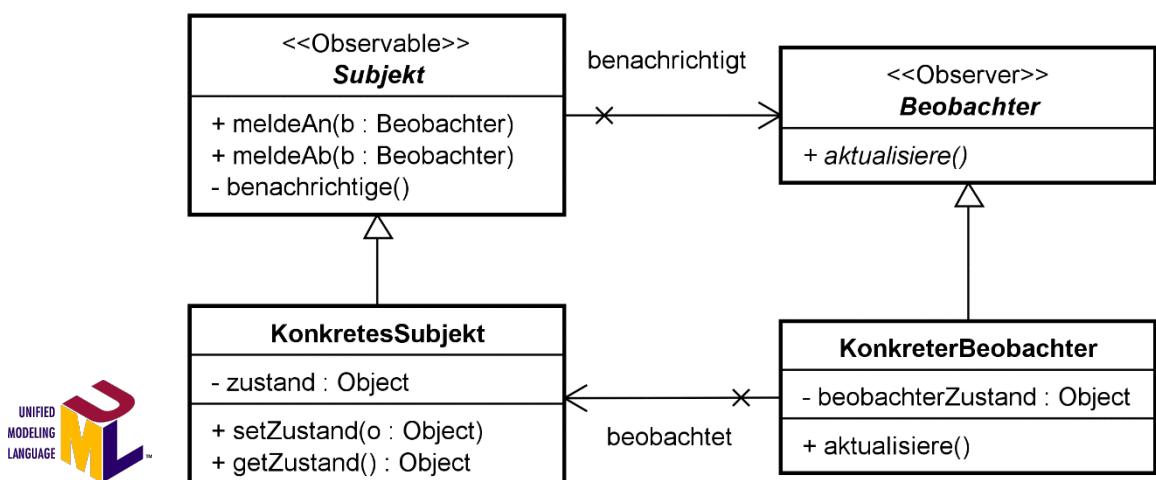
## Beispiel 4: Beobachter

- Definiert eine Abhangigkeit zwischen einem Subjekt (**Observable**) dessen Zustand ndern kann, und einer Menge von Beobachtern (**Observer**) die darber informiert werden sollen.



# Beispiel 4: Beobachter - Teilnehmer

- **Subjekt – Observable:**
  - Verwaltet seine Beobachter (0..n).
  - Bietet Methoden zur An- und Abmeldung an.
- **Beobachter – Observer:**
  - Definiert eine Benachrichtigungsschnittstelle.
- **Konkretes Subjekt / Konkreter Beobachter:**
  - Konkrete Typen senden und empfangen Aktualisierungen.

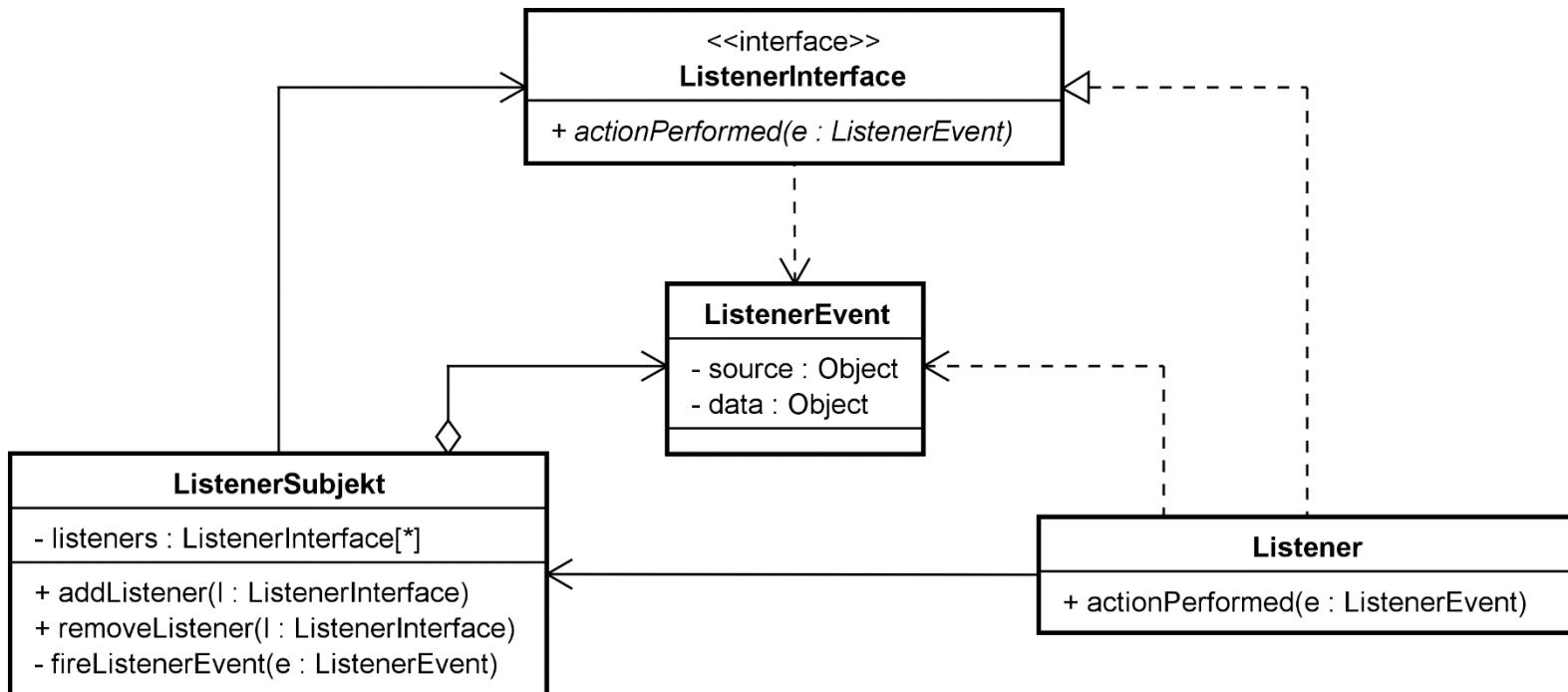


## **Beispiel 4: Beobachter**

- Verhaltensmuster, objektbasiert.
- Motivation für Einsatz:
  - Wenn nur eine lose Kopplung der Zuhörer bestehen soll/darf.
  - Wenn die Anzahl der vorhandenen Zuhörer nicht interessiert.
  - Zur Kommunikation entgegen der Abhängigkeitsrichtung.
  - Auch zur Auflösung von zyklischen Referenzen.
- Sehr typisch für MVC: Änderungen des Modelles müssen an die verschiedenen Views propagiert werden.

# Beispiel 4: Beobachter als Event/Listener (Java)

- Bei Java nutzt man als Ersatz für das Observer-Pattern das **Event/Listener-Pattern** welches auf Interfaces und Event-Klassen basiert.
  - Bekanntestes Listener-Interface: **ActionListener**



## Beispiel 4: Namensgebung bei Event/Listener (Java)

- Event → Eigentliches Subjekt.
- Eventquelle → Verwaltet die Beobachter.
  - public addXxxListener(...)**
  - public removeXxxListener(...)**
  - private fireXxxEvent(...)**
- Listener → Beobachter
  - public Xxx[Event|Performed](...)**
- Beispiel für **ActionListener**:
  - addActionListener(ActionListener listener)**
  - removeActionListener(ActionListener listener)**
  - actionPerformed(ActionEvent actionEvent)**

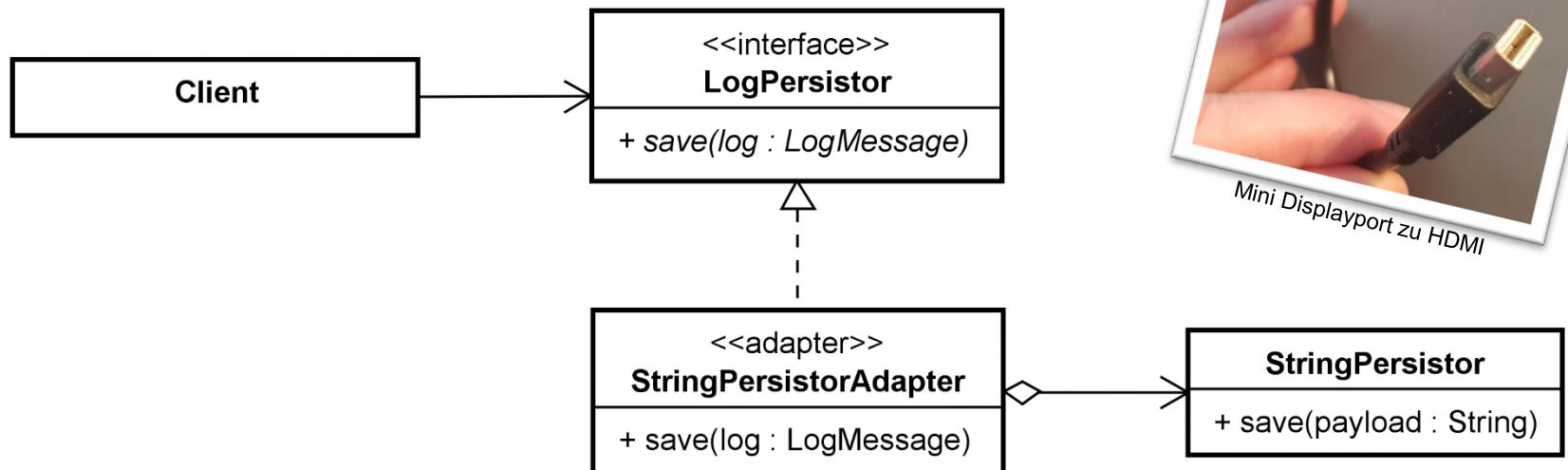
## **Beispiel 4: Event/Listener-Modell in Java**

- Das Event/Listener-Modell von Java ist deutlich besser und flexibler als das «reine» Observer-Pattern der GoF!
    - Java kennt Interfaces, somit kann die Vererbung vom abstrakten Subjekt und Beobachter entfallen → besseres Design!
  - Java kennt seit Version 1.0 ein Interface **Observer** und eine Klasse **Observable**. Beide werden zur Verwendung schon lange **nicht mehr empfohlen**, und sind seit Java 9 (endlich) **deprecated**.
- Bei Java konsequent das Event-/Listener-Modell verwenden!**

# **Adapter**

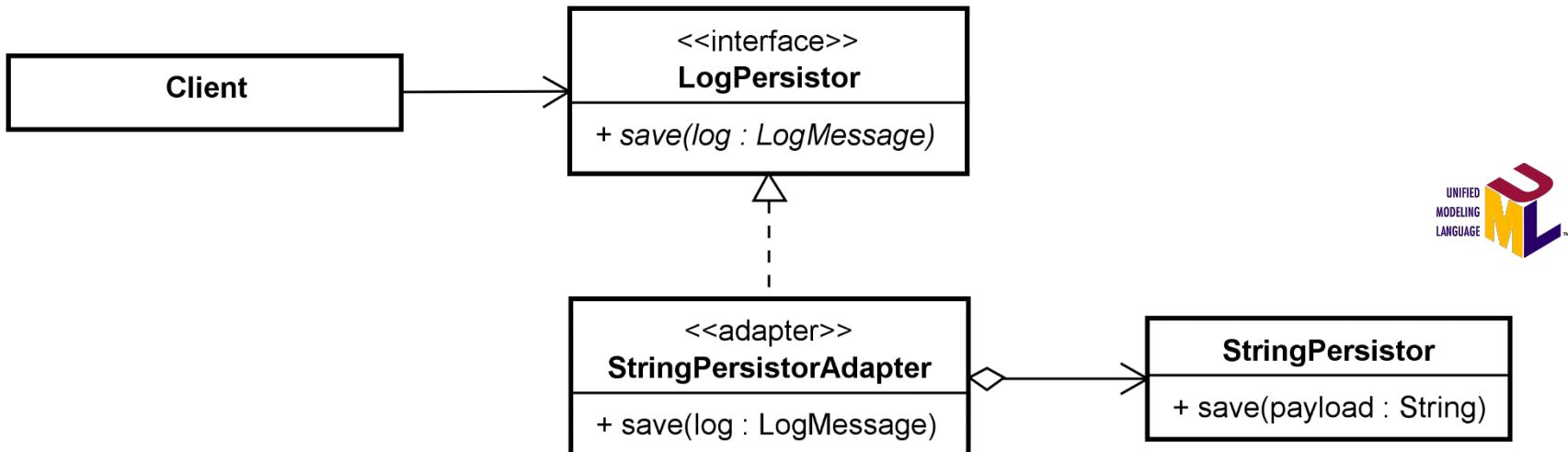
# Beispiel 5: Adapter

- Anpassen der Schnittstelle einer Klasse an die von den Klienten erwartete (Ziel-)Schnittstelle.
- Auch bekannt als „Wrapper“-Pattern.



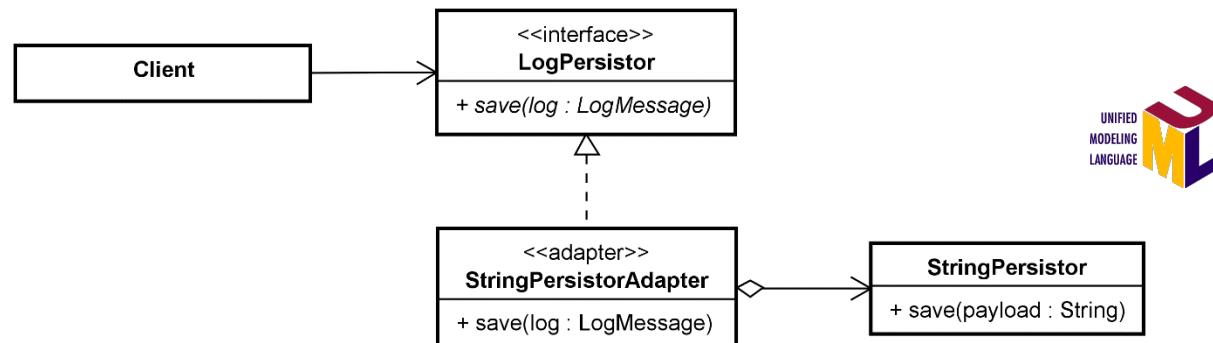
# Beispiel 5: Adapter

- Strukturmuster, klassen- oder objektbasiert
- Motivation für Einsatz:
  - Einfachere Wiederverwendung von existierenden Klassen oder Komponenten, deren Schnittstelle aber unpassend ist.
  - Eine möglichst allgemeine Schnittstelle zu implementieren, und diese dann prinzipiell durch Adapter anzupassen.



# Beispiel 5: Adapter - Teilnehmer

- Interface – Beispiel: **LogPersistor**
  - Die effektiv gewünschte Zielschnittstelle.
  - Kann eine abstrakte Klasse oder ein Interface sein.
- Adapter – Beispiel: **StringPersistorAdapter**
  - Verwendet die adaptierte Klasse/Objekt.
  - Spezialisiert oder implementiert die Zielschnittstelle.
- Adaptierte Klasse – Beispiel: **StringPersistor**
  - Adaptierte Klasse, deren Schnittstelle adaptiert (vgl. wrappen) werden soll.



# **Ergänzende Hinweise**

# Kommentar zu den ausgewählten Patterns

- Einfache Patterns, die aber sehr häufig eingesetzt werden!
- Mögliche Einstiegspunkte in die Anwendung von Entwurfsmustern:
  - Denken Sie an Ihre eigenen Projekte!
  - Gibt es Muster, die Sie in Ihren Projekten einsetzen könnten?
- Am Anfang müssen Sie auch aus Fehler lernen.
  - Gefahr: Mit Kanonen auf Spatzen schiessen.
  - Konkret: Das Muster kann plötzlich komplexer als das darin enthaltene/gelöste Fachproblem sein.

→ **Entwurfsmuster sind kein goldener Hammer!**

# Logger-Projekt - Aufträge

- Beachten Sie die Muss-Features im Projektauftrag!
    - Implementation eines Adapter-Pattern.
    - Implementation eines Strategy-Pattern.
  - Gute Beispiele für «**Separation of Concerns**» (SoC)
    - Lassen sich dadurch sehr gut Unit Testen.
    - Führen zu einem feingranularem Design (kleine[re] Klassen).
    - Einfaches Testing mit →Test Doubles möglich.
- Es sind somit explizit **keine** «Schulbeispiele»!
- Nur als Muss-Features ist es aussergewöhnlich. ☺

# Zusammenfassung

- Relativ grosse Zahl an Entwurfsmuster verfügbar (20+)
  - Für verschiedene Zwecke / Problemlösungen nutzbar.
  - Weitgehend sprachenunabhängiger Entwurf.
  - Gut und ausführlich dokumentiert.
  - Rad muss nicht immer wieder neu erfunden werden.
  - Hoher Bekanntheitsgrad und Wiedererkennung.
- Nicht für Alles gibt es ein Entwurfsmuster das passt
  - Entwurfsmuster sind kein goldener Hammer!
  - Auswahl des geeigneten Musters nicht immer einfach.
  - Erzwungener Einsatz geht meistens schief.
  - Manchmal Kombinationen von Mustern anwenden.

# Quellen 1 - Literatur

- *Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides:*

## Entwurfsmuster

MITP

Januar 2015

ISBN: 978-3-8266-9700-5

Original



- *Eric Freeman, Elisabeth Freeman, Kathy Sierra, Bert Bates:*

## Entwurfsmuster von Kopf bis Fuss

O'Reilly Deutsch

Februar 2015

ISBN: 978-3-95561-986-2



- *Matthias Geirhos:*

## Entwurfsmuster

Rheinwerk Computing

Mai 2015

ISBN: 978-3-8362-2762-9



## Quellen 2

- Internet (Auswahl):

- Design Patterns (Wikipedia, englisch)

- [https://en.wikipedia.org/wiki/Design\\_Patterns](https://en.wikipedia.org/wiki/Design_Patterns)

- Entwurfsmuster (Wikipedia, deutsch)

- <https://de.wikipedia.org/wiki/Entwurfsmuster>

- Patterns, Anti-Patterns, Refactoring und UML

- [https://sourcemaking.com/design\\_patterns](https://sourcemaking.com/design_patterns)

- Entwurfsmuster - Übersicht (Uni Rostock)

- <https://wwwswt.informatik.uni-rostock.de/deutsch/Infothek/Entwurfsmuster/patterns>

- Refactoring Guru – Design Patterns

- <https://refactoring.guru/design-patterns>

**Fragen?**

Verteilte Systeme und Komponenten

# Konsistenz und Replikation

Martin Bättig



# Inhalt

- CAP-Theorem
- Konsistenz und Konsistenzgarantien
- Replikation
- Ausfallsicherheit mittels Replikation

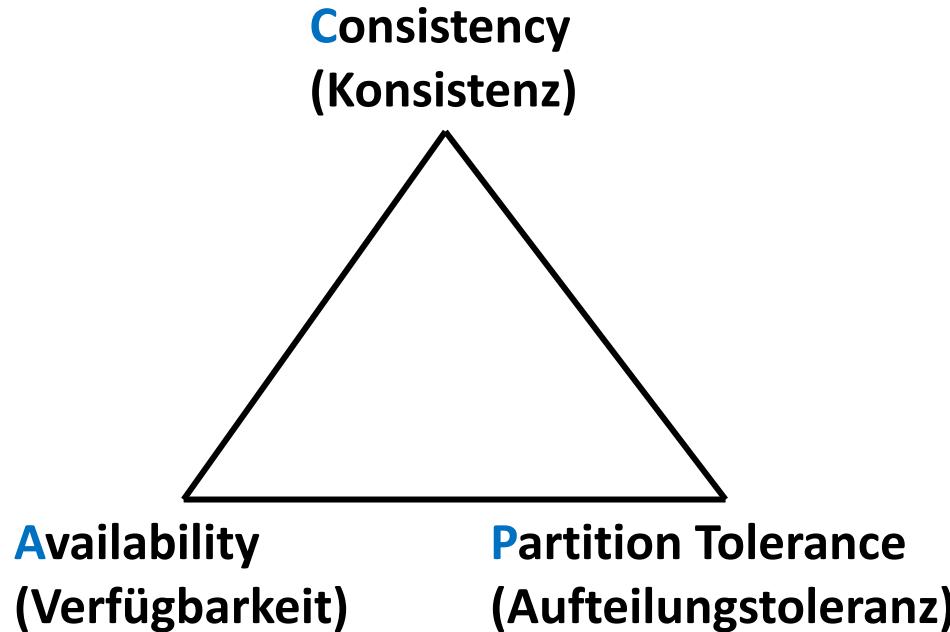
# Lernziele

- Sie kennen das CAP-Theorem (Consistency, Availability, Partition Tolerance) und können verteilte Systeme anhand dieses Protokolls einordnen.
- Sie kennen den Begriff der Konsistenz sowie die Konsistenzgarantien «sequential consistency» und «eventual consistency» und können einschätzen, bei welchen Arten von verteilten Systemen diese Konsistenzgarantien eingesetzt werden.
- Sie kennen den Prozess der Replikation und verschiedene Möglichkeiten ein Replika zu erzeugen.
- Sie können Replikation in Zusammenhang mit den Konsistenzanforderungen setzen.
- Sie kennen die Mechanismen Heartbeat und Leader-Election und wie Sie damit Replikas zum Zweck der Ausfallsicherheit eines verteilten Systems einsetzen können.

# **CAP-Theorem**

# CAP-Theorem

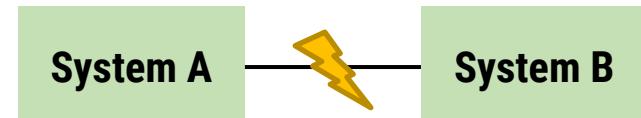
Brewer's Theorem (2002) besagt, dass ein verteiltes System nicht mehr als zwei der folgenden Eigenschaften garantieren kann:



- **Konsistenz:** Alle beteiligen Systeme haben identische und aktuelle Daten.
- **Verfügbarkeit:** Daten sind für alle Lese-/Schreiboperationen sofort bereit.
- **Aufteilungstoleranz:** System kann trotz Aufteilung in zwei oder mehr bzgl. Kommunikation getrennte Partitionen weiter operieren.

# CAP-Theorem: Intuition

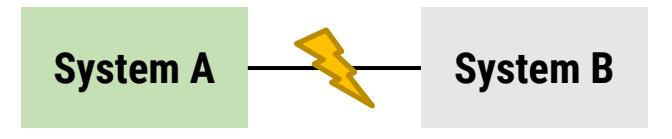
- **AP-Systeme** operieren trotz ausgefallener Kommunikation weiter.  
Reduktion der Konsistenz.  
(Beispiele: Domain-Name-System, Consumer-Filesynchronisation)
- **CP-Systeme** operieren trotz ausgefallener Kommunikation weiter, aber nur soweit die Konsistenz gewährleistet ist.  
Reduktion der Verfügbarkeit.  
(Beispiele: Email, Bankomat)
- **CA-Systeme** bieten sowohl Konsistenz als auch Verfügbarkeit darum nicht partitioniert werden (\*).  
Reduktion der Partitionstoleranz.  
(Beispiel: Typische Webapplikation).



Zustand X

System B

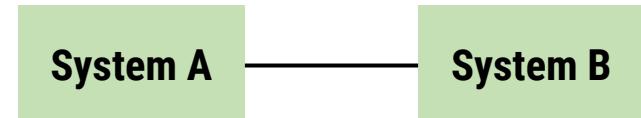
Zustand Y



System A

System B

Nicht verfügbar



System A

System B

Partitionierung  
nicht möglich  
(alles oder nichts)

(\*) Relevanz sinkend: <https://www.infoq.com/articles/cap-twelve-years-later-how-the-rules-have-changed/>

# **Konsistenz und Konsistenzgarantien**

# Konsistenz

- Sobald ein System A und ein System B über die gleichen Daten verfügen (z.B. wenn B ein Replikat von A ist), stellt sich die Frage bzgl. deren Konsistenz.
- Liefern A und B bei identischen Anfragen die identische Antwort sind die Daten der beiden Systeme **konsistent**.
- Systeme können unterschiedliche Garantien bzgl. der Konsistenz der Daten geben.

# Konsistenzgarantien

- Anforderungen an die verteilte Applikation bestimmen die notwendige Art der Konsistenz.
- Typische Konsistenzgarantien sind:
  - **Sequential Consistency:** Die Sequenz der Operationen eines Systems S ist für alle Systeme die gleiche und passt in eine Gesamtordnung.
  - **Eventual Consistency:** Modifikationen werden erst mit einer gewissen Verzögerung sichtbar.

# Sequential Consistency

«Sequential consistency» gibt folgende Konsistenzgarantie:

Das Resultat einer verteilten Ausführung ist das gleiche wie:

- wenn Lese- und Schreiboperationen aller beteiligen Systeme einer globalen sequentiellen Abfolge zugeordnet werden können.
- Und in dieser Abfolge für jedes System die Operationen in der gleichen Reihenfolge, in welcher es die Operationen getätigt hat, vorkommen.
- Die Zeit spielt dabei keine Rolle!

Typische Anforderung einer verteilten Eingabe und Verarbeitungs-Applikation.

Vorteil: Unabhängige Teiloperationen blockieren das Gesamtsystem nicht.

# Beispiel: Sequentielle Konsistenz bei Konten

Annahme: Applikation, in welcher wir in ein Konto lesen oder schreiben.

## Notation:

**K  $\Rightarrow$  B** lese von Konto K und erhalte Betrag B.

**B  $\Rightarrow$  K** schreibe Betrag B in Konto K.

## Anforderung an sequentielle Konsistenz **erfüllt**:

System A: 

X $\Rightarrow$ 10	Y $\Rightarrow$ 20	15 $\Rightarrow$ X	15 $\Rightarrow$ Y
--------------------	--------------------	--------------------	--------------------

System B: 

20 $\Rightarrow$ Y	X $\Rightarrow$ 10	Y $\Rightarrow$ 20	20 $\Rightarrow$ Z
--------------------	--------------------	--------------------	--------------------

## Anforderung an sequentielle Konsistenz **nicht erfüllt**:

System A: 

X $\Rightarrow$ 10	Y $\Rightarrow$ 20	15 $\Rightarrow$ X	15 $\Rightarrow$ Y
--------------------	--------------------	--------------------	--------------------

System B: 

X $\Rightarrow$ 10	Y $\Rightarrow$ 25	15 $\Rightarrow$ X	20 $\Rightarrow$ Y
--------------------	--------------------	--------------------	--------------------

# Eventual Consistency und Monotonic Reads

## Beschreibung (informell):

- Das Gesamtsystem konvergiert gegen die aktuellen Daten. Die Konsistenzanforderung «[Eventual Consistency](#)» erlaubt, dass ein System zu einem Zeitpunkt T noch mit veralteten Daten arbeitet.
- Typischerweise in Verbindung mit [Monotonic Reads](#): Aufeinander folgende Leseoperationen auf ein Objekt O sehen entweder den gleichen oder einen aktuelleren Wert als den zuletzt gelesen.

## Nutzen:

- Zeitliche Entkoppelung, Verfügbarkeit, tolerieren von Netzpartitionierung.

## Einsatzszenario:

- Falls Daten nicht immer aktuell sein müssen **UND**
  - falls nur eine Instanz die Hoheit über Modifikationen besitzt**ODER**
  - nur Schreiboperationen getätigt werden.

# Fallbeispiel: «Zuletzt besucht»

Besucher eines Profils P werden Liste L gespeichert.

Bei Betrachten von Profil P, sieht der Besitzer Liste L.

Who's viewed your profile  
People who viewed your profile in the past 90 days

All viewers 2 work at Lucerne University of Applied Sciences and Arts  
1 works at [REDACTED] 1 works at TEDxHochschuleLuzern  
3 found you through LinkedIn Search

18 Profile viewers

Senior Scientist in IoT Systems & Software at [REDACTED]  
Viewed 2d ago  
7 mutual connections

Unlock the rest of the list with Premium  
Chris and millions of other members use Premium  
Try Premium for free

## Notation:

- $P \Rightarrow L_t$  erhalte Liste der letzten Besucher von Profil P zum Zeitpunkt t.
- $L_t \Rightarrow P$  füge Besucher B zur Liste von Profil P hinzu.

## Anforderung an Eventual Consistency mit Monotonic Reads **erfüllt**:

System A:  $L_1 \Rightarrow P$   $L_2 \Rightarrow P$   $L_3 \Rightarrow P$

System B:  $P \Rightarrow L_1$   $P \Rightarrow L_3$   $P \Rightarrow L_3$   $P \Rightarrow L_3$

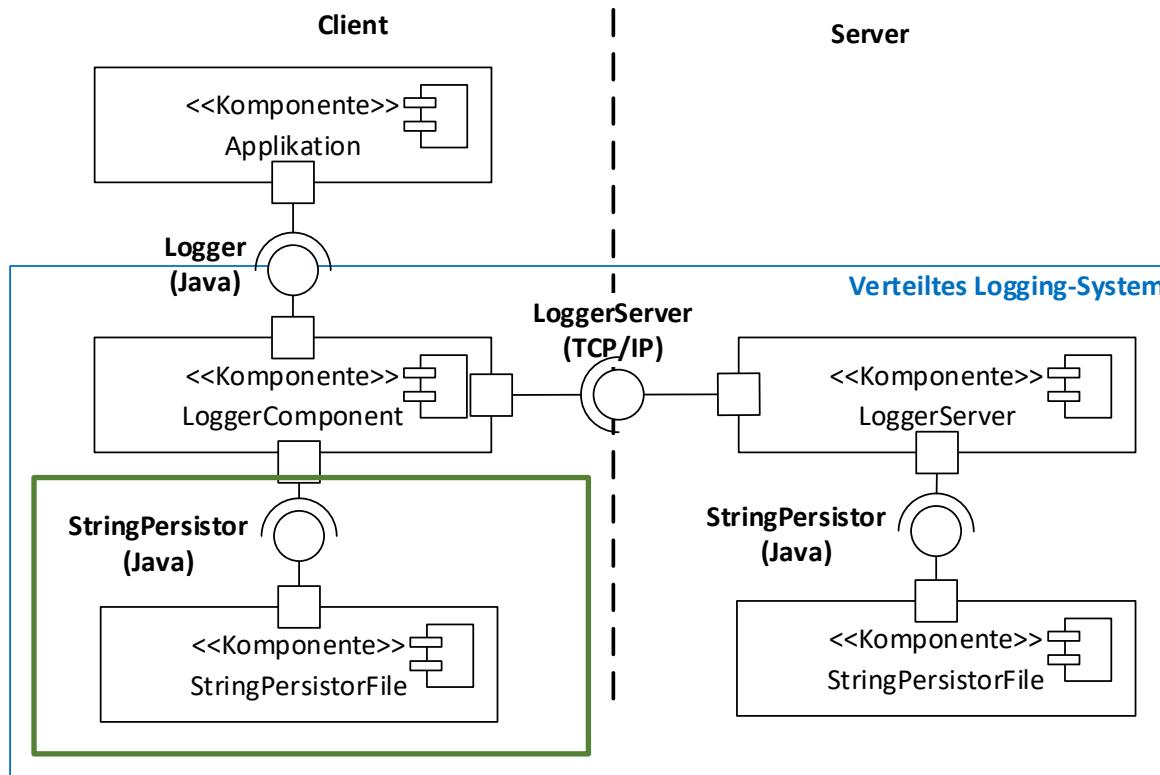
## Anforderung an Eventual Consistency mit Monotonic Reads **nicht erfüllt**:

System A:  $L_1 \Rightarrow P$   $L_2 \Rightarrow P$   $L_3 \Rightarrow P$

System B:  $P \Rightarrow L_3$   $P \Rightarrow L_1$

# Übung: Konsistenzgarantien des verteilten Logger-Systems

- Welche Konsistenzgarantien wünschen Sie sich für das verteilte Logging-System?
- Mit welcher Begründung?



**Lokaler  
Cache**

# **Replikation**

# Replika und Replikation

## Replika:

- Eine Kopie eines Systems.
- Diese muss nicht Bit für Bit identisch sein, sollte aber die gleichen Informationen enthalten

**Beispiel:** Abbild eines Dateisystems erstellen vs. kopieren Datei-für-Datei.

## Replikation:

- Der Prozess aus einem Original (**Primary**) eine Kopie (**Replikat**) herzustellen.
- Hauptanwendungen der Replikation bei verteilten Systemen:
  - **Ausfallsicherheit:** Fällt Primary A aus, kann ein Replikat B übernehmen.
  - **Performance:** Verteilung von Anfragen über mehrere Systeme hinweg.

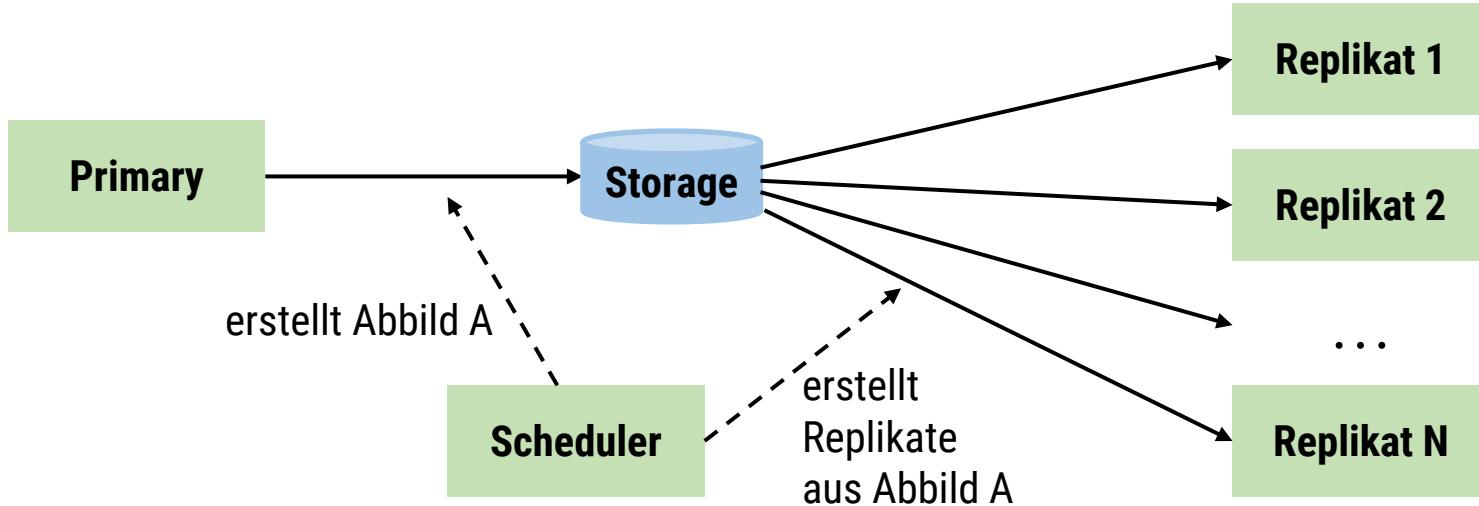
# Fragestellungen im Zusammenhang mit Replikation

- Wie konsistent müssen Primary und Replikate zu einem bestimmten Zeitpunkt sein?
- Wie kann ein Ausfall eines Systems erkannt werden?
- Wie kann der Ausfall eines Systems toleriert werden?
- Wo müssen Primary und Replikate platziert sein? (=> Thema des Inputs «Skalierung und Verteilung»)
- Wie wird die Last verteilt? (=> Thema des Inputs «Skalierung und Verteilung»)

# «Klassische» Replikation

Ein Scheduler führt in regelmässigen Intervallen (z.B. alle 6h) folgende Anweisungen aus:

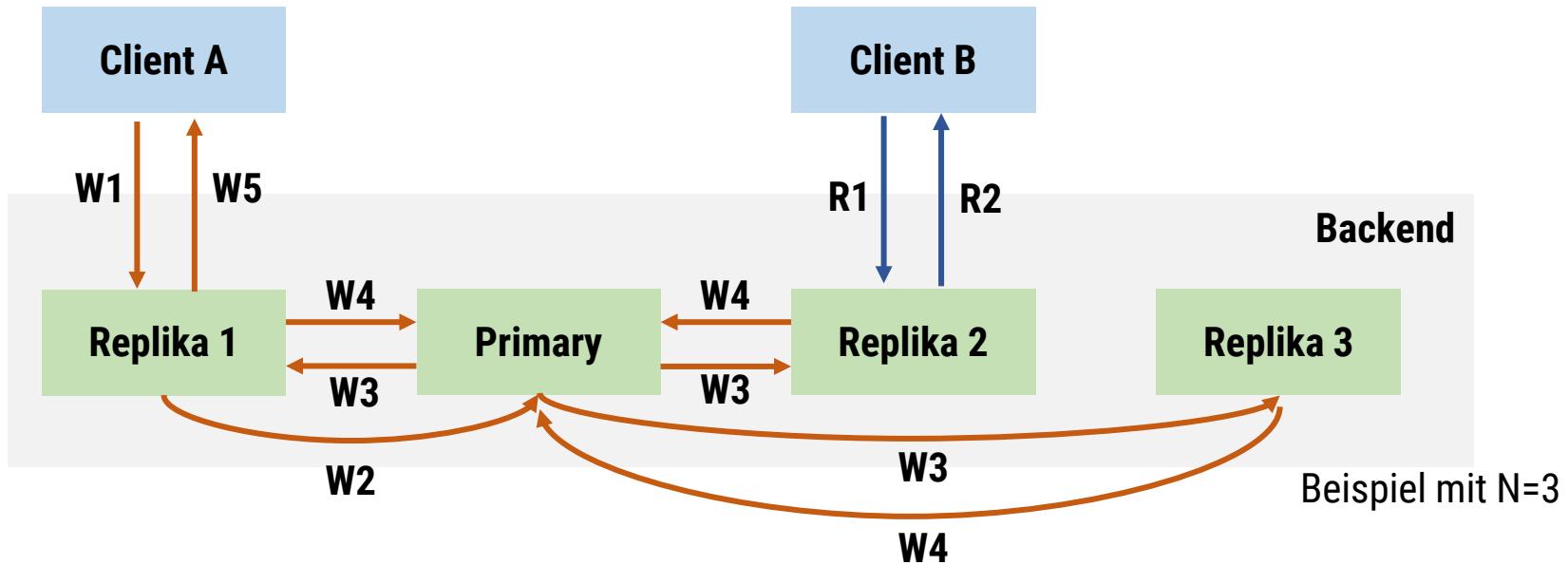
1. Erstellung eines Abbilds A des Primarys.
2. Einspielung von Abbild A in alle Systeme, welche als Replikat dienen sollen.



- Wie konsistent sind diese Replika?
- Was gilt es bei dem Prozess zu beachten?

# Primary-Backup-Protokoll für «On-the-Fly» Replikas

- 1 Primary: Behandelt alle Schreizugriffe.
- N Replikate: Beantwortet Lesezugriffe, leitet Schreizugriffe an Primary weiter.



W1: Schreibanfrage

W2: Weiterleitung an Primary

W3: Update von Primary an Replikat

W4: Replikat bestätigt Update

W5: Bestätige Schreibanfrage

R1: Leseanfrage

R2: Beantwortet Leseanfrage

# Eigenschaften des Primary-Backup-Protokolls

Standard-Implementation: Antwort auf Schreibanfrage kehrt erst zurück, sobald Replikas die Daten geschrieben haben -> blockierend.

- Push-Prinzip.
- Sequentielle Konsistenz.
- Aber Ausfallsicherheit?
- Auswirkung auf Antwortszeit?

# Fallstrick: Filesystem-Cache

- Betriebssysteme haben i.d.R. einen Filesystem-Cache (Zwischenspeicher).
- Änderungen werden zuerst in den Cache und erst nach gewisser Zeit auf Disk geschrieben.
- Für eine **sequentielle Konsistenz** (z.B. bei Datenbanken) muss Schreiben auf Disk erzwungen werden.

## Beispiel: java.nio.channels.FileChannel

### force

```
public abstract void force(boolean metaData)
                            throws IOException
```

Forces any updates to this channel's file to be written to the storage device that contains it.

If this channel's file resides on a local storage device then when this method returns it is guaranteed that all changes made to the file since this channel was created, or since this method was last invoked, will have been written to that device. This is useful for ensuring that critical information is not lost in the event of a system crash.

# Viele Variationen des Primary-Backup-Protokolls

Implementiert von vielen Produkten (z.B. DBs wie Oracle/MySQL/etc.), aber nicht in der ursprünglichen Form.

- **Nicht blockierender Schreibzugriff:**
  - Ggf. Datenverlust (Daten nicht auf Replikat) nach Crash von Primary oder Primary muss nach Crash wiederhergestellt werden (Downtime).
- **Push- vs. Pull-Prinzip:**
  - In festgelegtem Intervall: Keine sequentielle Konsistenz, aber je nach Daten und Anwendungszweck immer noch «Eventual Consistent».
- **Einzelner vs. verteilte Primarys:**
  - Jedes teilnehmende System kann für gewisse Objekte Primary und für andere Objekte Replikat sein.
  - Komplexer, aber schneller durch verteilte Schreibzugriffe.

# Push-Replikation: Replikationsqueue und Verbindungsauftbau

```
public class Replication implements Runnable {  
  
    private final String replicationAddress;  
    private final Queue<Message> replicationQueue  
        = new ConcurrentLinkedQueue<>();  
  
    public Replication(String replicationAddress) {  
        this.replicationAddress = replicationAddress;  
    }  
  
    @Override  
    public void run() {  
        try (ZContext context = new ZContext()) {  
            ZMQ.Socket socket = context.createSocket(SocketType.REQ);  
            socket.connect(replicationAddress);  
            processReplicationQueue(socket);  
        } catch (Exception ex) {  
            LOG.error(ex.getMessage(), ex);  
        }  
    }  
}
```

Queue für alle zu synchronisierenden Aktivitäten.

Verbindung zum Replikat aufbauen und replizieren.

# Push-Replikation: Messages hinzufügen und Verarbeitung

```
public void addMessage(Message message) {  
    replicationQueue.add(message); ← Message zur  
    notifyWaiters();  
}  
  
public void waitForSynchronization() {  
    waitForCondition(true); ← Warten bis Synchronisation  
}  
  
private void processReplicationQueue(ZMQ.Socket socket) {  
    while(true) {  
        waitForCondition(false);  
        Message message = replicationQueue.peek();  
        socket.send(message.toString());  
        socket.recvStr(); ← Verarbeiten: Synchrone  
        replicationQueue.remove();  
        notifyWaiters();  
    }  
}  
  
private synchronized void waitForCondition(boolean isEmpty) {  
    while (replicationQueue.isEmpty() != isEmpty) this.wait();  
}
```

Replikationsqueue hinzufügen

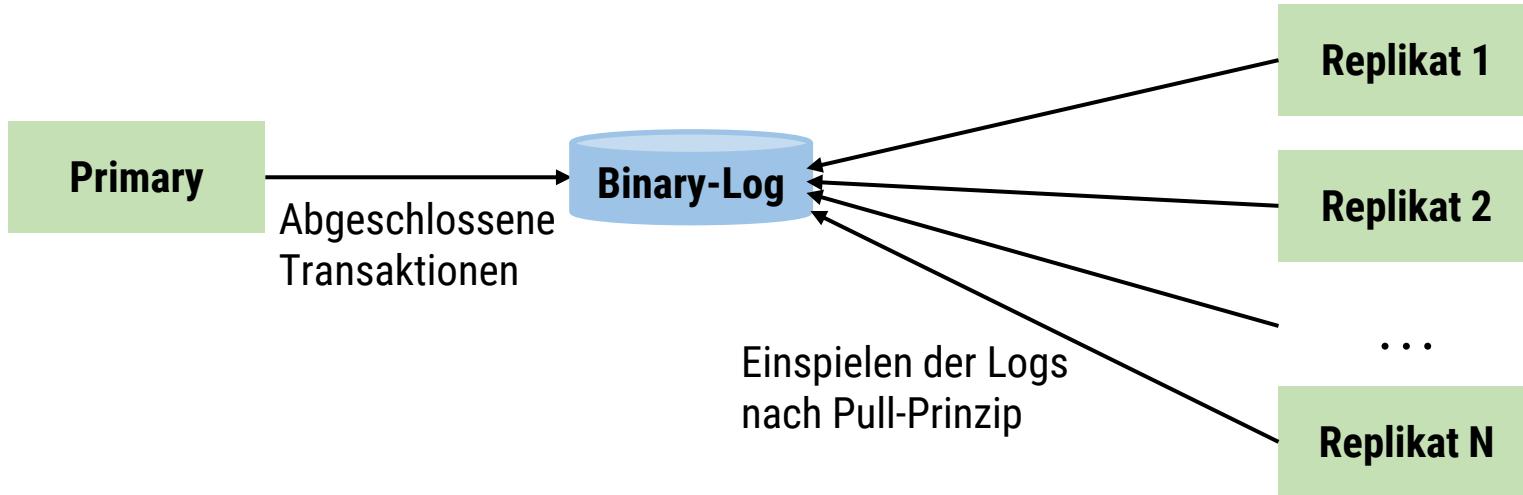
abgeschlossen ist.

Senden von Messages zum  
Replikat.

# Fallstudie: Standard-Replikation bei Datenbank «MariaDb»

## Grober Ablauf:

- Abgeschlossene Transaktionen werden in ein Binary-Log geschrieben.
- Replikats lesen neue Informationen im Binary-Log nach dem Pull-Prinzip und applizieren es lokal.
- Konsistenz: Replikas verfügen nur mit Verzögerung über die gleichen Daten.



# Klassenraumübung: Eigene Versuche mit Replikation

Übung mittels Online-Programmierumgebung:

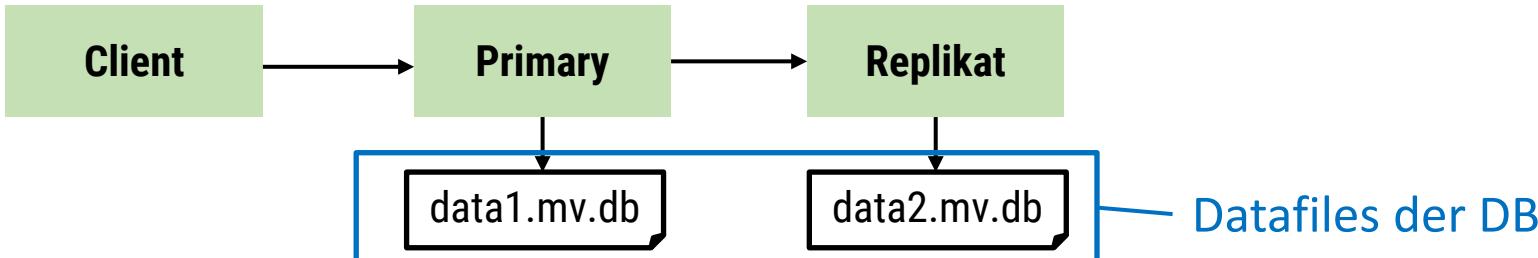
- <https://replit.com/@mbaettig/Replication>
  - Kein HSLU-Dienst (benötigt separates Konto), erstellen Sie einen Fork.

**Aufgabe:** Machen Sie eigene Versuche mit dieser *Teilimplementation* des Primary-Backup-Protokolls:

**Beim Start mittels folgenden Anweisungen:**

```
java AccountServer ./data1 tcp://localhost:5555 tcp://localhost:5556 primary  
java AccountServer ./data2 tcp://localhost:5556 tcp://localhost:5555 replica  
java AccountClient tcp://localhost:5555
```

erhält man folgendes Setup:



# Klassenraumübung: Eigene Versuche mit Replikation

## Machen Sie beispielweise folgendes:

1. Starten Sie den Primary, Replika und Client jeweils in einer eigenen Shell.
2. Erstellen Sie zwei Konten und Übertragen Sie einen Betrag.

```
create acc1 1000
```

*erstellt Konto acc1 mit 1000 CHF*

```
create acc2 500
```

*erstellt Konto acc2 mit 500 CHF*

```
transfer acc1 acc2 300
```

*überweist 300 CHF von acc1 zu acc2*

```
balance acc1
```

*zeigt Kontostand von acc1 an*

3. Stoppen Sie den Primary (Simulation eines Crashes).
4. Stoppen Sie den Client und starten Sie ihn mit Verbindung auf das Replika.
5. Machen eine weitere Überweisung.

## Machen sie sich folgende Überlegungen:

- Wie konsistent ist das Replika? Wie könnte man das ändern?
- Wie könnten Sie den Primary wieder konsistent erhalten? Gibt es Varianten?
- Könnten Sie mehrere Replikas mit diesem System verwalten? Performance?

# **Ausfallsicherheit mittels Replikation**

# Replikate für Ausfallsicherheit

**Fragestellung:** Wie erfolgt automatischer Wechsel auf Replikat, sobald Primary ausfällt?

## Teilfragen:

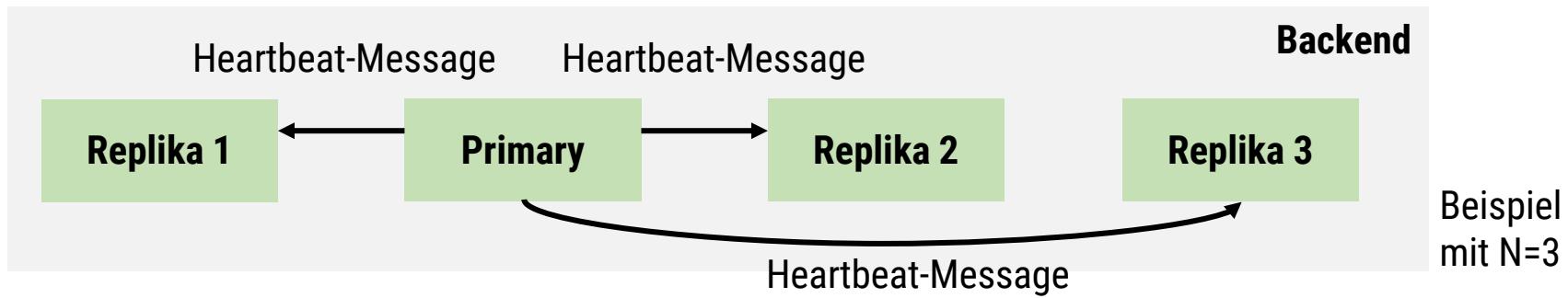
- Wie erkennen Replikate zuverlässig, dass der Primary ausgefallen ist?  
⇒ [Heartbeat-Protokolle](#)
- Wie ernennen die Replikate neuen Primary, falls mehrere Replikate existieren?  
⇒ [Leader-Election-Protokolle](#)
- Wie wissen Clients auf welches System sie verbinden müssen?  
⇒ [Namesdienste / IP-Routing](#)

# Heartbeat-Protokoll

**Ziel:** Erkennen eines Ausfalls des Primarys.

**Vorgehen:**

- *1 Primary* sendet Heartbeat-Message in festem Intervall (z.B. 100ms, abhängig von der Roundtrip-Time zwischen den Systemen).
- *N Replika* überprüfen, ob diese mindestens ein Heartbeat-Signal innerhalb eines Timeout-Intervalls erhalten haben (z.B. 1000ms). Ist dies nicht der Fall, wird der Server als Down betrachtet.



# Beispiel: Einfacher Heartbeat Sender (Auszug)

```
private static final long HEARTBEAT_INTERVAL = 100; // [ms]

private final ScheduledThreadPoolExecutor executor
    = new ScheduledThreadPoolExecutor(1);

private ZContext zContext;
private ZMQ.Socket socket;

private final Runnable action = new Runnable() {
    public void run() {
        socket.send(new byte[0]); ← Sende Heartbeat
    }                                (Message ohne Inhalt)
};

public void start() {
    zContext = new ZContext();
    socket = zContext.createSocket(SocketType.PUB);
    socket.bind("tcp://localhost:7777");
    executor.scheduleWithFixedDelay(action, HEARTBEAT_INTERVAL,
        HEARTBEAT_INTERVAL, TimeUnit.MILLISECONDS);
}
```

Sende Heartbeat  
(Message ohne Inhalt)

Sende Heartbeat  
über separaten Kanal

# Beispiel: Einfacher Heartbeat Detector (Auszug)

```
private static final long HEARTBEAT_DETECTION_INTERVAL = 1000; // [ms]
private ZContext zContext;
private ZMQ.Socket socket;
private final Runnable action = new Runnable() {
    public void run() {
        while(running) {
            if (socket.recv() == null) { ←
                running = false;
                noResponseHandler.run(); ←
            }
        }
    }
};

public void start() {
    zContext = new ZContext();
    socket = zContext.createSocket(SocketType.SUB);
    socket.subscribe(new byte[0]);
    socket.connect("tcp://localhost:7777");
    socket.setReceiveTimeOut(HEARTBEAT_DETECTION_INTERVAL);
    Thread thread = new Thread(action); ←
    running = true;
    thread.start();
}
```

Ist das Timeout eingetreten  
wird die Behandlungsroutine  
gestartet.

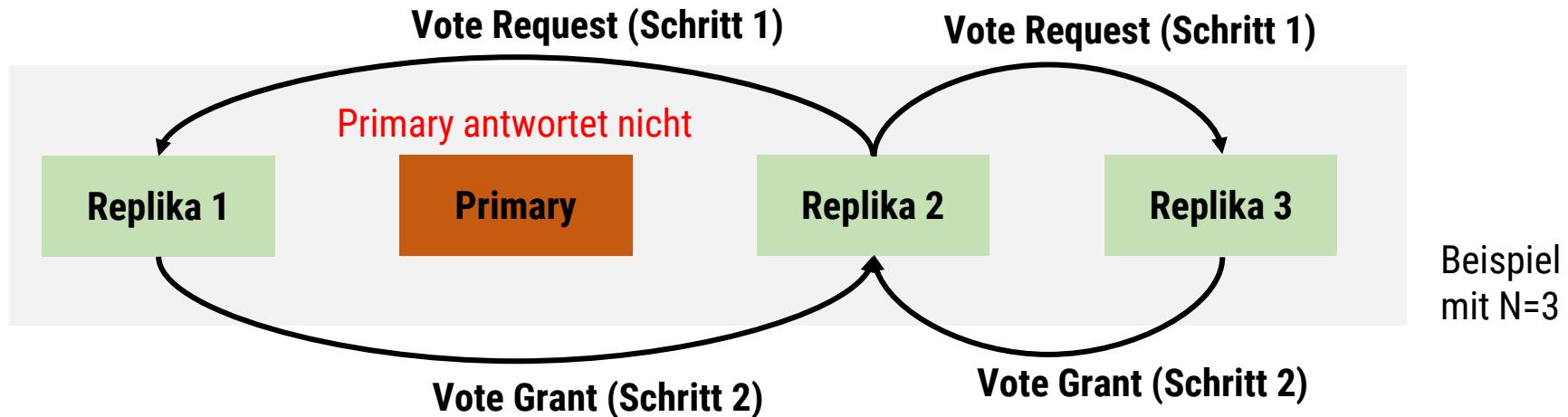
Setzte Timeout auf maximale  
Zeit ohne Heartbeat

# Leader-Election mittels Quorum

Ziel: Ernennen eines neuen Leaders mittels Quorum (Mehrheitsentscheid)

Vorgehen:

- 1 (oder mehr) Replika  $R$  erkennen Ausfall des Primary.
- Replika  $R$  sendet *Vote Request* an alle anderen Replika.
- Die anderen Replika senden *Vote Grant*, falls diese mit Wechsel einverstanden sind (keine anderen Requests; erkennen Primary als Down).
- Replika  $R$  wird neues Primary, falls es  $\text{ceil}(N / 2)$  *Vote Grants* erhält.

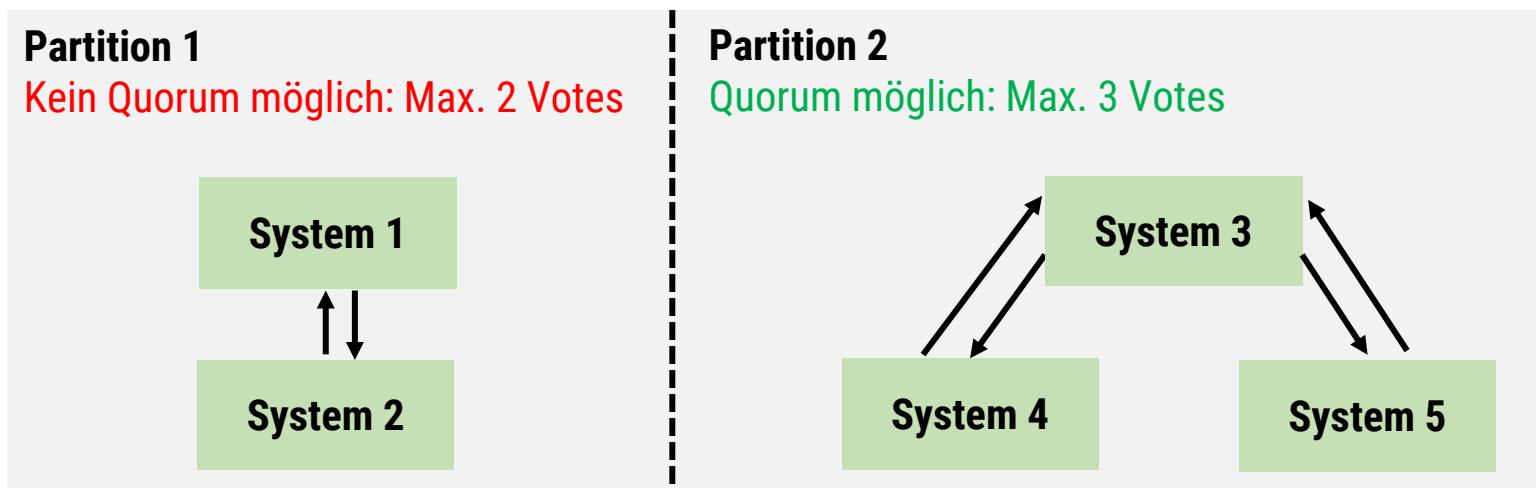


# Notwendigkeit eines Quorums

- Bei Netzwerkpartitionierung darf nur eine Partition weiter operieren.
- Quorum ab  $N=3$  Systemen möglich, wobei N ist sinnvollerweise ungerade ist.

**Beispiel:** Verteiltes System mit 5 Systemen wird durch Netzwerkausfall in 2 Partitionen unterteilt.

- Quorum:  $\text{ceil}(N / 2) = 3$
- Zwischen Partitionen ist keine Kommunikation mehr möglich.

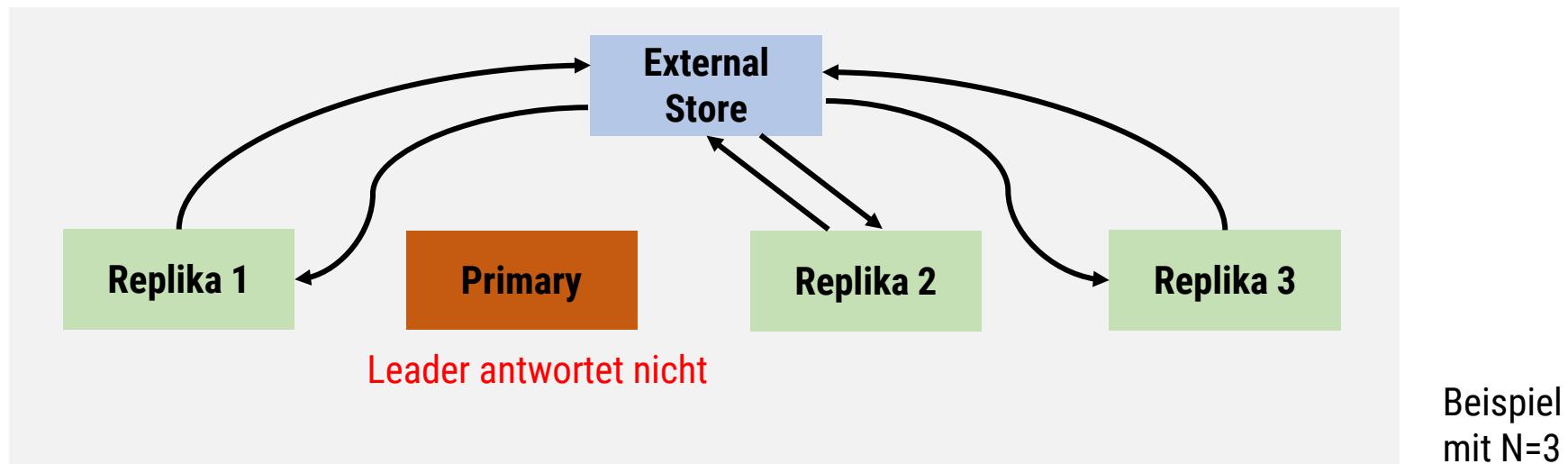


# Leader-Election mittels externem Store

**Idee:** Kleiner aber ausfallsicher zentraler Store implementiert Quorum-Algorithmus. Dieser Store wird von grösseren System verwendet.

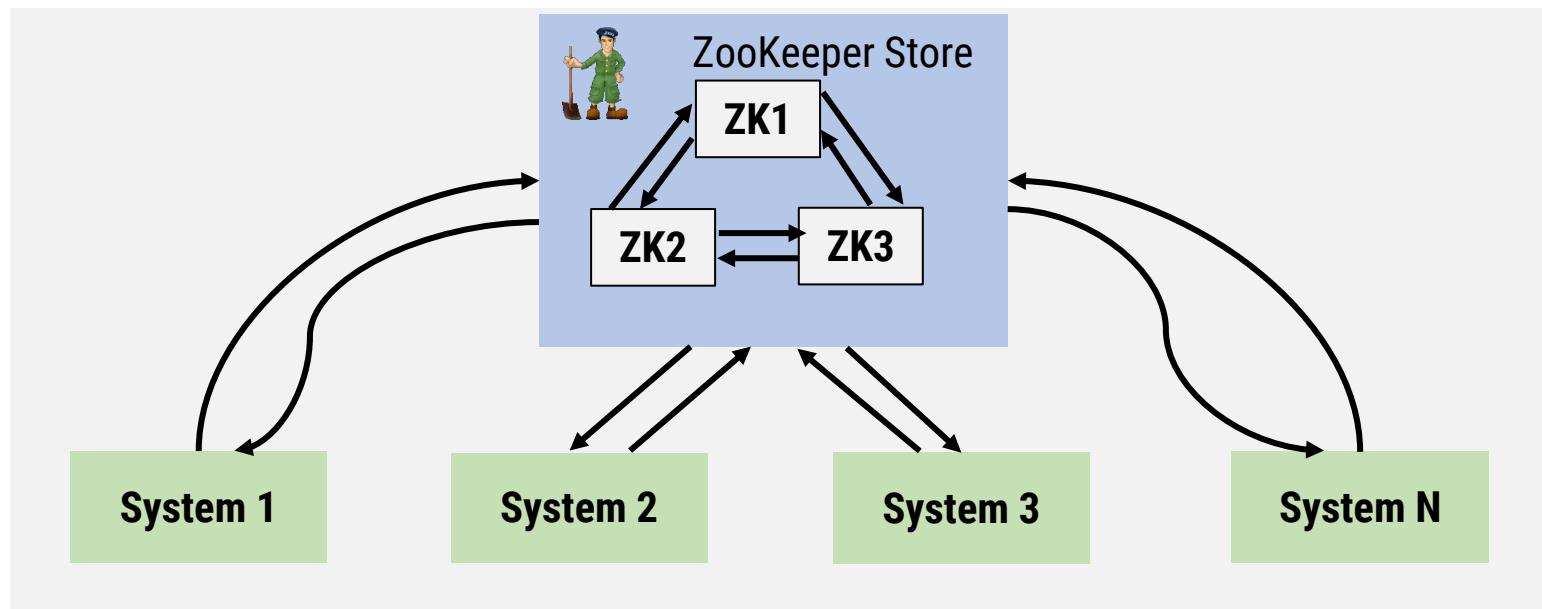
- **Beispiele:** ZooKeeper (generischer Store in Java) oder etcd (Kubernetes).

**Ablauf:**  $N$  Replika senden Election-Request an zentralen Store, einer der Replika wird als Primary eingetragen.

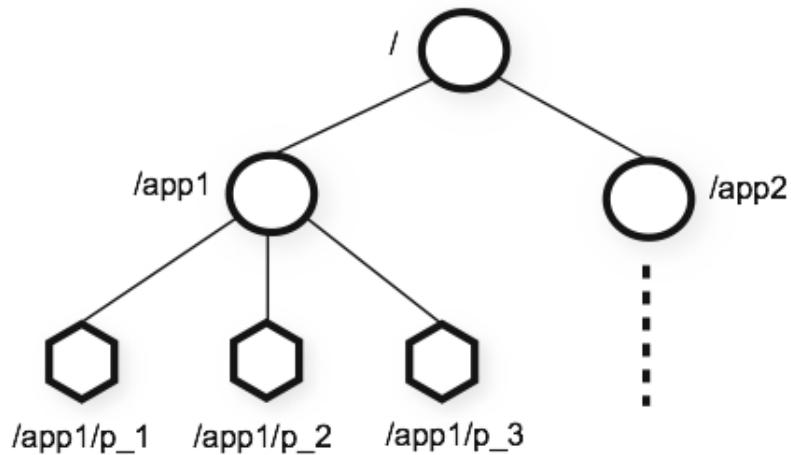


# Beispiel: ZooKeeper

- Zentraler Informationsdienst in Java.
- Ausfallssicher durch Konsensus-Protokoll.
- Bietet verteilten Systemen folgende Funktionalität:
  - Namensdienste (Auffinden von Systemen).
  - Verteilte Synchronisation (z.B. systemübergreifende Locks).
  - Verwaltung von Gruppenzugehörigkeit (z.B. für Replikas).

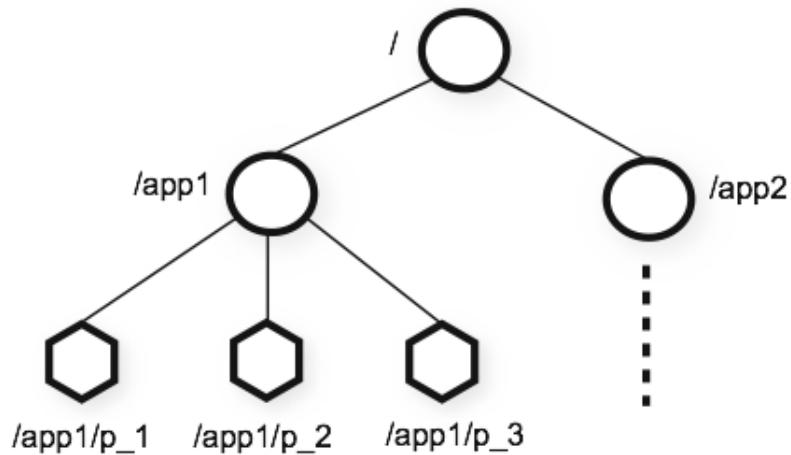


# ZooKeeper: Hierarchischer Namesraum als Store



- ZooKeeper implementiert hierarchischen Namensraum mittels einer Baumdatenstruktur.
- Nodes des Baums sind können Informationen (bytes) und/oder weitere Kinder haben.
- Die Nodes sind PERSISTENT oder EPHEMERAL:
  - PERSISTENT: Überdauern Verbindungsabbruch durch Client.
  - EPHEMERAL: Gelöscht nach Verbindungsabbruch durch Client.

# ZooKeeper: Beispieloperationen auf Store



- Erstelle persistenten Node /app1/p\_3 mit Inhalt myData (byte-Array) und erlaube Zugriff von allen Nodes (Ids.*OPEN\_ACL\_UNSAFE*):

```
zk.create("/app1/p_3", myData, Ids.OPEN_ACL_UNSAFE, CreateMode.PERSISTENT);
```

- Frage Inhalt von Node /app2 ab (nachträgliche Änderungen, lösen Callback setChanged() aus, in stat werden Statistiken geschrieben.

```
byte[] data = zk.getData("/app2", event -> setChanged(), stat);
```

# Beispiel (Auszug): Leader-Election mittels ZooKeeper

```
public final static String PATH = "/myAppLeader";
public final String instance_name = ... ; // unique instance name
public String primary_name = null; // primary name (not known at startup)

public void getOrBecomeLeader() {
    try {
        zk.create(PATH, instance_name.getBytes(), Ids.OPEN_ACL_UNSAFE,
                  CreateMode.EPHEMERAL);
        primary_name = instance_name;
    } catch (KeeperException.NodeExistsException e) {
        try {
            byte[] data = zk.getData(PATH, event->getOrBecomeLeader(), stat);
            primary_name = new String(data);
        } catch (KeeperException.NoNodeException ex) {
            getOrBecomeLeader(); // Leader has changed
        }
    } catch (KeeperException | InterruptedException ex) {
        throw new RuntimeException(ex); // must not occur
    }
}
```

Erstelle an Verbindung gebundenen Node.  
Falls erfolgreich: werde Primary.

Anderfalls: Node  
existiert: Frage Inhalt  
(PrimaryName) ab

# Zusammenfassung

- CAP-Theorem zeigt Grenzen für verteilte Systeme auf bzgl. Konsistenz, Verfügbarkeit und Partitionstoleranz.
- Konsistenz: Alle beteiligen Systeme haben identische und aktuelle Daten.
- Konsistenzgarantien:
  - Sequentielle Konsistenz bedingt eine zeitliche Kopplung der Systeme.
  - Eventual Consistency erlaubt Systeme zu entkoppeln.
- Replikation:
  - Primary-Backup-Protokoll für on-the-fly Replikation.
  - Standardimplementierungen mit verschiedenen Variationen.
- Ausfallsicherheit mittels Replikas:
  - Erkennung des Ausfalls des Primary mittels Heartbeat.
  - Wahl des neuen Primary: Entweder mittels Consensus oder zentralem Store.

# Literatur

- Distributed Systems (3rd Edition), Maarten van Steen, Andrew S. Tanenbaum, Verleger: Maarten van Steen (ehemals Pearson Education Inc.), 2017.

# **Fragen?**

Verteilte Systeme und Komponenten

# **Deployment**

Roland Gisler

# Inhalt

- Deployment Grundlagen
- Deployment Diagramme
- Umfang des Deployments
- Releases und Versionierung
- Technisches Deployment (Anhand von Java)
- Beispiel 1: VSK-Logger-(Server-)Projekt
- Beispiel 2: JavaFX-Projekt (z.B. DemoApp oder LoggerViewer)
- Beispiel 3: Deployment als Container

# Lernziele

- Sie kennen die verschiedenen Aspekte die es beim Deployment zu beachten gilt.
- Sie können verstehen einfache Deploymentdiagramme und können diese erstellen.
- Sie kennen das dreistellige Versionsschema nach «semantic versioning» und können es anwenden.
- Sie kennen Sinn und Zweck eines Binär-Repository und können dieses nutzen.
- Wie kennen verschiedene Deployment-Arten von Java und können diese umsetzen.

# **Deployment Grundlagen**

# Übersicht: Deployment

- **Verteilung:** Verteilung der Software über Downloads / Datenträger oder SMS (Software Management System) oder z.B. OpenWebStart\* (Java), inkl. Dokumentation.
- **Installation:** Kopieren der nötigen Dateien an die vorgesehenen Orte und Registrieren der Anwendung, allenfalls überprüfen, ob das Zielsystem für die Anwendung geeignet ist.
  - Hardwareausstattung, Betriebssystemversion etc.
- **Konfiguration:** Einstellungen der/des Programme(s) auf Benutzer, Netzwerkumgebung, Hardware etc.
- **Organisation:** Planung, ggf. Produktion, Information (Marketing), Schulung (Mitarbeiter\*innen), Support bereitstellen

# Wann findet Deployment statt?

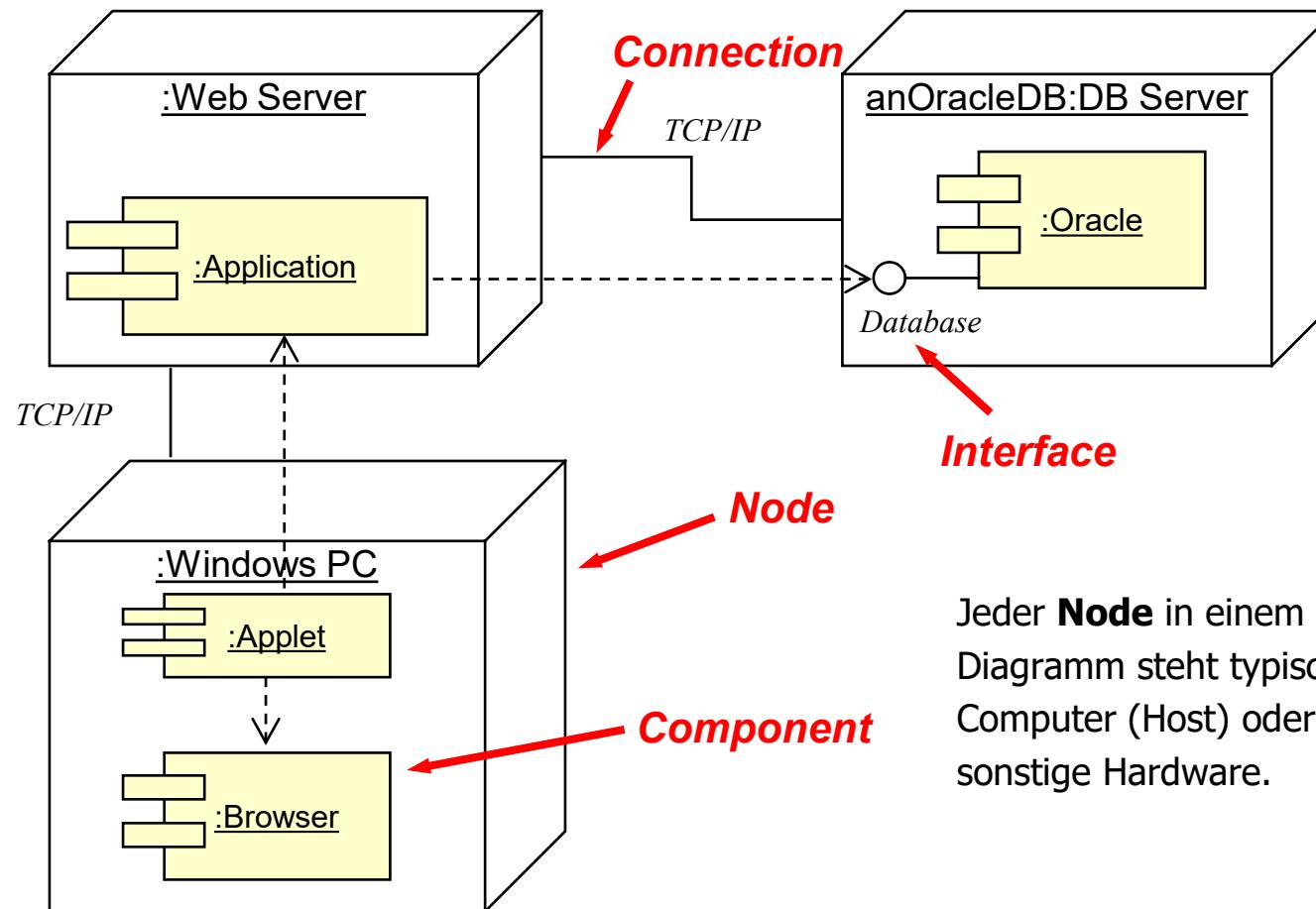
- Deployment findet natürlich am Ende des Projektes statt.
- Aber: Aufgrund iterativer und agiler Entwicklungsmodelle soll das Deployment kontinuierlich (und früher) stattfinden!
  - CI/CD(elivery) erfordert auch Continuous Deployment!
- Einzelne Build-/Sprint-/Iterationsergebnisse fortlaufend deployen.
  - z.B. auf interne Testumgebungen oder direkt beim Kunden (Alpha, Beta, Release Candidate etc.).
  - sogar bis in die Produktion mittels einer Build-Pipline!
- Continuous Delivery (CD)
  - Analogie zum Testen, vgl. Continuous Integration!
  - Nutzung von DevOps Technologien, Infrastructure as Code.
- Staging: Deployment auf unterschiedliche Umgebungen!
  - Entwicklung, Test, Integration, Vor-Produktion, Produktion.

# Deployment - Umfang

- Technische Aspekte:
    - Deployment Diagramme (Zuordnung Komponenten / Hardware)
    - Installations- und Deinstallationsprogramme / -skripte
    - Konfiguration (Default~, Beispiel~ etc.)
    - Installationsmedium
    - Repositories (Ablage der Binaries)
  - Organisatorische Aspekte:
    - Konfigurationsmanagement: Welchen Komponenten bilden welchen Release (Baseline) oder BOM (bill of material).
    - Installations- und Bedienungsanleitungen
    - Erwartungsmanagement: Welche Funktionalität ist vorhanden?
    - Bereitstellung von Support (intern und extern, Levels)
- Deployment-Dokumentation als Quelle der Informationen!

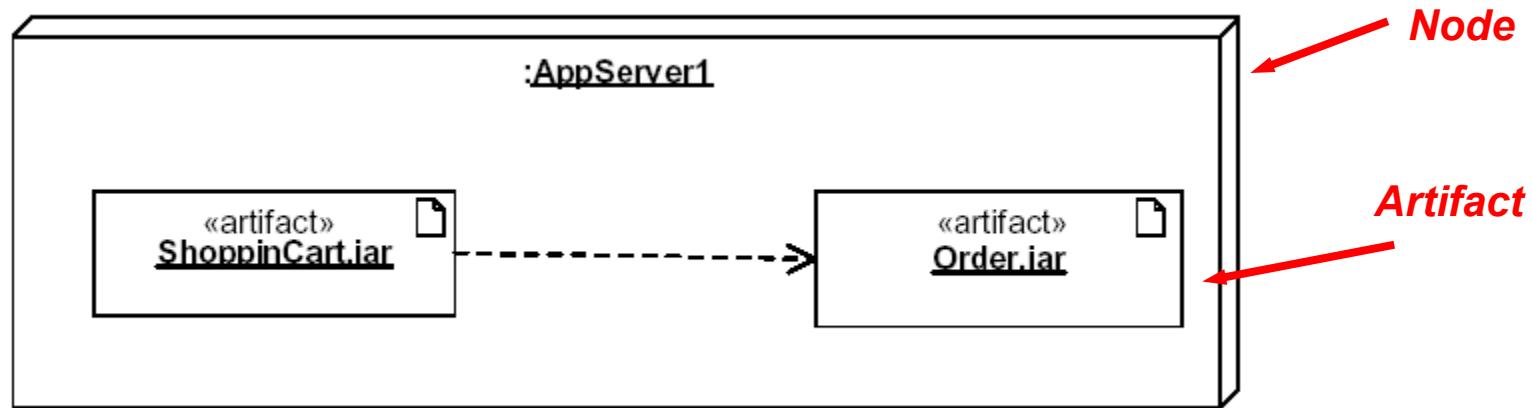
# **Deployment-Diagramme**

# UML 1.x Deployment Diagramm



Jeder **Node** in einem Deployment Diagramm steht typisch für einen Computer (Host) oder eine sonstige Hardware.

# UML 2 Deployment Diagramm



- Node: Stellt einen Computer (Host) oder eine sonstige Hardware dar (identisch zu UML 1.x).
- Artifact: Stellt ein ausführbares Binary, ein Skript etc. dar welches durch die Installation explizit einem Node zugeordnet wird.
  - Wird in einer Deployment-Spezifikation detaillierter beschrieben.

# **Deployment - Aspekte**

# Installation und Deinstallation

- Installation (und Update) einer Software soll möglichst vollständig automatisiert (→Reproduzierbar) sein.
  - Saubere Deinstallation ist ebenfalls wichtig, nicht vergessen!
  - Vollautomatisierte Verteilung möglich (Software-Management).
  - Potential von Package-Management Systemen.
- Sehr unterschiedliche Kundenbedürfnisse / Zielgruppen:
  - **Endbenutzer\*in:** Grafische, interaktive GUI-Installation für Endanwender auf dem Desktop.
  - **Administrator\*in:** Möglichst Script-basierte, durch Parametrisierung voll automatisierte Installation auf dem Server für die Administrator\*in.
  - **Entwickler\*in / Tester\*in:** Spezielle Distributionen, entweder manuell (download) oder über (zentrale) Repositories.

# Konfiguration von Anwendungen

- Ein klassischer Zielkonflikt!
- Die neu installierte Anwendung soll:
  - 1. möglichst sofort «out-of-the-box» lauffähig sein, ...
  - 2. sich an verschiedenste Umsysteme anpassen können, und...
  - 3. trotzdem möglichst einfach aktualisierbar sein.
- Typische Beispiele / Anforderungen:
  - Datenbankanwendung: Lauffähigkeit auf einer individuellen, bestehenden DBMS-Umgebung.
  - Logging / Audit: Einsatz unterschiedlicher Logging- und Überwachungs-Mechanismen/Frameworks.
  - Security: Support verschiedener Authentifizierungs- und Autorisierung-Techniken (z.B. LDAP, Kerberos etc.).

# Konfigurationsmanagement

- Mit der Zeit haben verschiedene Kunden unterschiedliche Versionen einer Software (typisch für Individual-Entwicklungen).
  - Welche Version läuft bei welchem Kunden?
  - Welcher Kunde hat welche Lizenzen?
- Verschiedene Kunden haben unterschiedliche Produkte und Versionen der Umsysteme (z.B. Datenbank) und der Hardware (z.B. Einfluss auf Performance)
  - Wer hat welche Konfiguration? Mit welchen Komponenten?
  - Welche Kombinationen sind überhaupt lauffähig?
- Ist ein Update von jeder existierenden Konfiguration möglich?
  - Müssen bestimmte Abfolgen eingehalten werden?
  - Wurden die unterschiedlichen Szenarien auch getestet?

# Deploymentdokumentation / Manuals

- **Release Notes** - Erste Quelle über/zu:
  - Neuen Funktionsumfang.
  - Veränderte / zusätzliche Vorbedingungen.
  - Neue / veränderte Datenformate oder Protokolle.
  - etc.
- **Installationsanleitung**
  - Haben sich HW- oder SW-Voraussetzungen geändert?
  - Varianten unterschiedlicher Konfigurationen berücksichtigen.
  - Müssen bestimmte Abfolgen eingehalten werden?
  - Tipp: Installation möglichst automatisieren. → weniger Doku.
- **Bedienungsanleitung / User-Manual**
- **VSK-Logger: KISS, muss aber vorhanden sein!**



# **Releases und Versionierung**

# Releases und Versionierung

- Deployment findet mit einem wohldefinierten Release statt
  - Eindeutige Bezeichnung und Version
    - technische Version ist die eindeutige Identifikation.
    - 'Tagging' im Versionskontrollsystem!
    - 'Marketing-Version' unbedingt trennen, als Ergänzung.
- Bewährt: Dreistellige Version: **x.y.z** (z.B. **7.2.3**) mit Semantik
  - Semantic Versioning <http://semver.org/>
- Anhand der Version soll möglichst einfach und klar ersichtlich sein was sich prinzipiell verändert hat:
  - Änderungen, Erweiterungen oder Korrekturen
  - Version hat speziell bei technischen Releases (z.B. Frameworks, Libraries, Komponenten) eine sehr wichtige Aussage

# Semantic Versioning - Repetition

- **Major**-Release (**X.x.x**): Veränderungen in der API, in der fachlichen Funktion oder in der Konfiguration, welche zu früheren Versionen nicht mehr kompatibel sind und somit Anpassungen notwendig machen
- **Minor**-Release (**x.X.x**): Erweiterungen in der API, der fachlichen Funktion oder der Konfiguration, welche aber vollständig Rückwärtskompatibel sind und (zumindest ohne Nutzung derselben) keine Anpassungen notwendig machen
- **Bugfix/Maintenance**-Release (**x.x.X**): Reine Korrekturen oder Änderungen in der Implementation, voll rückwärtskompatibel, keinerlei neue Funktionen, keine veränderte Funktionen, direkter Einsatz möglich

# Alternative: Time-Based Release Versioning

- Man hält einen festen «Takt» an Release-Terminen ein.
- Sinnvollerweise macht die Versionsnummer deutlich, um welchen Zeitpunkt es sich handelt.
- Beispiel Ubuntu-Releases: ..., **21.04**, **21.09**, **22.04**, **22.09**, ...
  - YY.MM-Format: Jahr und Monat
  - Zur Eindeutigkeit häufig mit «Buildnummer» ergänzt.
- Einsatz bei «grossen» Produkten und/oder Marketing-Versionen sinnvoll, aber weniger bei Libraries/Komponenten.
  - War Ende des 20. JH sehr beliebt (V2000), Anfang 21. JH eher unbeliebt (**0x**-er Jahre!), und wird jetzt sehr beliebt... ☺
- Empfehlung: Versionieren Sie mit **Vernunft!**
  - Unterscheiden Sie zwischen technischer und Marketing-Version!

# Time-Based Release Versioning bei Oracle / Java

- Java-Versionen seit September 2017 im Format:

**\$FEATURE.\$INTERIM.\$UPDATE.\$PATCH**

- Bedeutung der Stellen:

- **FEATURE**: Inkrementeller Counter, aktuell bei **16**.
  - **INTERIM**: Bleibt derzeit bei **0**.
  - **UPDATE**: Inkrementeller Counter für Updates.
  - **PATCH**: Optionale Patch/Build-Nummer (**17.0.3.1** erstmalig)

- Aktueller Release-Plan:

- Feature Releases alle 6 Monate (jeweils März und September).
  - Updates jeweils +1/+3 Monate (Apr./Jul. und Okt./Jan.).
  - Alle **zwei** bis **drei** Jahre wird der September-(Feature-)Release als ein **LTS** (Long Time Support) deklariert (Aktuell: **17.0.5**).
  - Nächster LTS für 2023 angekündigt, somit vermutlich 21.0

# Versionierung - Release-Notes

- Sauberes Nachführen von allen Änderungen, Erweiterungen und Korrekturen.
  - Nachvollziehbare Entwicklungsgeschichte.
  - Meist manuell nachgeführt, da qualitative Aussage.
  - Evt. unterstützt durch Issue-Tracking-Systeme.
    - Bugzilla, JIRA, Mantis, Trac, Redmine etc.
    - Direkter Bezug auf Change-Request oder Bug.
- Für Entwickler\*innen die zentrale Informationsquelle um die Möglichkeit bzw. die Notwendigkeit einer Migration auf eine neue Version (und das damit verbundene Risiko) einschätzen zu können.
  - Wenn Sie selber Release-Notes lesen, sollten Sie auch bereit sein, für Ihr Produkt ein Release-Notes zu schreiben! ☺

# **Technisches Deployment (Java)**

# Techniken und Methoden – Beispiel: Java

- Verschiedene Plattformen (OS) und Applikationsarten.
  - Java: «*write once, run anywhere*» (wora).
  - Einzelplatzanwendung (ein Host) - für Endanwender.
  - Serveranwendung (JEE, Applicationserver) - für Operating.
  - Library/Framework - für Entwickler
- Verteilung vieler einzelner \*.class-Dateien ist in der Regel nicht praktikabel (aber es gibt Ausnahmen).
- Basiskonzept: Zusammenfassung von \*.class-Dateien und Ressourcen in unkomprimiertem Zip-Format.
  - Java **ARchive** (jar-Datei) oder Java **MODul** (jmod-Datei).
- Erweiterte Konzepte (für Webcontainer und Applicationserver):
  - **WAR (Web ARchive**, mit META-INF/web.xml etc.)
  - **EAR (Enterprise ARchive**, mit META-INF/application.xml)

# Deploymentziele und Projektarten

- Je nach Produkt unterschiedliche Techniken zur Verteilung.
- Applikation / Anwendung für Endbenutzer\*in: Verteilung in binärer Form, evt. mit automatischem Installationsprogramm (setup.exe) und Anleitung/Manual, ggf. sogar inkl. JRE.
  - Separierte JAR-Dateien (Komponenten) oder Single-JAR.
- Server-Applikation für Operating: Klassisch meist separierte JAR-Dateien für gezielte Updates, bei Microservices eher Single-JAR.
  - Immer beliebter als Alternative: Container-Deployment.
- Library/Komponente, für Entwickler\*innen: Typisch Download, binäres Repository (z.B. Maven-Repo)
  - Mit konsistenter Versionssemantik (z.B. **x.y.z-SNAPSHOT**).
  - Inklusive Metadaten in strukturierter Form, für automatisches Dependency-Management (z.B. **pom.xml**).

# JAR-Datei (Java ARchive)

- Eine vollständige Applikation
  - kann selber aus **1..n** einzelnen JAR's bestehen.
  - kann zusätzlich **0..n** Dependency-JAR's benötigen.
- JAR's können extern (Name) oder intern (Manifest) versioniert sein.
  - Bei >1 JAR-Dateien ist ein Classpath notwendig!
- Eine Menge von JAR's:
  - Einzelne Komponenten/Libraries z.B. für Bugfixing leicht austauschbar (+) vs. Dynamik des Classpath. (-)
- Einzelnes JAR (Shade-/Uber-/Fat-JAR):
  - Zusammenfassung des Inhaltes mehrerer JAR-Dateien in einem einzigen JAR zwecks einfacherem Deployment.
  - Einfachheit (+) vs. Redundanzen der Dependencies (-)
  - Problematik der Neusignierung und META-INF (Manifest etc.)

# Modularisierung seit Java 9 (Project Jigsaw)

Mit Java 9 (September 2017) wurden endlich die langersehnten technische Möglichkeiten zur «echten» Modularisierung mit Java implementiert. Dabei standen drei Ziele im Vordergrund:

- **Reliable Configuration:** Den fehleranfälligen Classpath durch den auf Modul-Abhängigkeiten basierenden Modul-Path ablösen.
- **Strong Encapsulation:** Ein Modul definiert explizit sein öffentliches API. Auf alle restlichen Klassen ist von Ausserhalb kein Zugriff mehr möglich (auch wenn **public**)!
- **Scalable Platform:** Die Java-Plattform selber wurde modularisiert, so dass für Anwendungen individuell angepasste, schlankere Runtime-Images gebaut werden können.

# Modularisierung in Java 9 - Umsetzung

- Java-Packages können neu in Modulen zusammengefasst werden.
  - Optionale, zusätzliche Strukturebene in der Dateiablage.
  - Eindeutige Namensgebung nötig (analog zu Packages).
- Pro Modul wird ein `module-info.java` definiert. Darin werden explizit die Import, Exports und Abhängigkeiten definiert!
  - Somit wird eine Designverifikation zur Compile-Time möglich!
- Zusätzlich findet beim Start einer Applikation eine Laufzeitprüfung statt, ob alle notwendigen Komponenten vorhanden sind.
  - `ClassNotFoundException` Exeption sollte somit nicht mehr auftreten!
- Das Ende der «JAR-Hell»: Es wurde ein neues Format (`jmod`) definiert, der Classpath wird durch den Modul-Path abgelöst.
- Und das alles vollständig rückwärtskompatibel! ☺

# Beispiel eines sehr einfachen modul-info.java

- Sehr einfaches Beispiel eines `modul-info.java`
  - Ist eine «spezielle» Klasse.
  - Hält sich (analog zu `package-info.java`) bewusst **nicht** an die Namenskonvention von Java, so dass es nirgends zu Konflikten kommt.

```
module ch.hslu.sort.quicksort {  
    exports ch.hslu.sort.quicksort.impl;  
    requires ch.hslu.sort.api;  
}
```

# Verteilung über Binär-Repositories

- Ein **Binär**-Repository
  - hat **nichts** mit einem VCS/SCM (git etc.) zu tun!!
  - strukturierte Ablage von Binaries (JAR, WAR, EAR, JMOD etc.).
  - Primär für Entwicklungsprozess (Dependency-Management).
- Ursprünglich als «Maven-Repository» entstanden (Apache Maven).
  - <https://repo1.maven.org/maven2/>
  - Basierte ursprünglich auf einer Verzeichnisstruktur im FS.
  - Heute meist als spezielle Serversoftware implementiert, erlaubt z.B. effizientere Suche, Management und Analysen.
- Produkte für on-site-Betrieb (typisch in Organisationen).
  - Beispiele: JFrog Artifactory, Apache Archiva, Sonatype Nexus etc.
- Auch als reine Cloud-Dienste, z.B. integriert in Codehosting-Systeme wie <https://gitlab.com/> (Package Repository).

# Verteilung für Java: HSLU Nexus Repository

- HSLU EnterpriseLab bietet seit HS17 ein Repository auf Basis von Nexus zur Verfügung.
  - <https://repohub.enterpriselab.ch/>
- Über das Installierte **settings.xml** (Maven-Konfiguration) verwenden Sie dieses Repository nur lesend.
  - Cache/Mirror für externe Repositories → Schneller.
- Ausschliesslich der (Jenkins-)Buildserver verfügt über die notwendigen Rechte um in das Repository zu schreiben.
  - Projekte wurden laufend in das Repository deployed.
- Achtung: Unterschiedliches Verhalten des Deployments und der Dependency-Resolution bei **Releases** oder **SNAPSHOTS!**
  - Haben Teams, die den «SNAPSHOT»-Appendix entfernt haben, etwas festgestellt?

# Deployment bei Open Source Projekten

- Deployment erfolgt häufig über zwei verschiedene Distributionen:
  - **Binär**-Distribution: Enthält binäre Runtime plus Dokumentation, direkt einsetzbar. Häufig als ZIP-Datei zum Download.
  - **Source**-Distribution: Enthält nur den Quellcode und alle notwendigen Buildartefakte. Heute häufig über VCS-Zugriff.
- Aus der Source-Distribution sollte die Binär-Distribution jederzeit wieder erstellt werden können.
  - Entwicklungswerzeuge (JDK etc.) vorausgesetzt
- Alle Distributionen werden sauber versioniert
  - Ideal / Empfehlung: → Semantic Versioning
- **Hinweis:** Binäre Repositories unterstützen auch die Verteilung von Quellen und JavaDoc(\*-source.jar und \*-javadoc.jar).
  - Quellpacket entspricht **nicht** einer Source-Distribution!

# **Beispiel 1: VSK-Logger-(Server-)Projekt**

# VSK-Logger - Deploymentvarianten

- Jede Applikation bzw. Komponente bzw. Library ist ein eigenes (Teil-)Projekt und baut je ein eigenes, versioniertes JAR.
  - Nebenbei: Das wären in Zukunft die (gröbstmöglichen) Module.
- Variante **1**: Start der Applikationen (z.B. der Server) mit Classpath welcher alle notwendigen JAR-Dateien enthält.
  - Variante **1a**: Argument **-cp** beim Start (z.B. über Shell-Script).
    - Einfache Erweiterung des Classpath, z.B. für Ressourcen.
    - Einfacher Austausch (!) einzelner JAR-Dateien (z.B. Bugfix).
  - Variante **1b**: Class-Path-Eintrag in META-INF/MANIFEST.MF in der JAR-Datei des Anwendung.
    - JAR (Manifest) muss bei Änderung neu gebaut werden.
- Variante **2**: FAT-JAR, welches ALLE Klassen enthält (Shade-JAR).
  - Muss immer komplett neu gebaut werden → hier **nicht** sinnvoll!

# VSK-Logger – JAR-Dependencies auflösen (kopieren)

- Das «Maven Dependency Plugin» hilft uns:

<https://maven.apache.org/plugins/maven-dependency-plugin/index.html>

- Beispiel: Liste der Abhängigkeiten anzeigen (ohne test-Scope):

```
mvn dependency:list -DincludeScope=compile
```

- Resultat (Beispiel für g01-demoapp):

```
The following files have been resolved:
```

```
ch.hslu.vsk22hs.g01:g01-loggercommon:jar:1.0.0-SNAPSHOT:compile -- module g01.loggercommon (auto)
ch.hslu.vsk22hs.g01:g01-loggercomponent:jar:1.0.0-SNAPSHOT:compile -- module g01.loggercomponent (auto)
ch.hslu.vsk22hs:loggerinterface:jar:1.0.0:compile -- module loggerinterface (auto)
ch.hslu.vsk:stringpersistor-api:jar:5.0.4:compile -- module stringpersistor.api (auto)
ch.hslu.vsk22hs.g01:g01-stringpersistor:jar:1.0.0-SNAPSHOT:compile -- module g01.stringpersistor (auto)
```

- Mit Hilfe dieses Plugins lassen sich die Dependencies auch automatisch aus dem Binär-Repository zusammenkopieren:

```
mvn dependency:copy-dependencies<
-DincludeScope=compile
```

- Resultat liegt danach im Verzeichnis **./target/dependency**

# VSK-Logger – Starten der Applikationen – Variante 1

- Ausserhalb der IDE kann eine Applikation mit Hilfe von Apache Maven gestartet werden (also noch mit Hilfe des Build-Tools).

- «Maven Exec Plugin» hilft hier weiter:

- <https://www.mojohaus.org/exec-maven-plugin/>

- Beispiel (für g01-demoapp):

```
mvn exec:java  
      -D"exec.mainClass=ch.hslu.vsk.demoapp.DemoApp"
```

- Voraussetzung: Apache Maven ist installiert, der lokale Build war erfolgreich, und alle Buildartefakte liegen somit vor.
- Typisch nur für Entwickler\*innen-Arbeitsplätze nutzbar/sinnvoll.  
→ Wegen der Abhängigkeit zum Buildtool sicher **keine** Lösung für Endbenutzer\*innen.

## VSK-Logger – Starten der Applikationen – Variante 2

- Wenn die JAR-Dateien alle vorliegen, kann die Applikation auch direkt (nur über Java) gestartet werden!
- Beispiel (für g01-demoapp):

```
java -cp "./g01-demoapp-1.0.0.jar;./dependency/*"  
      ch.hslu.vsk.demoapp.DemoApp
```

- Voraussetzung: Dependencies wurden vorher kopiert; aktuelles Verzeichnis ist **./target** (dort befindet sich das Haupt-JAR).
- Befehl am einfachsten in ein Shell-Script (cmd, sh etc.) ablegen!
  - Sieht «rustikal» aus, ist aber einfach, robust und transparent!
- Achtung: Pfadangaben sind plattformspezifisch, unter Unix/Linux ist z.B. das Pfadtrennzeichen ein Doppelpunkt (:), bei Windows ein Semikolon (;).

## **Beispiel 2: JavaFX Applikation**

# Java FX – Modularisierte Architektur

- Java FX ab Version 11 (und höher) nutzt die neuen (seit Java 9) verfügbaren Techniken zur Modularisierung.
- Somit muss man sich bei einer JavaFX-Anwendung «zwingend» mit Modularisierung auseinander setzen, auch wenn die restliche Applikation diese Technik (noch) nicht einsetzt.
- Seit dem beschleunigten Releasing von Java (seit Version 9) herrscht eine relativ grosse Dynamik, und man «rennt» in den Buildsystemen und den IDE's der Realität etwas hinterher.

→ **Wichtig:** Das ist **kein** Nachteil oder Kritik an Java FX.

Es kommen hier «einfach» ein paar Dinge zusammen, um welche man sich bisher (bei Java) nicht kümmern musste.

- Potential offenbart sich derzeit leider noch nicht so deutlich.

# Java FX – Starten einer Applikation mit Maven

- Für Maven existiert ein Plugin von OpenJFX, welches den korrekten Start einer (teil-)modularisierten JavaFX-Applikation automatisiert.
  - **org.openjfx:javafx-maven-plugin:0.0.8** (Nov. 2022)
  - Befehle `mvn javafx:run [-debug]` oder `mvn javafx:jlink`
- Mittels `-debug`-Option von Maven kann man sich den Startbefehl (sinngemäss) herausholen. Beispiel (konzeptionell):

```
java
--module-path
javafx-base-17.0.5-win.jar;javafx-base-17.0.5.jar;
javafx-controls-17.0.5-win.jar;javafx-controls-17.0.5.jar;
...
--add-modules javafx.base,javafx.controls, ...
-cp app.jar;log4j-api-2.19.1.jar; ...
ch.hslu.demo.GuiStartClass
```

- Gute Hilfestellung unter <https://openjfx.io/openjfx-docs/#maven>

# **Deployment als Container**

# VSK-Logger – Deployment als FAT-JAR in Container

- Vorteile:
  - Bau des Images und Start-Befehl sehr einfach.
  - Simples Dockerfile zur Konfiguration.
  - FAT-JAR kann auch ausserhalb des Containers sehr einfach genutzt/getestet werden.
- Nachteile:
  - FAT-JAR kann relativ gross werden → Buildprozess (zusammenkopieren) wird langsam.
  - Fetter Layer im Container-Image (kleine Änderung, trotzdem ganzes JAR/Layer neu erstellt)
  - Problematik mit den Manifesten und Konfigurationen.
- Beispiel: vsk-echoserver, Docker-Image mit Fabric8-Plugin erstellt.
  - siehe Struktur des Images – Live-Demo (dive)!

# VSK-Logger – Deployment als «layered» Container

- Vorteile:
  - Feingranulare Trennung der Applikation, Dependencies und jeweilige Konfigurationen in getrennten Layern.
  - Unterschiedliche Änderungshäufigkeit (App. versus Deps.) beschleunigt den Build massiv und senkt Ressourcenbedarf.
  - Automatisierung z.B. durch Google JIB-Plugin.
- Nachteile:
  - Bau des Images komplizierter, weil differenzierter.
  - Komplexeres Dockerfile (aber ggf. gar nicht sichtbar).
  - Applikation in dieser Form ohne manuelle Eingriffe nur im/mit Container lauffähig.
- Beispiel: vsk-echoserver, Image mit JIB-Plugin erstellt.
  - siehe Struktur des Images – Live Demo!

# Demo's / Screencast's

- Java Deployment: Starten von Java-Applikationen ausserhalb der Entwicklungsumgebung:

**EP\_40\_SC01\_JavaDeployment.mp4**



# Zusammenfassung

- Verschiedene Aspekte des Deployments:  
Technisch (Verteilung, Installation etc.), Organisatorisch.
- Sinnvolle Deployment-Dokumentation:  
Unterschiedliches Zielpublikum berücksichtigen.
- Semantic Versioning
- Deployment-Techniken für/mit Java
- Spezifische Anforderungen für effizientes Deployment in/mit Container-Images.

**Fragen?**

Verteilte Systeme und Komponenten

# Skalierung und Verteilung

Martin Bättig



# Inhalt

- Grundlagen der Skalierung und Verteilung
- Lastverteilung mittels Reverse-Proxys
- Lastverteilung mittels In-Memory Datagrids
- Zusammenfassung

# Lernziele

- Sie kennen die Grundlagen der Skalierung und Verteilung.
- Sie wissen was ein Reverse-Proxy ist und wie dieser zur Lastverteilung eingesetzt werden kann.
- Sie verstehen, wie man mittels Kombination eines Load-Balancers und Replikas eine Skalierung erzielt.
- Sie können nachvollziehen wie ein In-Memory Data Grid (IMDG) Daten speichert und in einem Cluster verteilt und wissen in den Grundzügen, was passiert, wenn im Cluster ein neuer Knoten hinzukommt oder wegfällt.
- Sie können einfache Code-Beispiele am Beispiel von HazelCast nachvollziehen und selbst welche mit wenigen Zeilen schreiben.

# **Grundlagen der Skalierung und Verteilung**

# Quiz zum Aufwärmen

Angenommen Sie hätten vier Serversysteme zur Verfügung mit jeweils:

- 32GB Arbeitsspeicher (Zugriffszeiten im ns-Bereich).
- 1TB Harddisk (Zugriffszeiten im ms-Bereich).
- Jeweils identische und schnelle Netzwerkanbindung mit 10GBit/s.

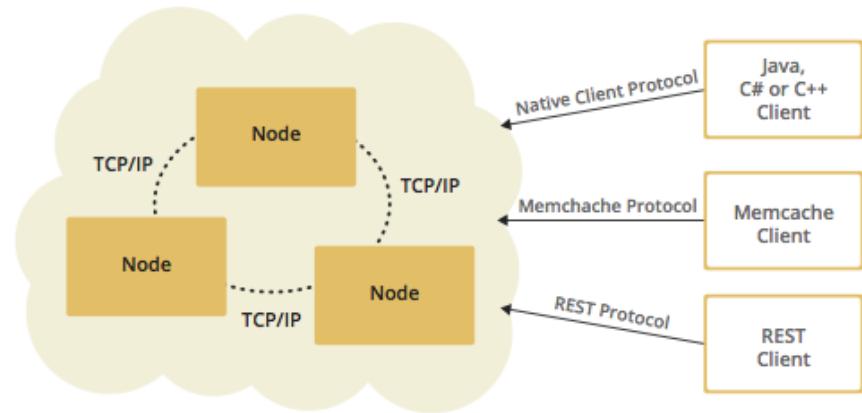
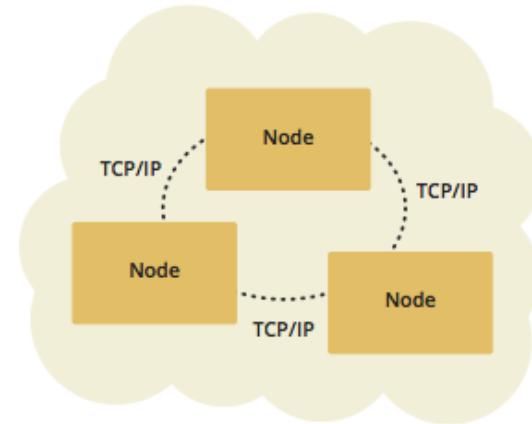
Sie möchten mittels diesen Systemen Bilddaten ausliefern:

- Ein Bild ist maximal 10M gross.
- Sie haben im Total 100GB an Bilddaten.

**Frage:** Wie können Sie diese Systeme programmieren, sodass Sie jedes Bild mit möglichst **mit kleiner Latenz** ausliefern können?

# Topologie verteilter Systeme

- **Embedded:** Für Anwendungen wo asynchrone Ausführung von Task oder Hochleistungs-Computing gefragt sind. Knoten enthalten in diesem Typ sowohl die **Anwendung** als auch die **Daten**.
- **Client / Server:** Für Cluster von Server-Knoten, die erstellt und skaliert werden können. **Clients** kommunizieren mit diesen Serverknoten, um auf die **Daten** zu zugreifen. Es gibt native Clients (Java, .NET und C++), Memcache-Clients und REST-Clients.



# Skalierung verteilter Systeme

## Zustandslose Systeme (alle Anfragen sind unabhängig):

- Mehrere Instanzen einer Serveranwendung starten.
- Jede Anfrage je nach Auslastung an die Instanzen verteilen.  
⇒ Einfache Lastverteilung.

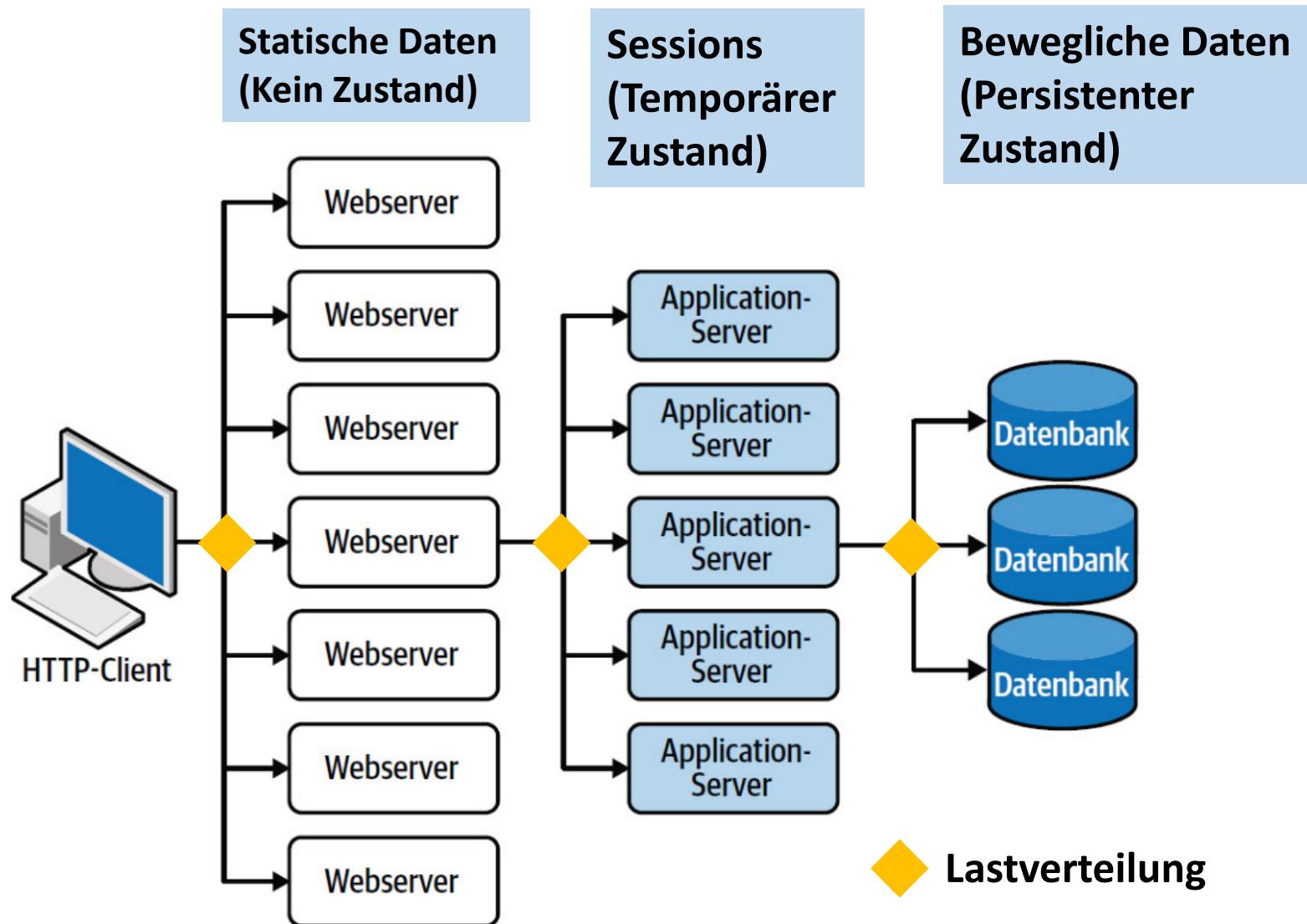
## Zustandsbehaftete Systeme mit temporären Daten:

- Mehrere Instanzen einer Serveranwendung starten.
- Erste Anfrage einer Session je nach Auslastung an eine Instanz N verteilen. Folgeanfragen jeweils wieder an die gleiche Instanz N leiten.  
⇒ Komplexere Lastverteilung oft mittels Inspektion der Kommunikation.

## Zustandsbehaftete Systeme mit persistenten Daten:

- Daten replizieren (ggf. Partitionieren, d.h. zw. Instanzen aufteilen).
- Anfragen an diejenigen Instanzen der Serveranwendung verteilen, welche entsprechende Daten vorhält.  
⇒ Komplexeste Lastverteilung (read vs. write, Konsistenz der Replikas, etc.)

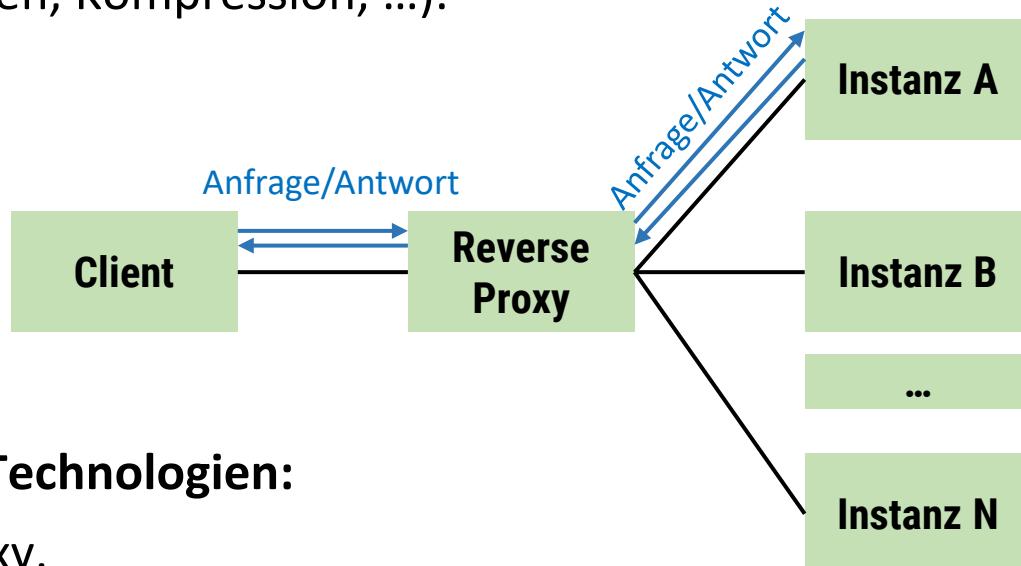
# Beispiel: Skalierung einer klassischen Webapplikation



# **Lastverteilung mittels Reverse-Proxys**

# Reverse-Proxy

- Vorgeschaltete Middleware, welche Anfragen auf verschiedene Instanzen einer Serveranwendung verteilt.
- **Load-Balancing:** Last wird auf mehrere Instanzen verteilt, sodass keine Instanz überlastet ist, solange noch andere Instanzen Kapazität haben.
- Oft weitere Funktionalität z.B. Sicherheit (Verschlüsselung, Monitoring, Metriken, Kompression, ...).



## Beispiel Technologien:

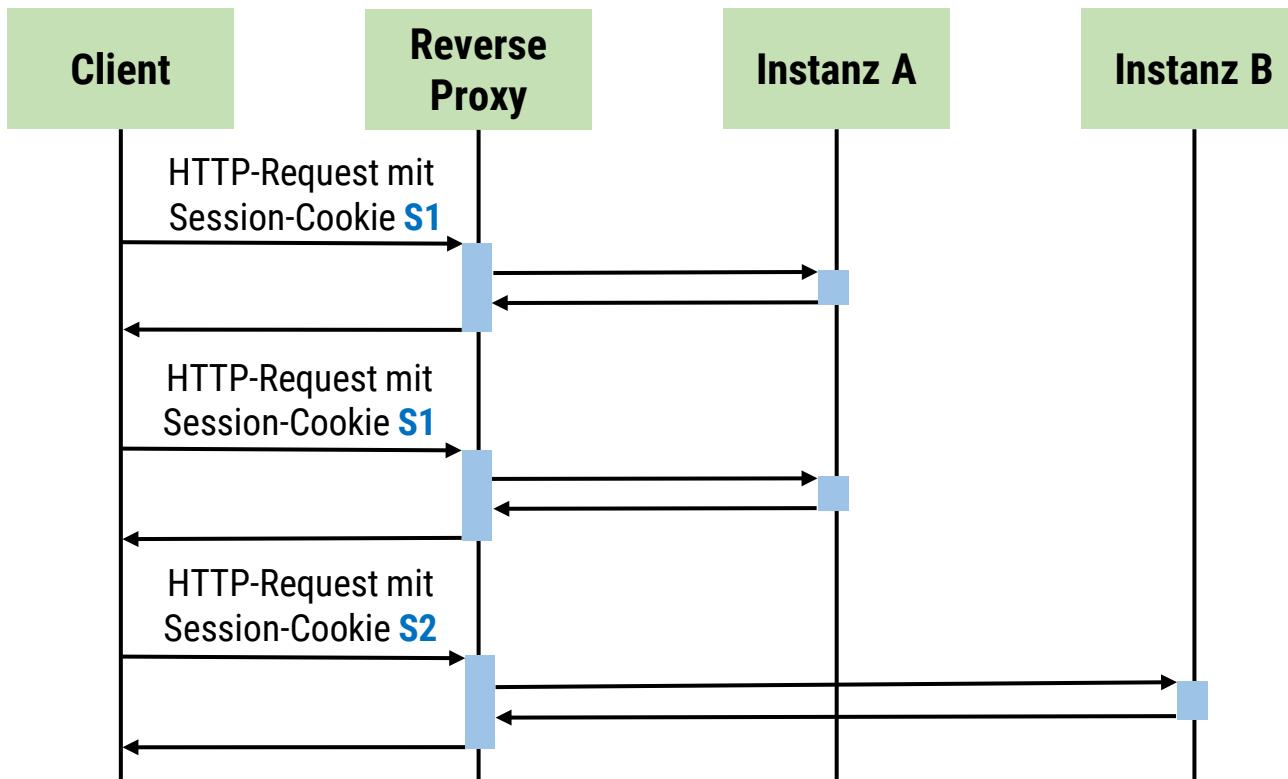
- HAProxy.
- Webserver wie Apache / Nginx
- Træfik

# Lastverteilungsmethoden (Auswahl)

- **Round-Robin:** (Der Reihe nach) Erstelle eine Liste mit allen Instanzen. Die Liste wird abgearbeitet und jede Instanz in dieser Liste erhält eine Anfrage. Hat die letzte Instanz eine Anfrage erhalten, beginnt der Proxy von vorne.  
*Gut bei homogener Last der Anfragen und homogener Systemausstattung.*
- **Anzahl bestehender Verbindungen:** Leite Anfrage zur Instanz mit der geringsten Anzahl Verbindungen weiter.  
*Gut für langdauernde TCP-Verbindungen.*
- **Hash:** Anwendung einer Hash-Funktion auf IP-Adresse des Clients um die Zielinstanz zu bestimmen. Alle Anfragen einer IP-Adresse werden an die identische Instanz weitergeleitet.  
*Einfache Möglichkeit Requests an gleichen Server zu leiten (benötigt jedoch stabile Client-IPs).*

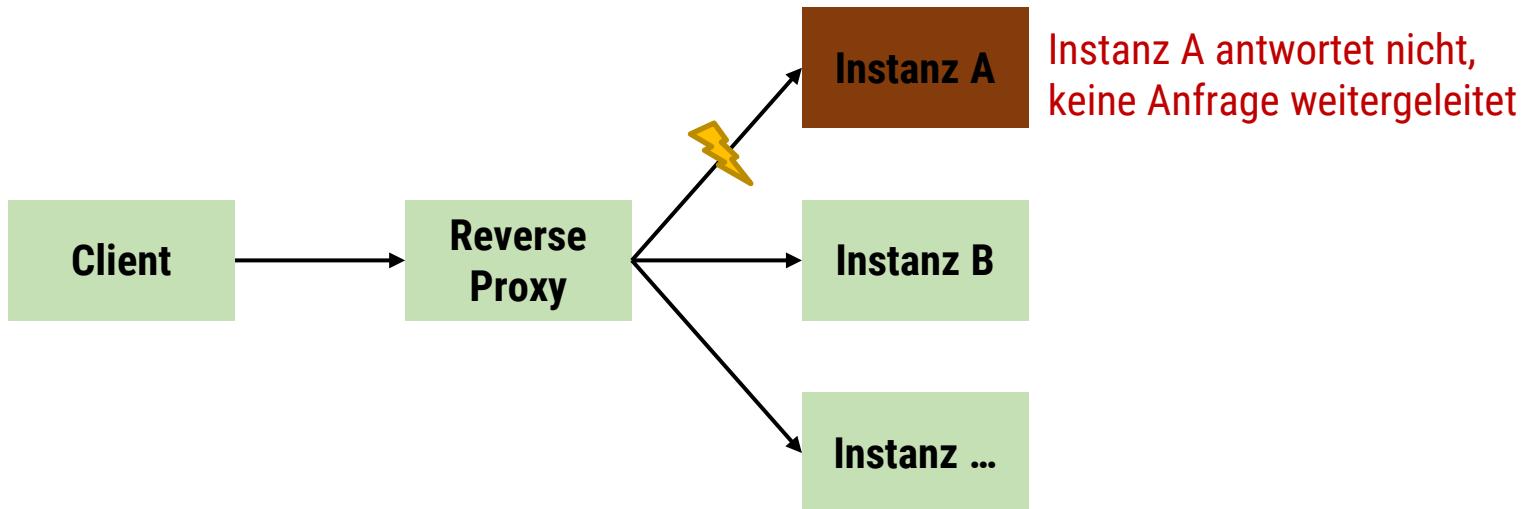
# Zusammenspiel mit Serverapplikation

Sollen mehrere aufeinander folgende Requests unabhängig der TCP-Verbindung zur gleichen Serverinstanz geleitet werden, muss der Reverse-Proxy das Protokoll inspizieren können (Teilverständnis).



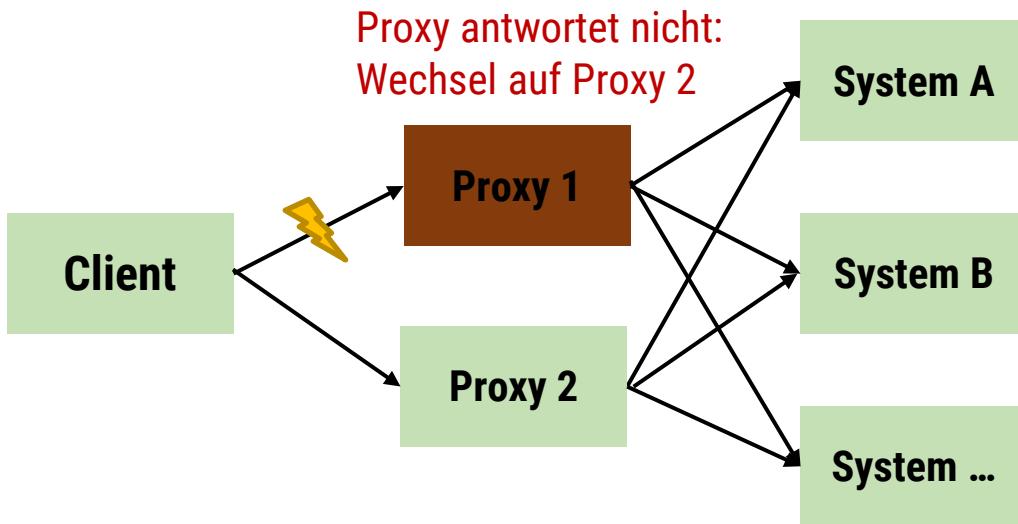
# Ausfallsicherheit auf Seite Serverinstanz

- Load-Balancer testen mittels Health-Checks, ob eine Instanz funktioniert:
  - Keine Weiterleitung von Anfragen an Instanzen, welche nicht antworten.
- Typische Health-Checks:
  - Steht TCP-Verbindung (Transport-Schicht).
  - Beliebiger HTTP-Request (Applikations-Schicht).
  - Ausführen eines Agents (Hilfsprogramm) auf Serverinstanz.



# Ausfallsicherheit auf Seite Proxy

- Einzelner Reverse-Proxy wäre **Single-Point of Failure**.
- Mindestens zwei Instanzen für Ausfallsicherheit benötigt:
  - Kombination mit Hochverfügbarkeitslösungen (z.B. keepalived) und Floating-IPs (IP, welche auf verschiedene Ziele geroutet werden kann).
  - Bei Ausfall eines Proxys wird die IP gewechselt.



# Beispiel: HAProxy

## Eckdaten:

- Beziehen unter: <https://www.haproxy.org>
- Open-Source und kommerzielle Version.
- Architektur: Asynchroner IOs in Kombination mit mehreren Threads.
- Kommerzielle Version hat diverse Erweiterungen (z.B. Routing nach Geolocalization oder Traffic-Mirroring).



## Einfacher Aufbau:

- Einzelnes Binary: haproxy
- mit Konfiguration: Z.B. haproxy.conf

## Start des Proxys:

```
haproxy -V -f haproxy.conf
```

(in Verbose-Mode mit Konfigurationsfile haproxy.conf):

# HAProxy: Konfiguration

Einfache Beispielkonfiguration für einen TCP-Server mit zwei Instanzen.

```
global
```

**Protokoll:** tcp | http | ...

```
defaults
```

```
    mode tcp
```

**Lastverteilung:**

roundrobin | leastconn | source | ...

```
    balance roundrobin
```

```
    timeout connect 5000ms
```

**Timeouts:**

- Verbindungsaufbau (connect)
- Client/Server Inaktivität (Client / Server)

```
    timeout client 50000ms
```

```
    timeout server 50000ms
```

```
frontend http-in
```

**Frontend:** Angabe des Listener-Interface für die Clients sowie des Ziel-Backends.

```
    bind *:4000
```

```
    default_backend servers
```

```
backend servers
```

**Backend:** Liste von Instanzen der Serverapplikation.

```
    server server1 127.0.0.1:8000 check
```

```
    server server2 127.0.0.1:8001 check
```

# Klassenraumübung: Load-Balancing Hands-on

Übung mittels Online-Programmierumgebung:

- <https://replit.com/@mbaettig/Load-Balancing>
  - Kein HSLU-Dienst (benötigt separates Konto), erstellen Sie einen Fork.

**Ziel:** Machen Sie eigene Versuche mit Load-Balancing.

**Vorgehen:** Starten Sie vier Shells:

- **Shell 1:** haproxy -V -f haproxy.conf
- **Shell 2:** java DaytimeServer srv1 127.0.0.1 8000
- **Shell 3:** java DaytimeServer srv2 127.0.0.1 8001
- **Shell 4:** java DaytimeClient 127.0.0.1 1300

**Hinweise:**

- Stoppen jeweils mit Ctrl-C.
- Zum Neuladen der Konfiguration muss HAProxy neugestartet werden.

# Klassenraumübung: Load-Balancing Hands-on (forts.)

**Machen Sie folgende Experimente und notieren Sie Ihre Beobachtungen:**

- Führen Sie die Zeitabfrage (DaytimeClient) mehrfach aus.
- Mehrere Zeitabfragen mit anderer Lastverteilung: source und/oder leastconn.
- Machen Sie mehrere Zeitabfragen während:
  - Ein oder beide Server gestoppt sind.
  - Einer oder beide Server wieder gestartet sind.

# In-Memory Datagrids

# Idee: Transparent verteilte statt lokale Maps

Beispiel mit einer Collection Map für die In-Memory Speicherung von Daten...

```
import java.util.Map;
import java.util.HashMap;

Map<Integer, String> map = new HashMap<>();
map.put(1, "value");
String result = map.get(1);
```

# Verteilte In-Memory Map

...wenn die In-Memory Speicherung von Daten parallel und verteilt sein soll (mit gemeinsamer Ressource)...

```
import com.hazelcast.core.HazelcastInstance;
import com.hazelcast.core.Hazelcast;
import java.util.Map;

HazelcastInstance client = Hazelcast.newHazelcastInstance();
Map<Integer, String> map = client.getMap("mymap");
map.put(1, "value");
String result = map.get(1);
```

# Was können In-Memory Datagrids?

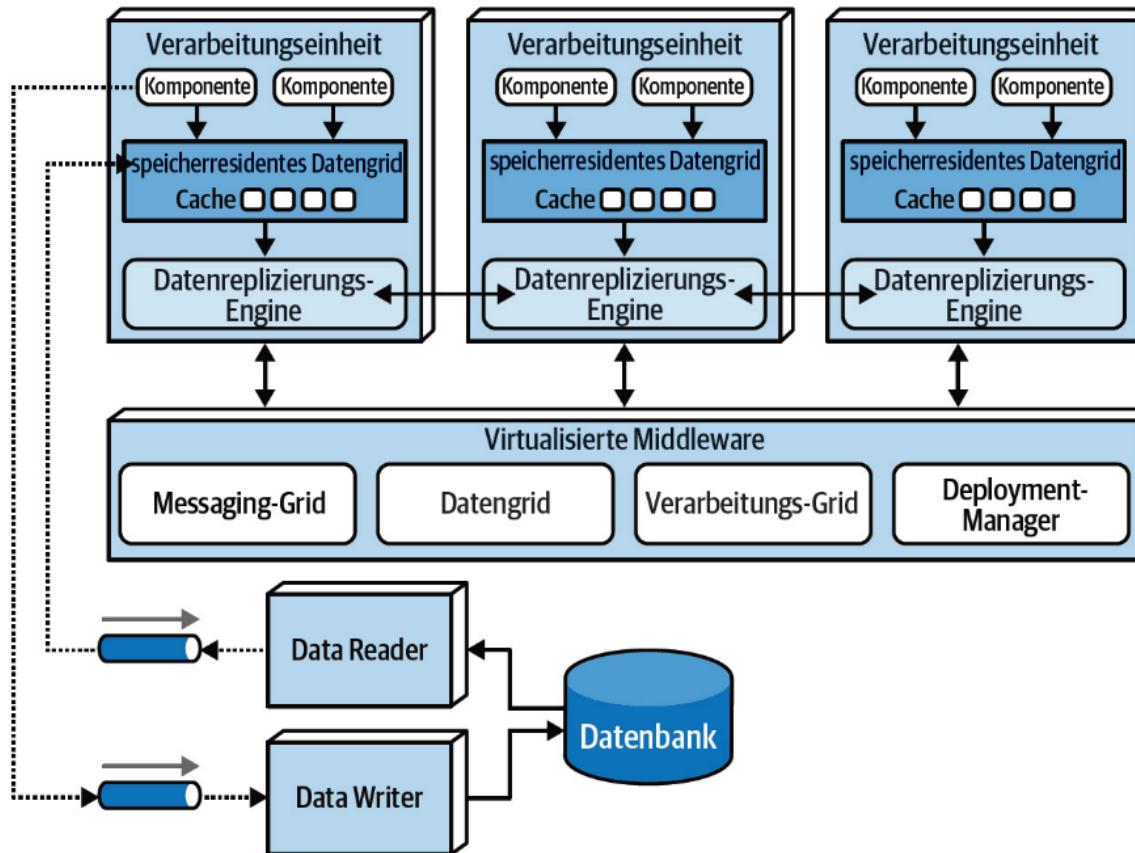
- Applikationen skalieren
- Daten über Cluster verteilen
- Daten partitionieren
- Nachrichten senden und empfangen
- Lasten verteilen
- Parallele Tasks verarbeiten
- ...

## Implementationen von In-Memory Datagrid-Technologien:

- HazelCast: <https://hazelcast.com>
- Apache Ignite <https://ignite.apache.org>
- Oracle Coherence <https://oreil.ly/XOUJL>

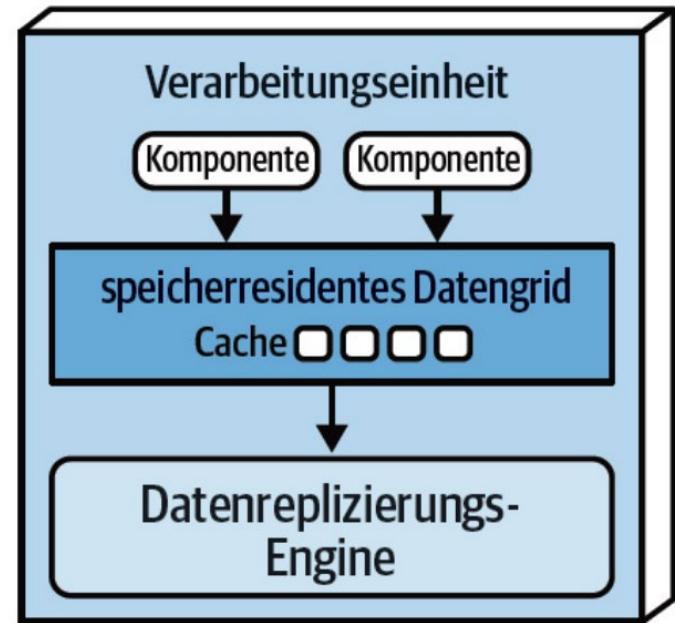
# Architektur auf grosser Flughöhe

- **Verarbeitungseinheiten:** Halten Teile der Daten im Hauptspeicher bereit.
- **Datenbank:** Speichert alle Daten dauerhaft.
- **Middleware:** Kommunikation zwischen Verarbeitungseinheiten.

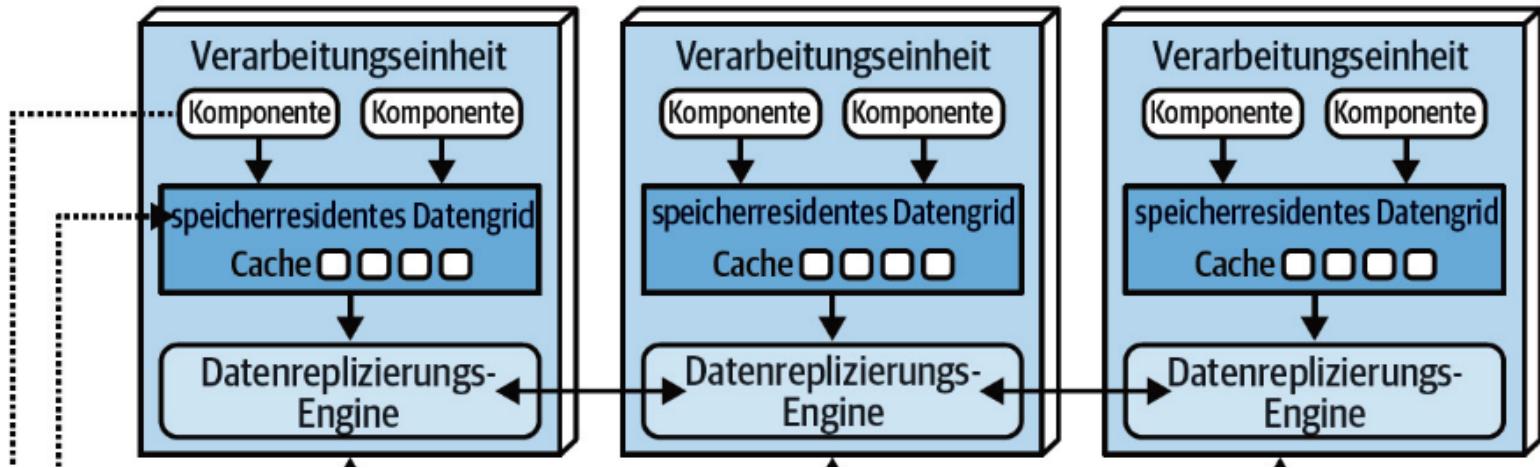


# Verarbeitungseinheiten (Cluster node)

- Embedded-Topologie: Logik und Daten.
- Daten liegen im RAM-Speicher des Cluster Nodes:
  - schneller als Disk optimierte Datenbanken.
  - verbesserte Reaktionszeit bei kritischen Anwendungen.



# Redundanz, Skalierbarkeit und Elastizität



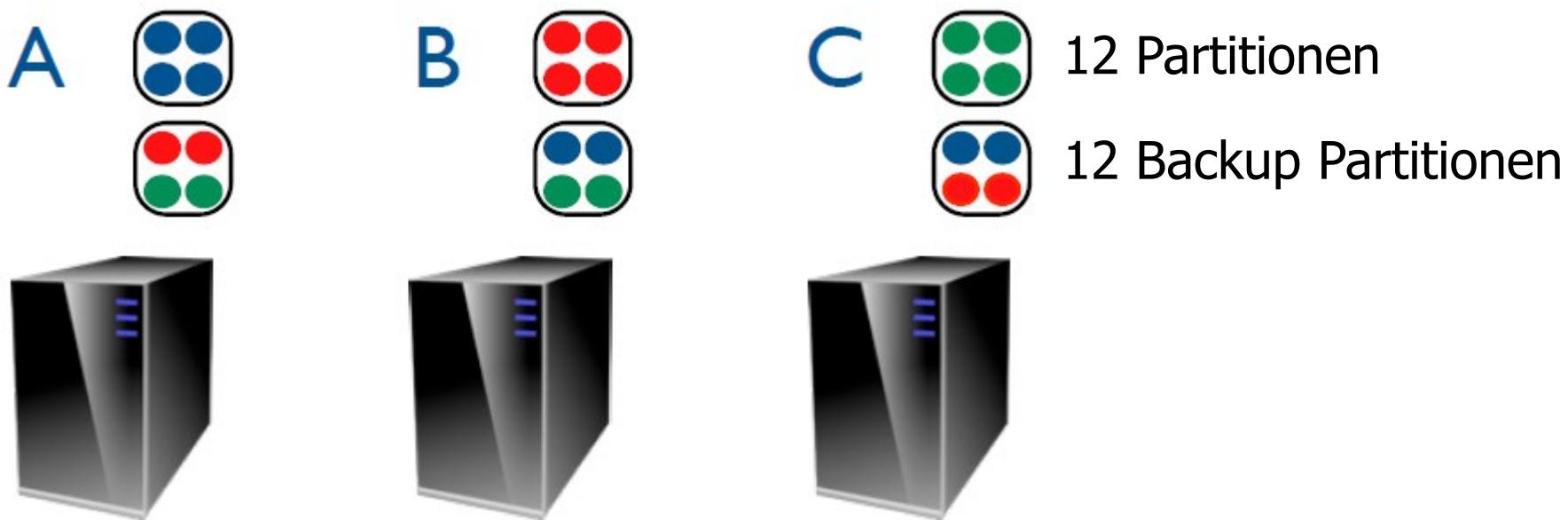
- **Skalierbarkeit**: über verschiedene Cluster-Nodes verteilt.
- **Redundanz**: mehrere Kopien der Daten in verschiedenen Cluster-Nodes
- **Elastizität**: Cluster-Nodes können im Betrieb hinzugefügt oder entfernt werden

# Datenpartitionierung in einem Cluster

- Fixe Anzahl von Partitionen
- Für jede Partition gibt es einen Schlüssel.

```
partitionId = hash(keyData) % PARTITION_COUNT
```

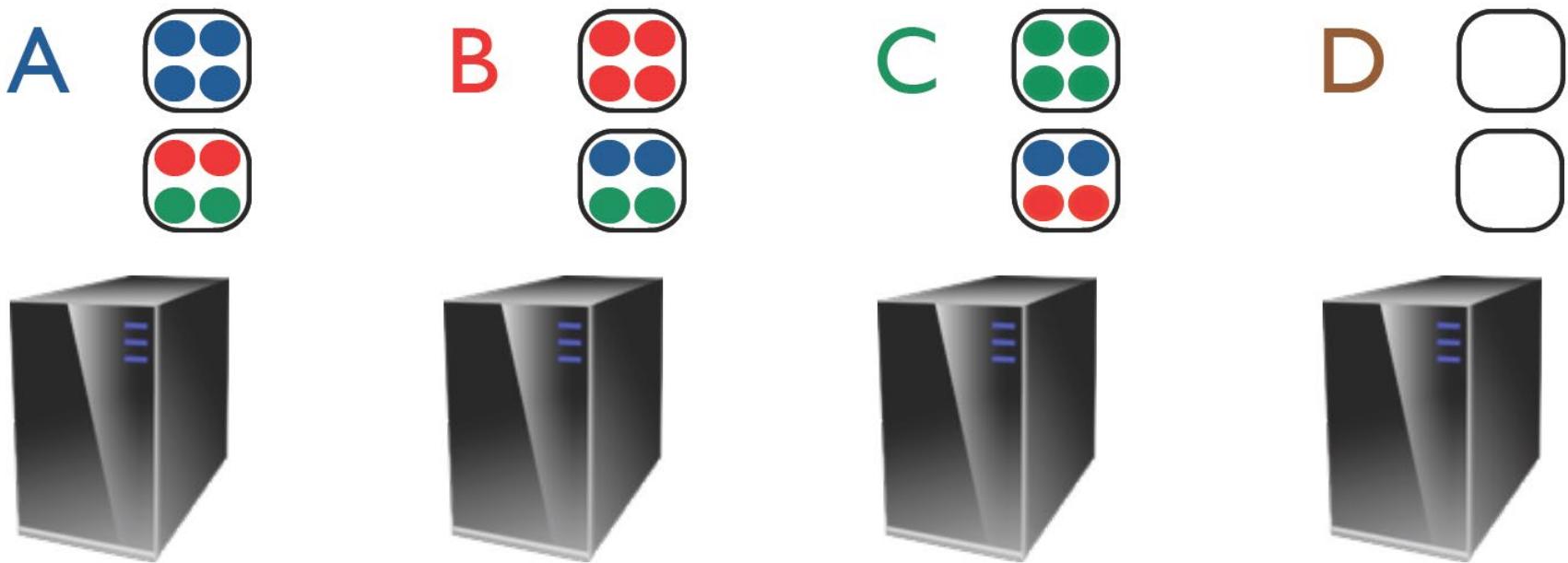
- Alle Partitionen sind möglichst gleichverteilt über den Cluster
  - Backups / Redundanz



# Funktionsweise von In-Memory Datagrids

Neuer Knoten (D) kommt hinzu...

- enthält noch keine Partitionen

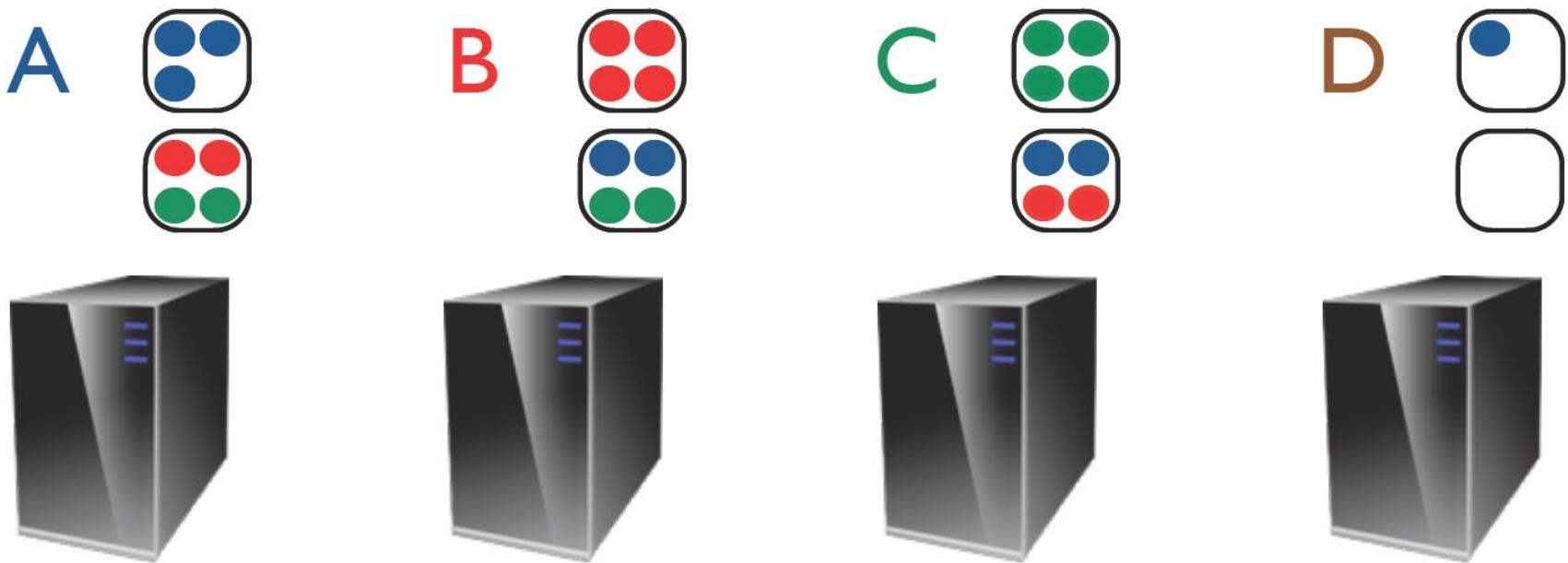


Die folgenden Aktionen müssen in der Praxis nicht in dieser Reihenfolge ablaufen.

# Funktionsweise von In-Memory Datagrids

Migration...

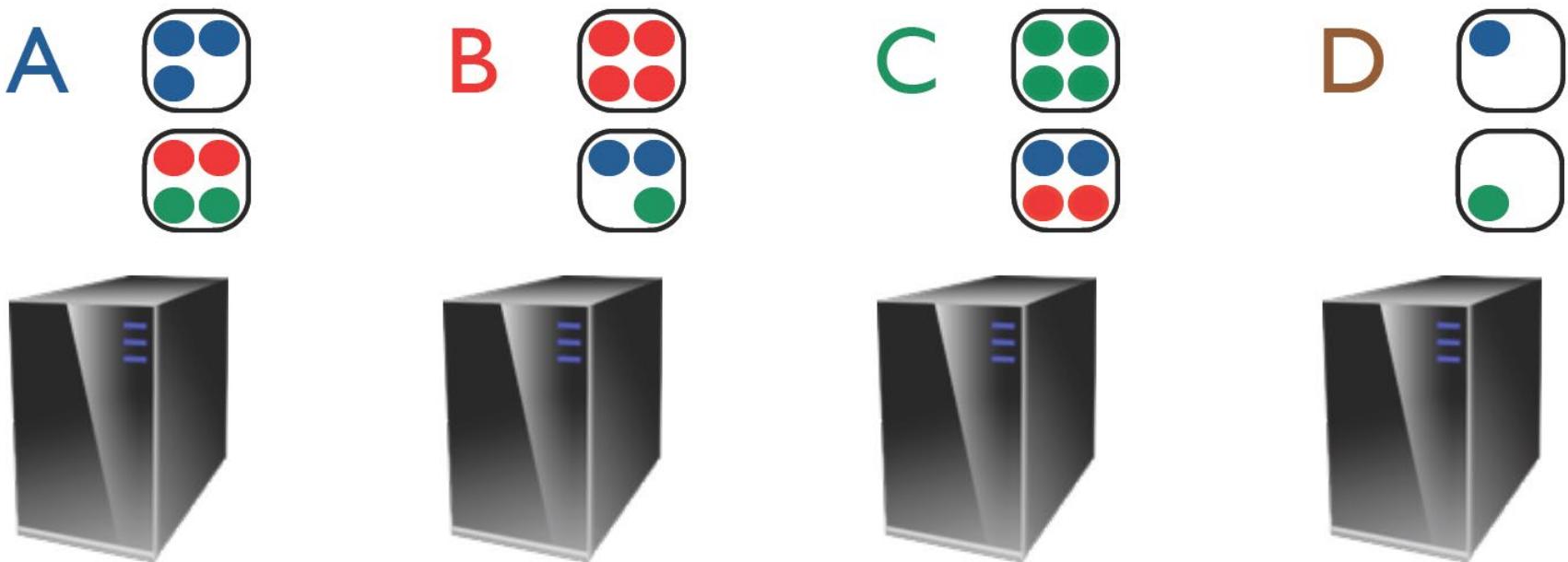
- Knoten D übernimmt Partition von Knoten A



# Funktionsweise von In-Memory Datagrids

Migration...

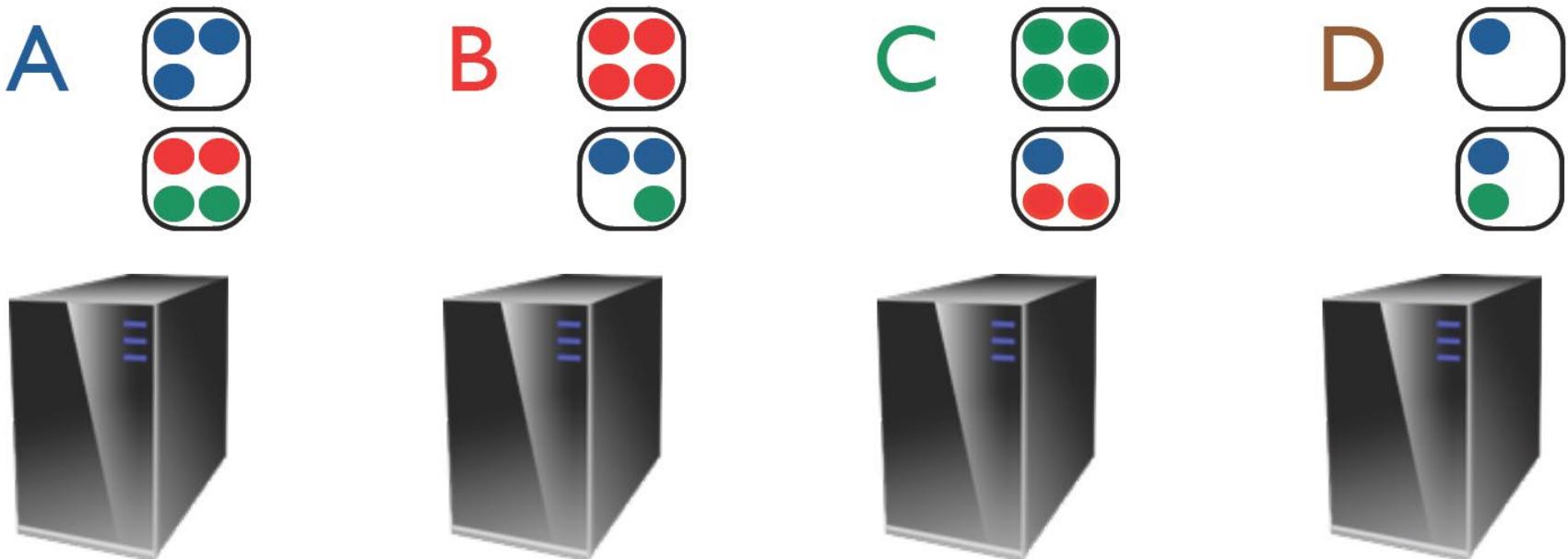
- Knoten D übernimmt Backup Partition C von Knoten B



# Funktionsweise von In-Memory Datagrids

Migration...

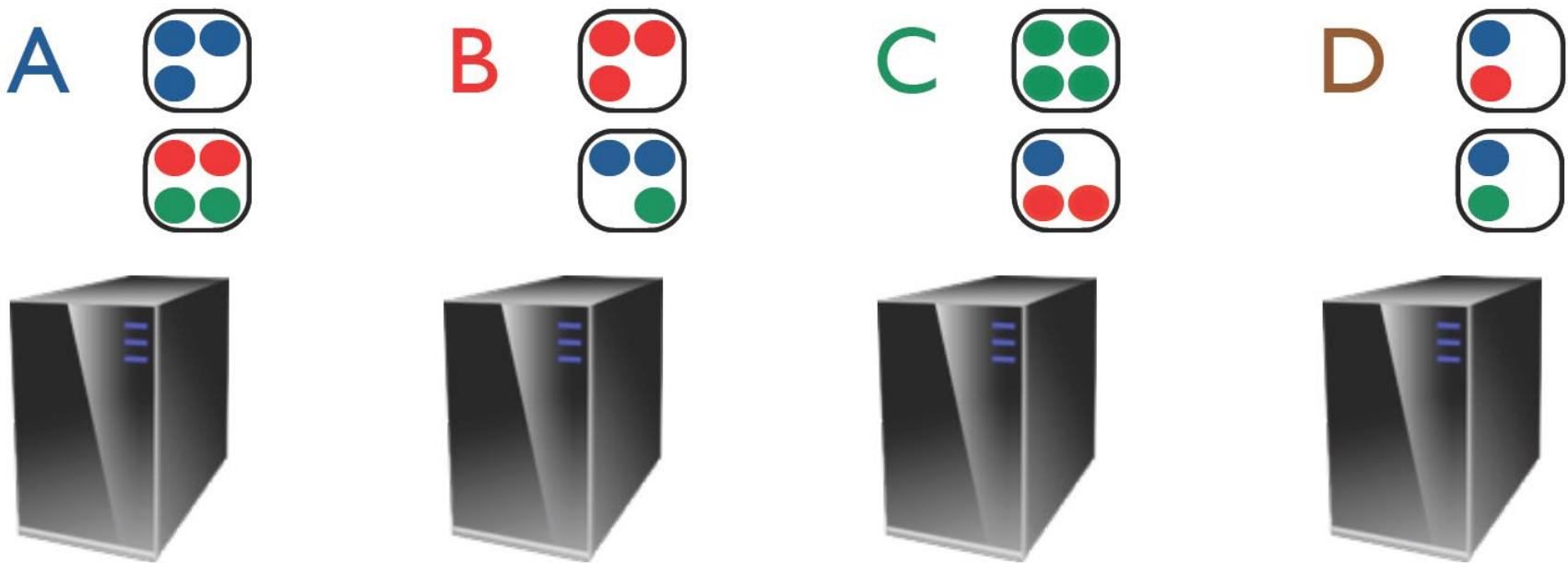
- Knoten D übernimmt Backup Partition A von Knoten C



# Funktionsweise von In-Memory Datagrids

Migration...

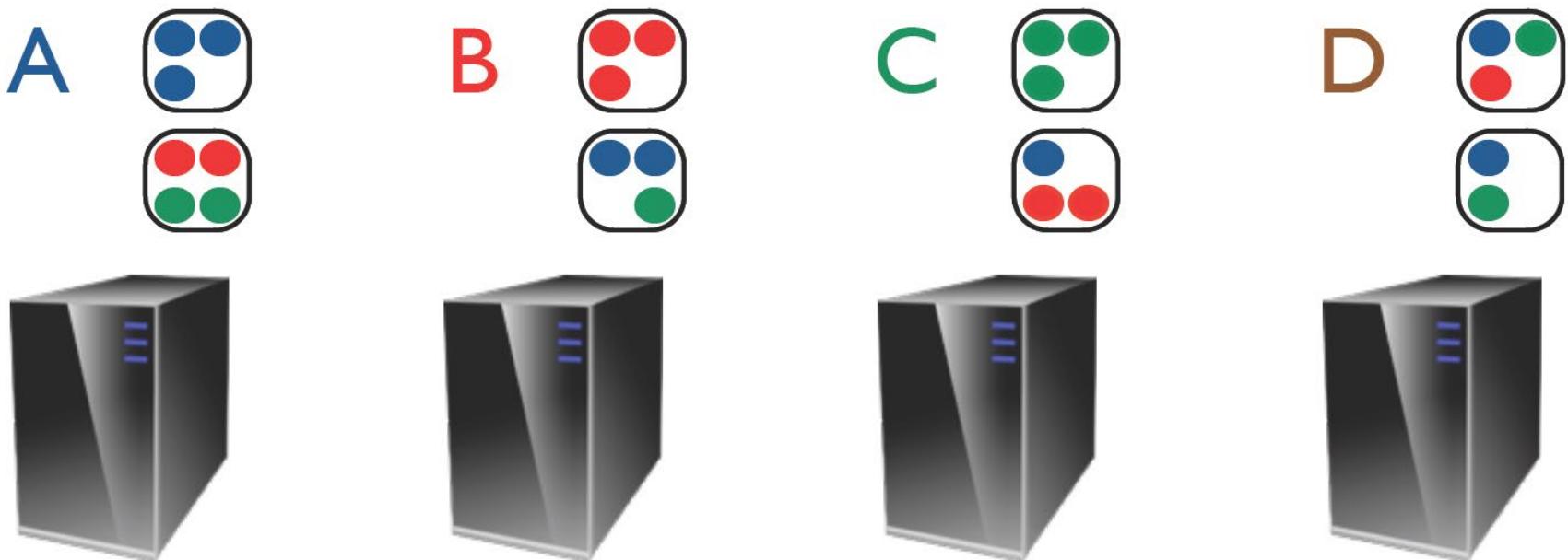
- Knoten D übernimmt Partition von Knoten B



# Wie funktioniert Hazelcast?

Migration...

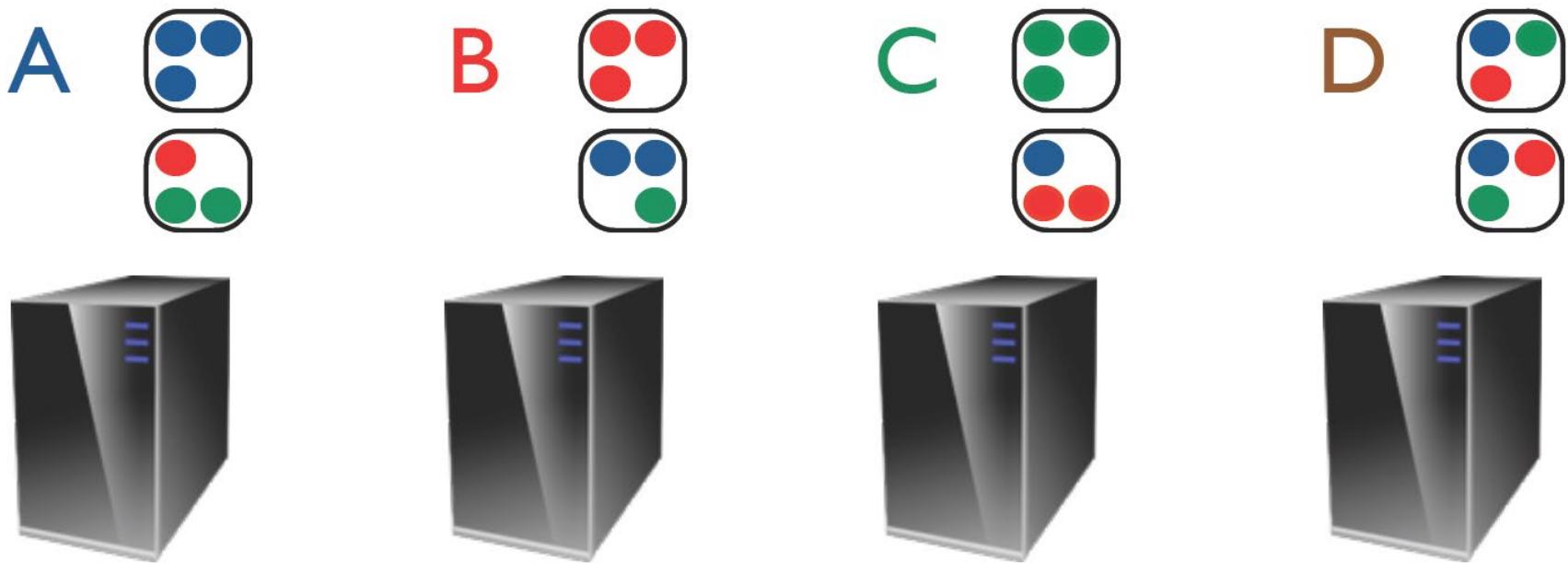
- Knoten D übernimmt Partition von Knoten C



# Wie funktioniert Hazelcast?

Migration...

- Knoten D übernimmt Backup Partition B von Knoten A

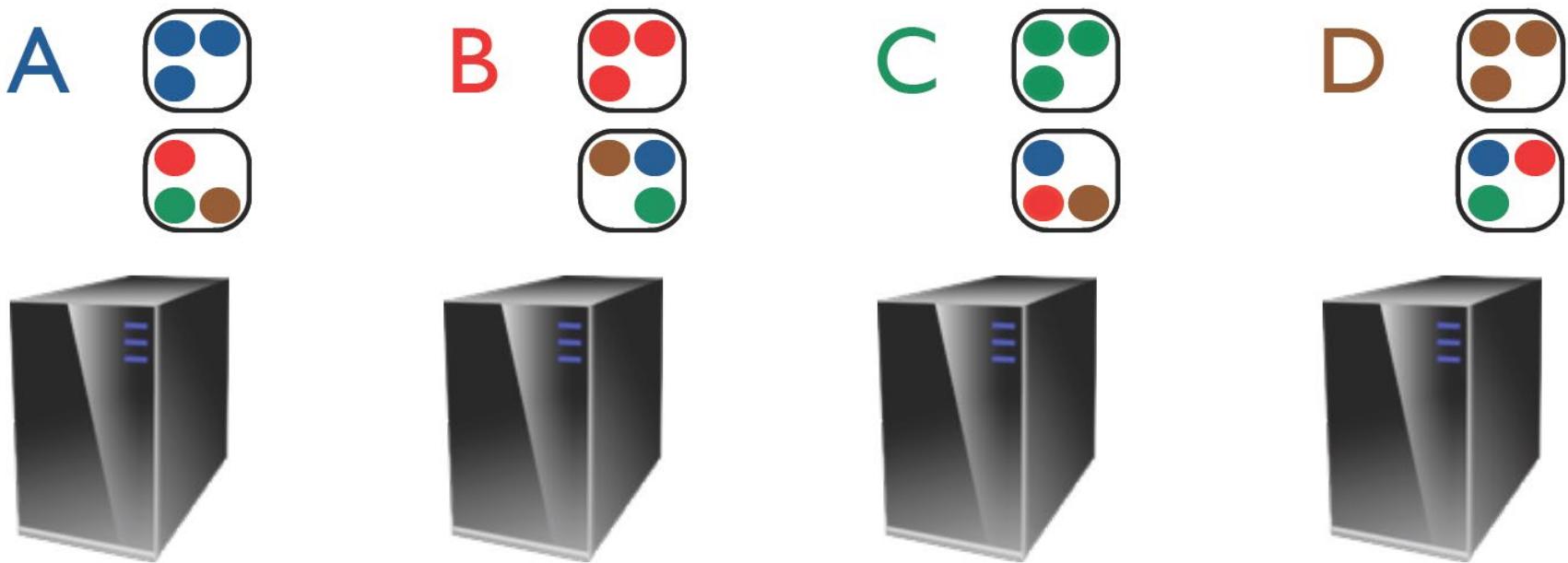


Knoten D besitzt nun Backup Partitionen der andern Knoten.

# Wie funktioniert Hazelcast?

Migration komplett – 12 Partitionen

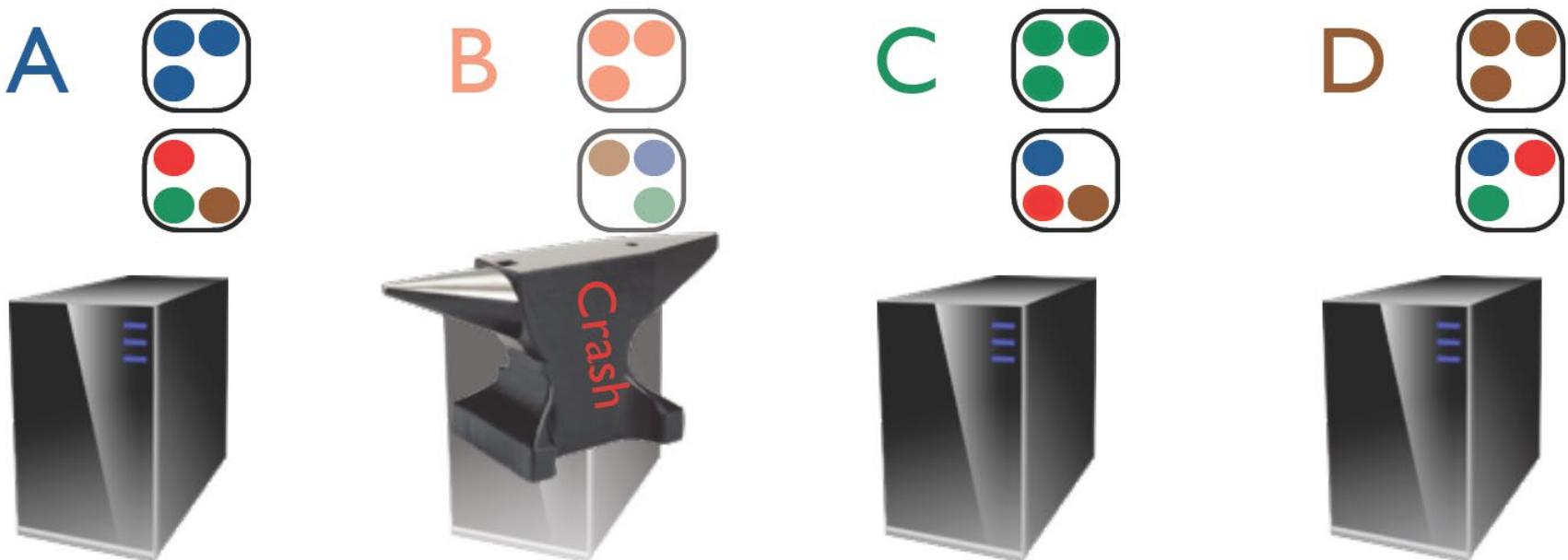
- Knoten D besitzt nun eigene Partitionen (ehemals Partitionen der andern Knoten – Besitzwechsel)



Alle andern Knoten besitzen ein Backup von Knoten D.

# Wie funktioniert Hazelcast?

Knoten B stürzt ab...



# Wie funktioniert Hazelcast?

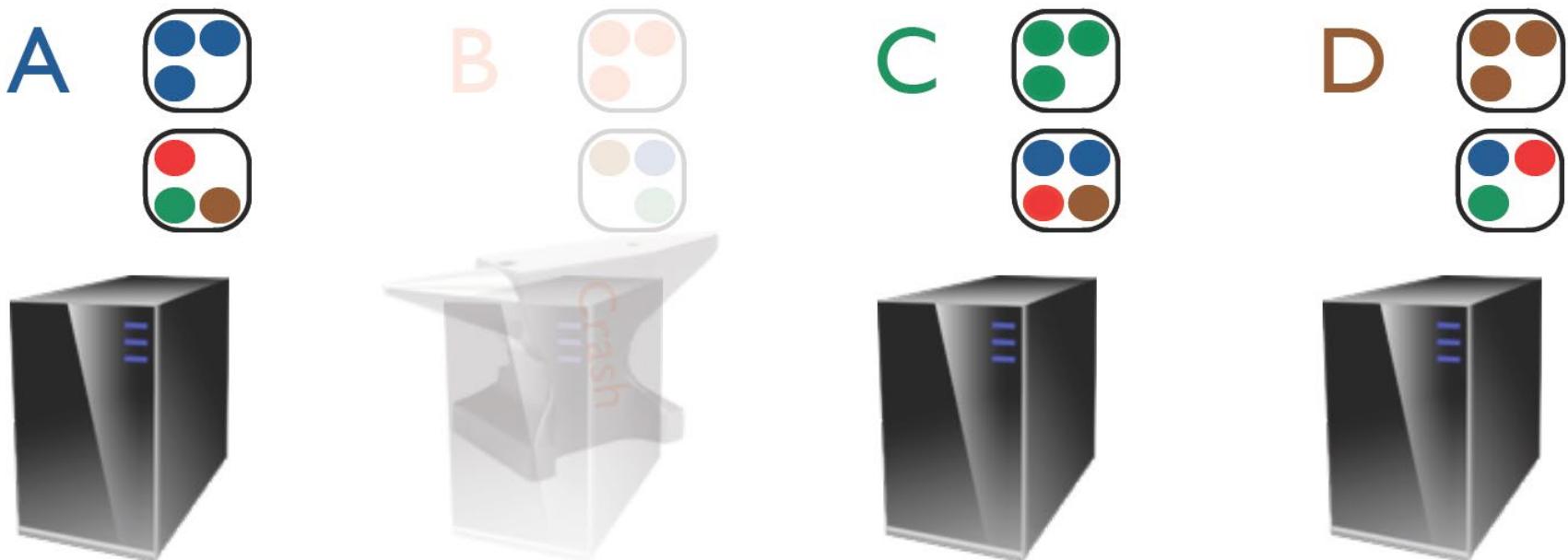
Wiederherstellung mit Hilfe der Backups...



# Wie funktioniert Hazelcast?

Backup anlegen...

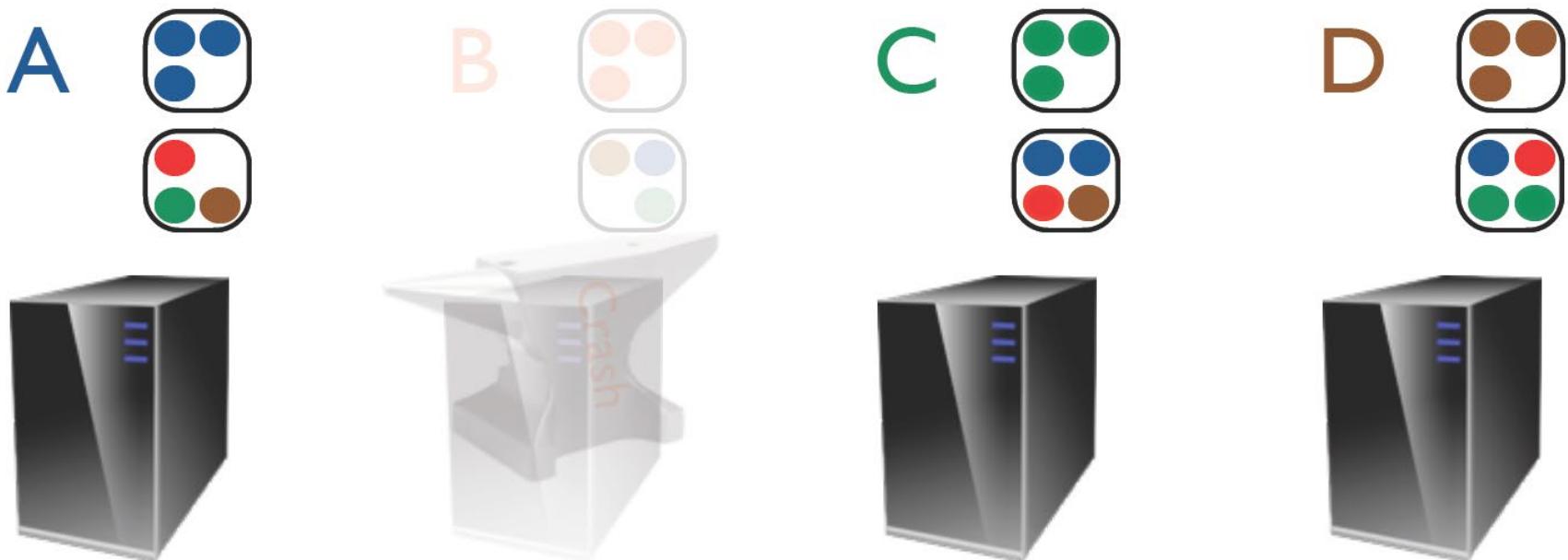
- Knoten C kopiert Partition von A (die auf Knoten B war)



# Wie funktioniert Hazelcast?

Backup anlegen...

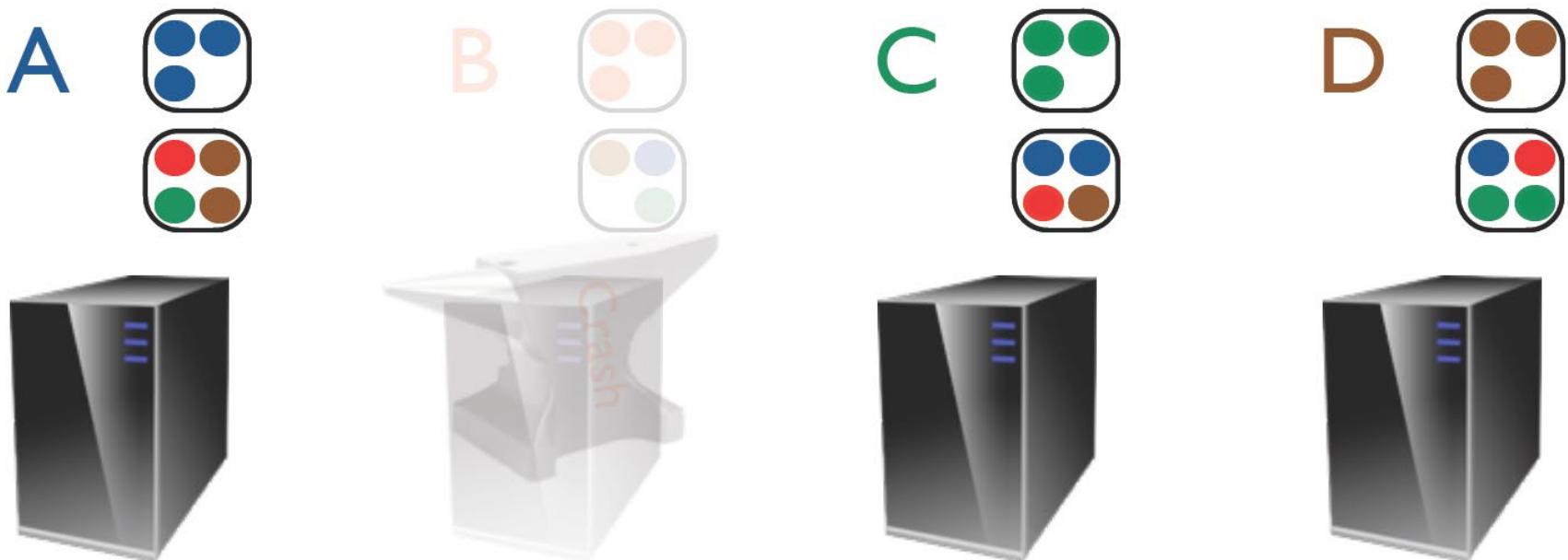
- Knoten D kopiert Partition von C (die auf Knoten B war)



# Wie funktioniert Hazelcast?

Backup anlegen...

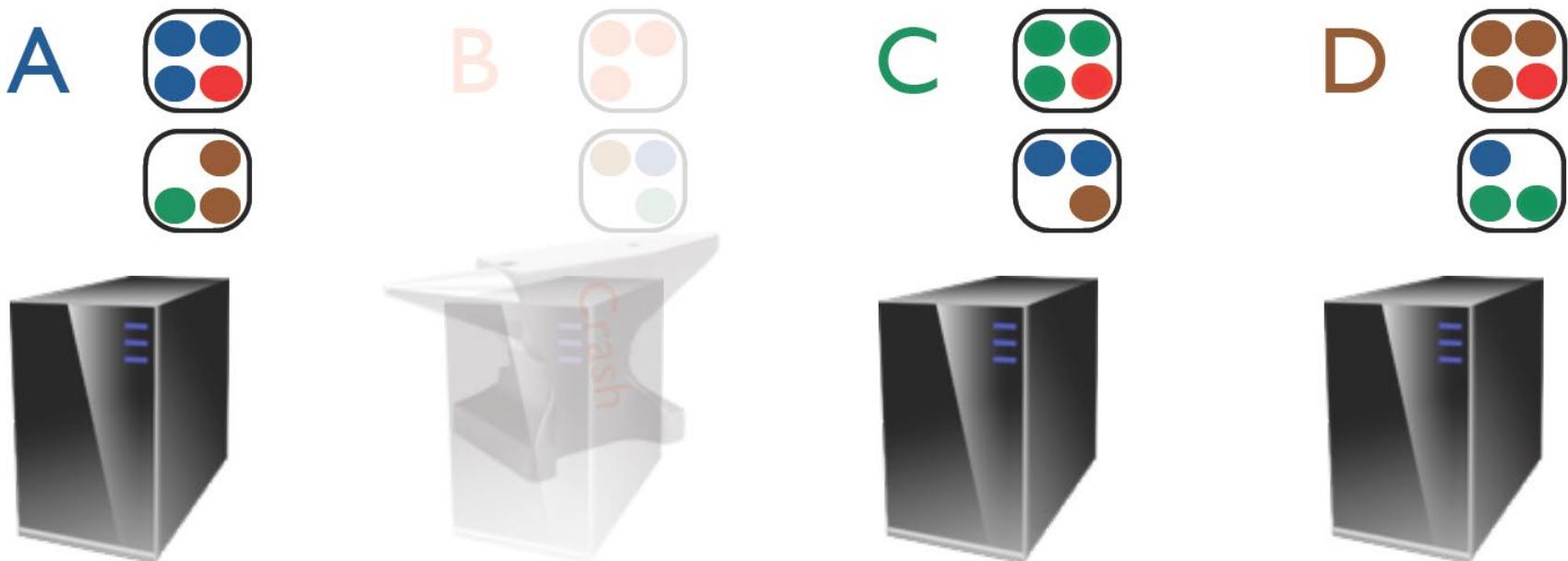
- Knoten A kopiert Partition von D (die auf Knoten B war)



# Wie funktioniert Hazelcast?

Daten wiederherstellen...

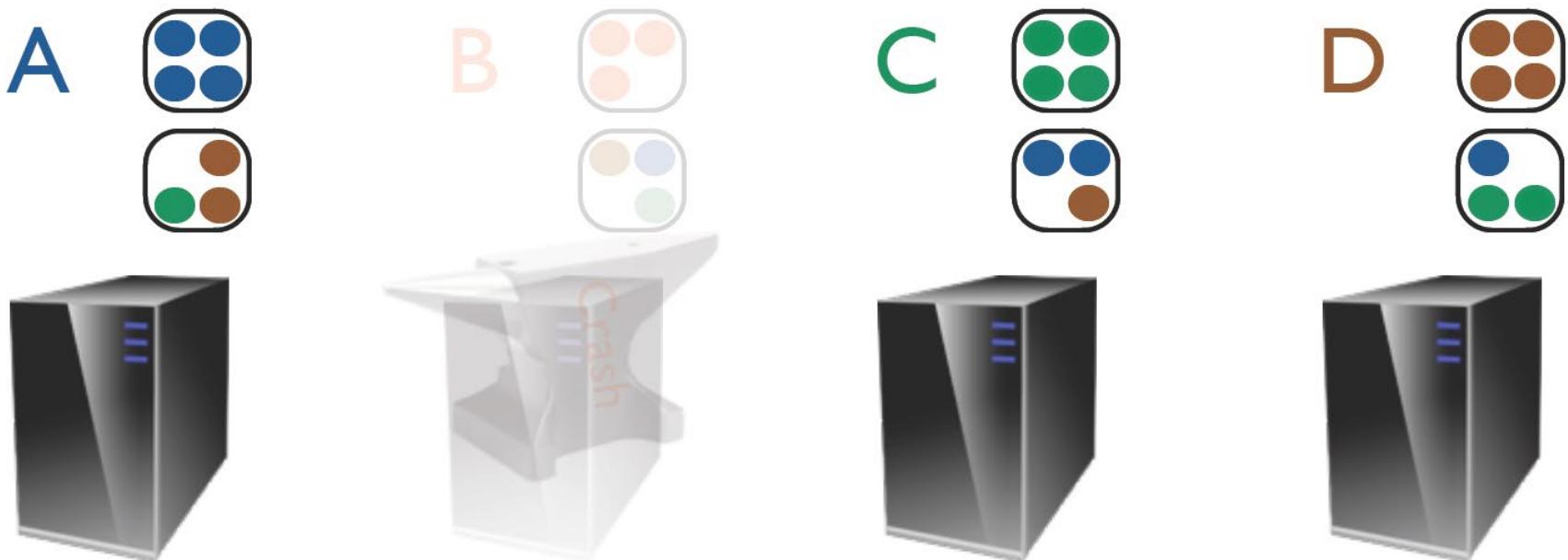
- Backup Partitionen vom ehemaligen Knoten B werden in die eigenen Partitionen übernommen



# Wie funktioniert Hazelcast?

Daten wiederhergestellt – 12 Partitionen

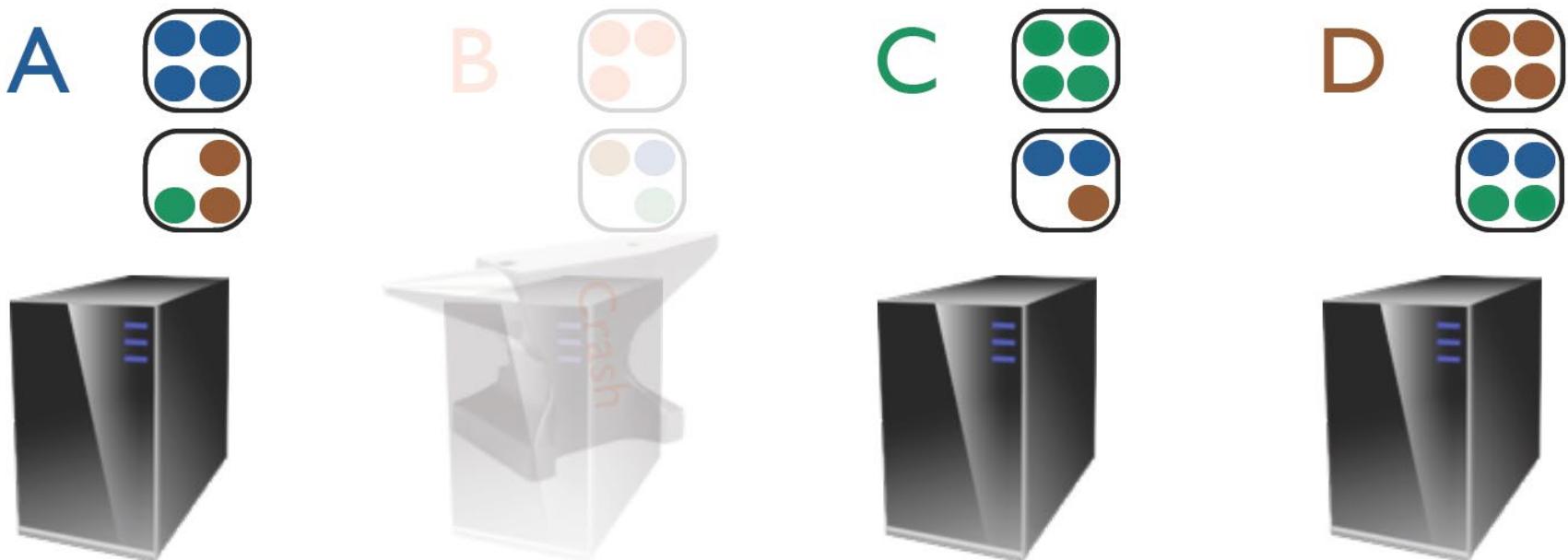
- Die Knoten besitzen wieder alle Partitionen inkl. die vom ehemaligen Knoten B (Besitzwechsel)



# Wie funktioniert Hazelcast?

Backup anlegen von den wiederhergestellten Daten...

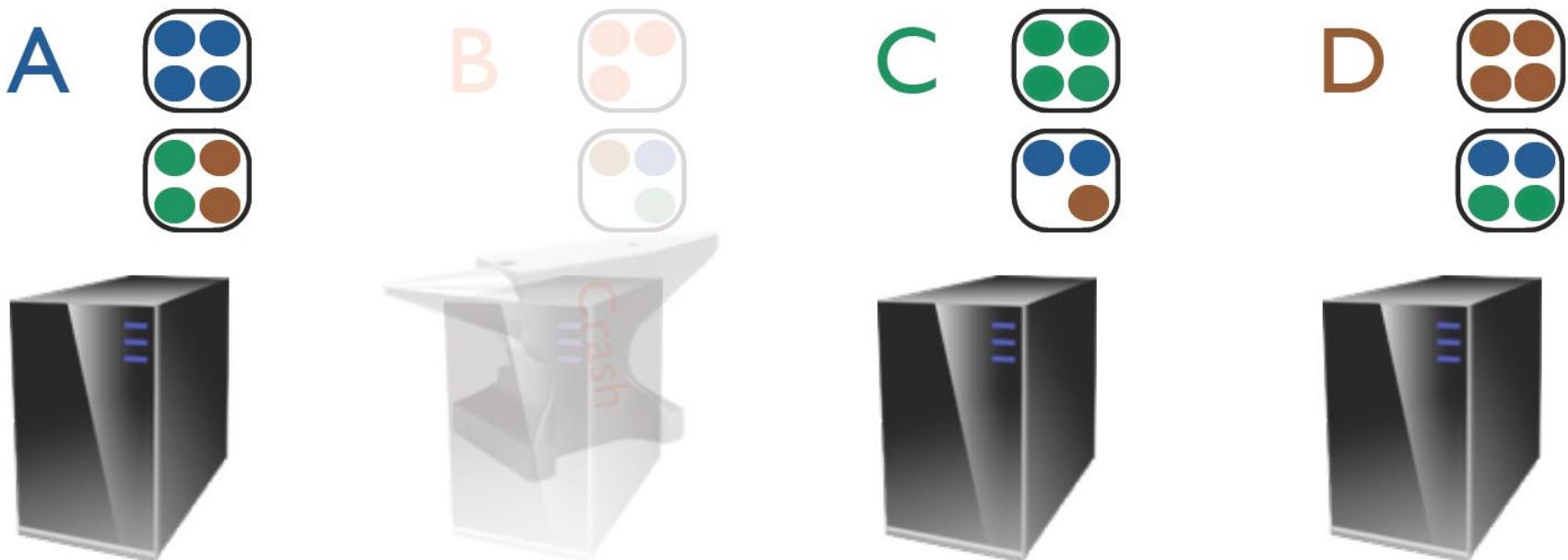
- Knoten D kopiert wiederhergestellte Partition von A



# Wie funktioniert Hazelcast?

Backup anlegen von den wiederhergestellten Daten...

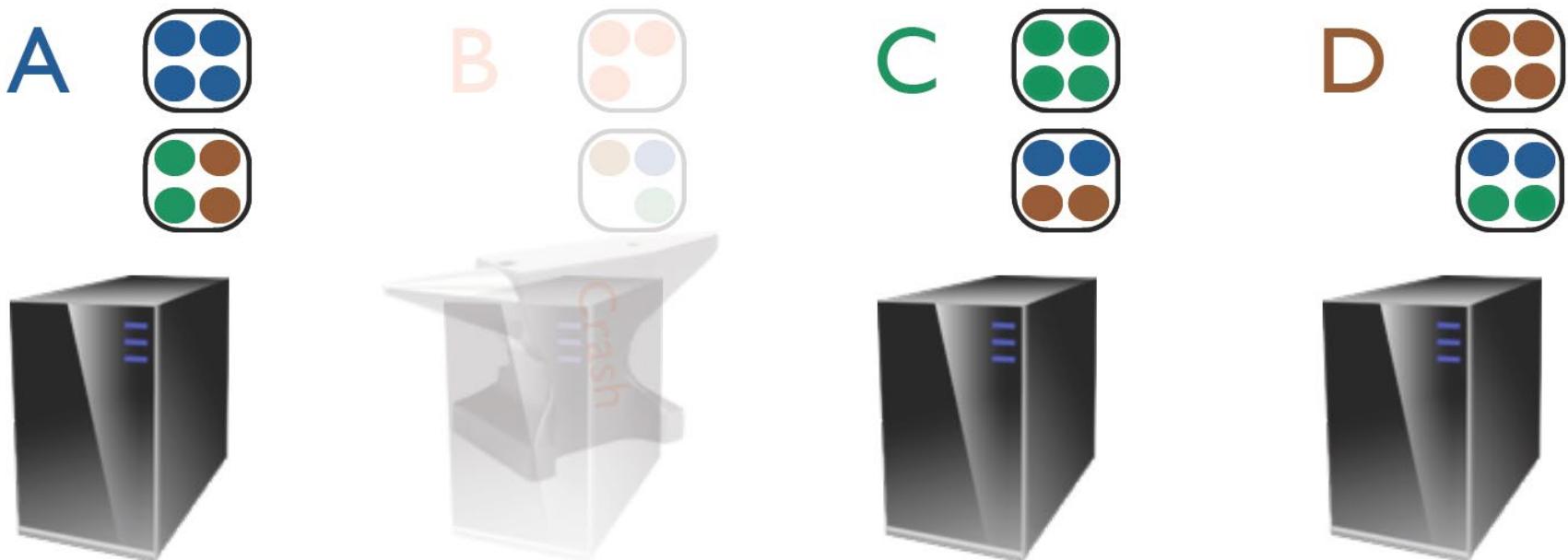
- Knoten A kopiert wiederhergestellte Partition von C



# Wie funktioniert Hazelcast?

Backup anlegen von den wiederhergestellten Daten...

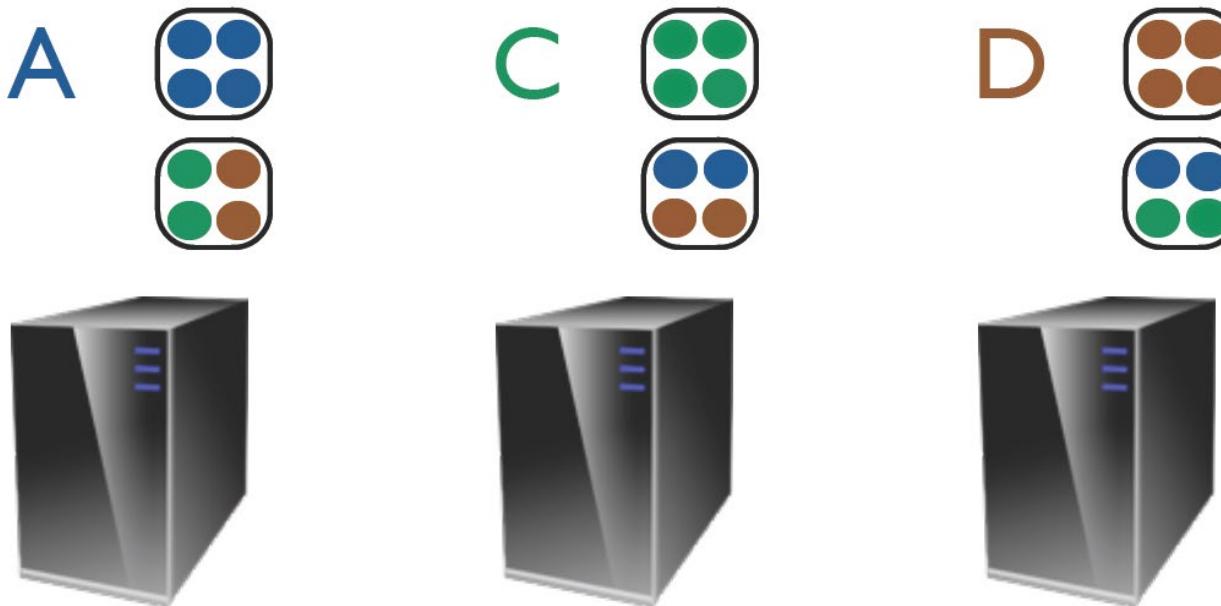
- Knoten C kopiert wiederhergestellte Partition von D



# Wie funktioniert Hazelcast?

Alles wieder vollständig und sicher!

- 12 Partitionen
- 12 Backup Partitionen verteilt auf alle Knoten



## Beispiel: Hazelcast



- In-Memory Data Grid in Java
- Open Source (Apache License 2.0)
  - <https://hazelcast.com/open-source-projects/downloads/>
- Kommerzielle Versionen & Subscription Plans: Enterprise & HD

# Management Center

The screenshot shows the Hazelcast Management Center interface. At the top, there is a header bar with the Hazelcast logo, a dropdown for 'dev' cluster, 'Cluster Connections' with a plus sign, 'Docs' with a link icon, 'Console' with a right arrow, 'Settings' with a gear icon, and 'Dev Mode' with a double arrow icon.

The left sidebar contains navigation sections:

- Cluster**:
  - Dashboard
  - Clients (4)
  - Client Filtering
  - Members (1)
  - Administration
  - WAN Replication (0)
  - Scripting
  - Healthcheck
- Storage**
- Streaming**
- Compute**
- Messaging**:
  - Queues (1)
  - Topics (0)
  - Reliable Topics (0)

The main content area includes the following components:

- Cluster Status**: Shows State (Active), Migrations (0 waiting migrations), CP Subsystem (CP subsystem is disabled), and Filtering (Disabled / Allow-list).
- Heap Memory Distribution**: A chart showing memory usage: Other (34.6%) in purple and Free (65.4%) in blue.
- Partition Distribution**: A chart showing the distribution of partitions across nodes, with one node labeled 192.168.56.1:5701.
- Used Heap, Used Native...**: A line graph showing heap memory usage over time, starting at 28.61 MB and rising to 95.37 MB.
- CPU Load (Outliers)**: A line graph showing CPU load outliers over time, with the message "No data to plot".

<https://hazelcast.com/products/management-center/>  
<https://hazelcast.org/imdg/download/>

# Programmierung: Aufbau und Abbau einer Connection

- Hazelcast ist Thread safe:

```
Map<String, Employee> map = client.getMap("employees");
List<Customer> list = client.getList("customers");
```

- Viele Instanzen auf der gleichen JVM:

```
HazelcastInstance h1 = Hazelcast.newHazelcastInstance();
HazelcastInstance h2 = Hazelcast.newHazelcastInstance();
// ...
```

- Alle zu verteilende Objekte müssen interface DataSerializable implementieren:

```
public class Employee
    implements com.hazelcast.nio.serialization.DataSerializable
```

# Programmierung: Cluster Interface

zeigt Informationen über die Mitglieder des Clusters:

```
MembershipListener listener = new MembershipAdapter();
HazelcastInstance client = Hazelcast.newHazelcastInstance();

Cluster cluster = client.getCluster();
cluster.addMembershipListener(listener);
Member localMember = cluster.getLocalMember();
LOG.info(localMember.getAddress());

Set<Member> members = cluster.getMembers();
Iterator<Member> it = members.iterator();
while (it.hasNext()) {
    Member member = it.next();
    LOG.info(member.getUuid() + " from " + member.getAddress());
}
```

# Programmierung: Distributed Lock

- Falls ein systemweiter Zugriff auf gemeinsame Ressource erfolgen muss, wird dieser mit Locks geschützt:

```
String mylockobject = "MyLock";
FencedLock mylock = client.getCPSubsystem().getLock(mylockobject);
mylock.lock();
try {
    // do something
} finally {
    mylock.unlock();
}
```

- Auch Hazelcast Collections bieten Locks an:

```
IMap<String, Customer> map = client.getMap("customers");
map.lock("1");
try {
    // do something
} finally {
    map.unlock("1");
}
```

# Programmierung: Distributed Map

Collection Map (hier nebenläufig) mit den Möglichkeiten der verteilten Speicherung von Daten:

```
HazelcastInstance client = Hazelcast.newHazelcastInstance();

ConcurrentMap<String, Customer> map = client.getMap("customers");
Customer customer = new Customer("John Doe", 21, false);
map.put("4711", customer);
customer = map.get("4711");
LOG.info(customer);

//ConcurrentMap methods
map.putIfAbsent("4712", new Customer("Chuck Norris", 80));
map.replace("4711", new Customer("Bruce Lee"));
customer = map.get("4711");
LOG.info(customer);
customer = map.get("4712");
LOG.info(customer);
```

# Programmierung: Distributed Queue

Queues können verteilt gespeichert werden und z.B. für verteiltes Ausführen von Tasks genutzt werden.

```
HazelcastInstance client = Hazelcast.newHazelcastInstance();
//Queue operations
BlockingQueue<Task> queue = client.getQueue("tasks");
Boolean done = queue.offer(new Task());
Task task = queue.poll();
LOG.info(task);

//Timed blocking operations
done = queue.offer(new Task(), 500, TimeUnit.MILLISECONDS);
task = queue.poll(5, TimeUnit.SECONDS);
LOG.info(task);

//Indefinitely blocking operations
queue.put(new Task());
task = queue.take();
LOG.info(task);
```

# Programmierung: Distributed Topic

Verteilungsmechanismus für das Veröffentlichen von Nachrichten, an mehrere, registrierte Abonnenten.

```
public class DistributedTopic implements MessageListener<MyMessage>{
    @Override
    public void onMessage(Message<MyMessage> msg) {
        MyMessage message = msg.getMessageObject();
        LOG.info("Got msg: " + message.getText() +
                 " from " + message.getName());
    }
}

HazelcastInstance client = Hazelcast.newHazelcastInstance();
DistributedTopic distributedTopic = new DistributedTopic();

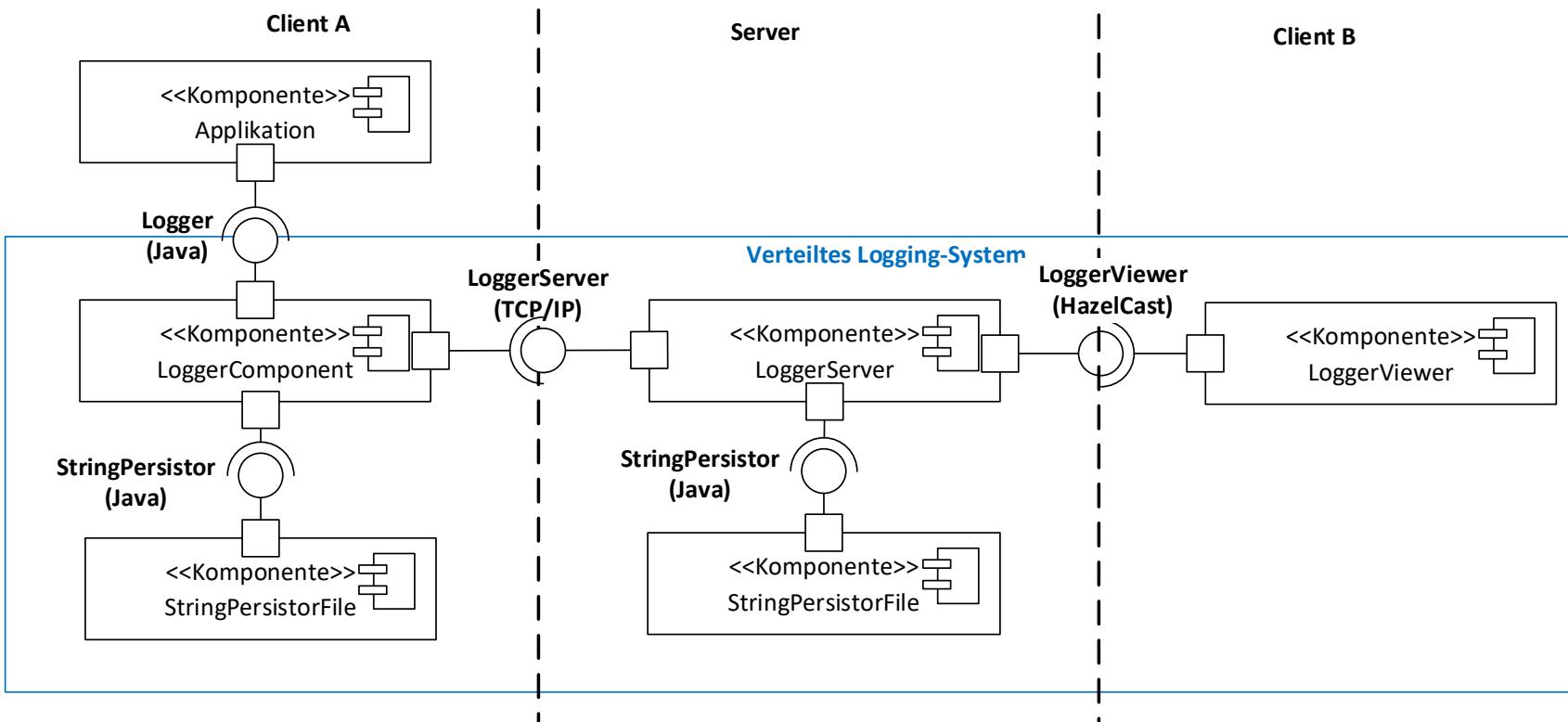
ITopic<MyMessage> topic = client.getTopic("default");
topic.addMessageListener(distributedTopic);
topic.publish(new MyMessage("my message object", "hello"));
```

# Klassenraumübung: LoggerViewer mittels HazelCast

**Ziel:** Anzeige einer möglichst aktuellsten Ansicht der Log-Messages in einem oder mehreren verteilten Clients. Anbindung soll mittels Hazelcast erfolgen.

**Fragestellung:** Welche Datenstrukturen (Map/Queue/Topic) von Hazelcast würden Sie dafür einsetzen und wie?

⇒ Diskutieren Sie in Ihrem Team.



# Zusammenfassung

- Topologie: Separate Verteilung von Daten und Anwendung (Client/Server) oder Zusammenlegen von Daten und Anwendung (Embedded).
- Lastverteilung dient der Skalierung einer verteilten Applikation.
- Skalierung erschwert durch temporären oder persistenten Zustand.
- Reverse-Proxy sind eine Middleware, welche typischerweise die Lastverteilung mittels verschiedenen Methoden ermöglicht.
- In-Memory Data-Grid speichern Daten in Partitionen im Hauptspeicher.
  - Beispiel einer Embedded-Topologie.
  - Daten sind gleichmässig über die Nodes des Systems verteilt.
  - Zur Ausfallsicherheit sind die Daten redundant gespeichert.
- In-Memory-Data-Grids bieten oft vielfältige Möglichkeiten zur verteilten Datenspeicherung, welche sich zur Kommunikation zwischen Systemen eignet.

# **Fragen?**

Verteilte Systeme und Komponenten

# **Komponentenmodelle**

## **Vertiefung anhand von Fallbeispielen**

Martin Bättig



# Inhalt

- Übersicht Komponentenmodelle
- Minimales Komponentenmodell in Java
- Das OSGi-Komponentenmodell
- Ein Komponentenmodell für Microservices

# Lernziele

- Sie kennen die Eigenschaften eines Komponentenmodells.
- Sie können das Konzept des Komponentenmodells anhand von Beispielen erklären.
- Sie können identifizieren, ob ein Konzept ein Komponentenmodell darstellt.

# **Übersicht Komponentenmodelle**

# Nutzen eines Komponentenmodells

"Eine Software-Komponente ist ein Software-Element, das zu einem bestimmten Komponentenmodell passt und entsprechend einem Composition-Standard ohne Änderungen mit anderen Komponenten verknüpft und ausgeführt werden kann." *Council, Heineman: Component-Based Software Engineering, Addison-Wesley, 2001*

## ▪ Komponentenmodell:

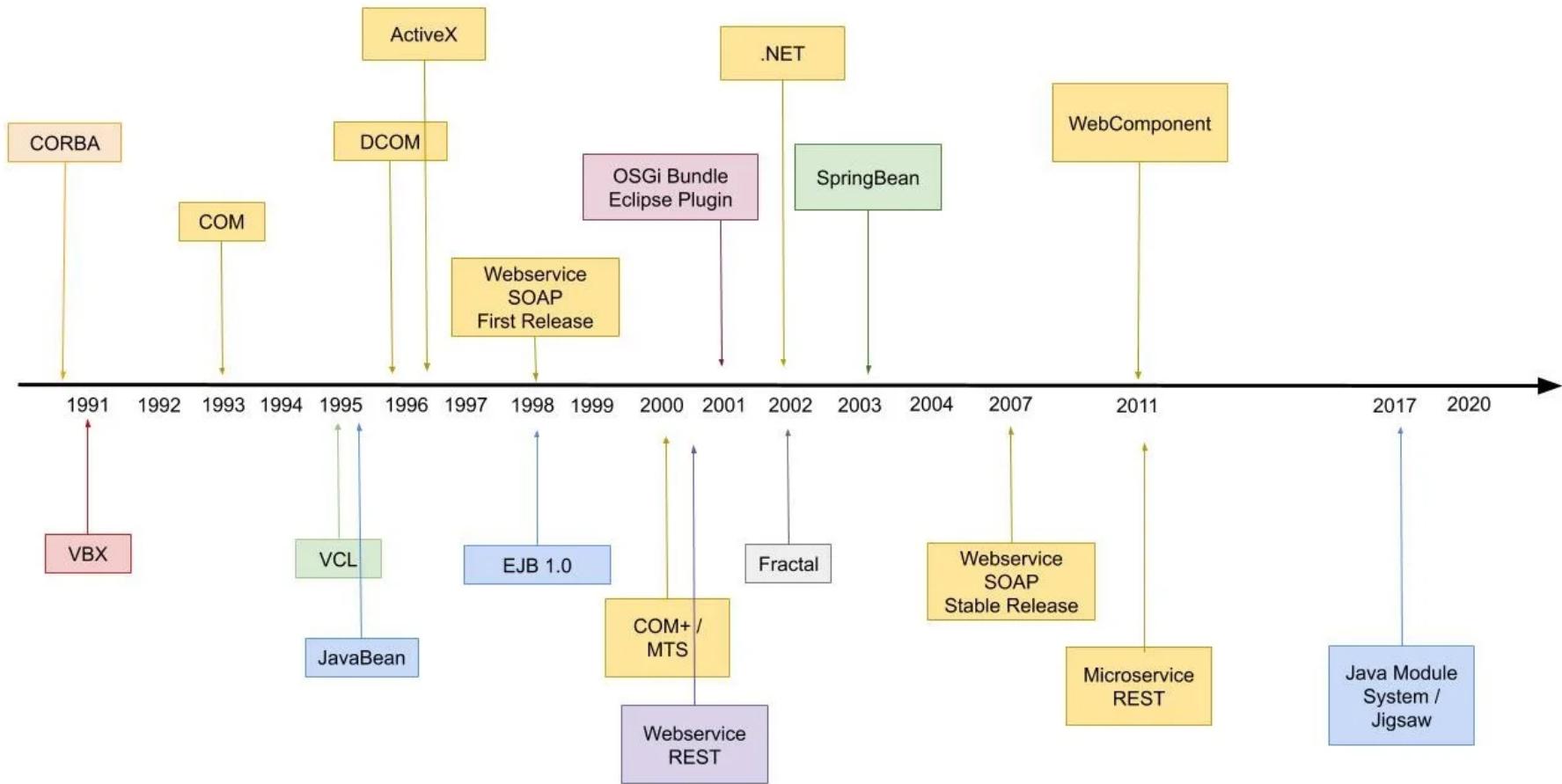
- Legt die grundlegenden Eigenschaften einer Komponente fest.
- Rahmen für die Komponentenentwicklung oft in Zusammenhang mit einer Laufzeitumgebung.
- Basis für die Interaktion: Nur kompatible Komponenten können miteinander interagieren.
- Komponentenmodelle sind unterschiedlich spezifisch.

# Grundlegende Eigenschaften eines Komponentenmodells

Eigenschaft	Inhalt
<b>Schnittstellendefinition</b>	Wie wird die Schnittstelle einer Komponente beschrieben?
<b>Kommunikation und Verteilung</b>	Wie kommuniziert die Komponente mit anderen Komponenten? Ist eine Verteilung auf mehrere Prozesse bzw. physische System möglich?
<b>Komposition und Auffindbarkeit</b>	Können zwei Komponenten miteinander verbunden werden? Wie wird dies gemacht? (Austauschbarkeit, Auffindbarkeit).
<b>Deployment</b>	Wie werden Komponenten ausgeliefert? Wie werden Komponenten gebündelt?

(Obige Liste zeigt minimale Anforderungen; teilweise spezifizieren Komponentenmodelle weitere Eigenschaften wie Sicherheit, Logging, Monitoring, etc.).

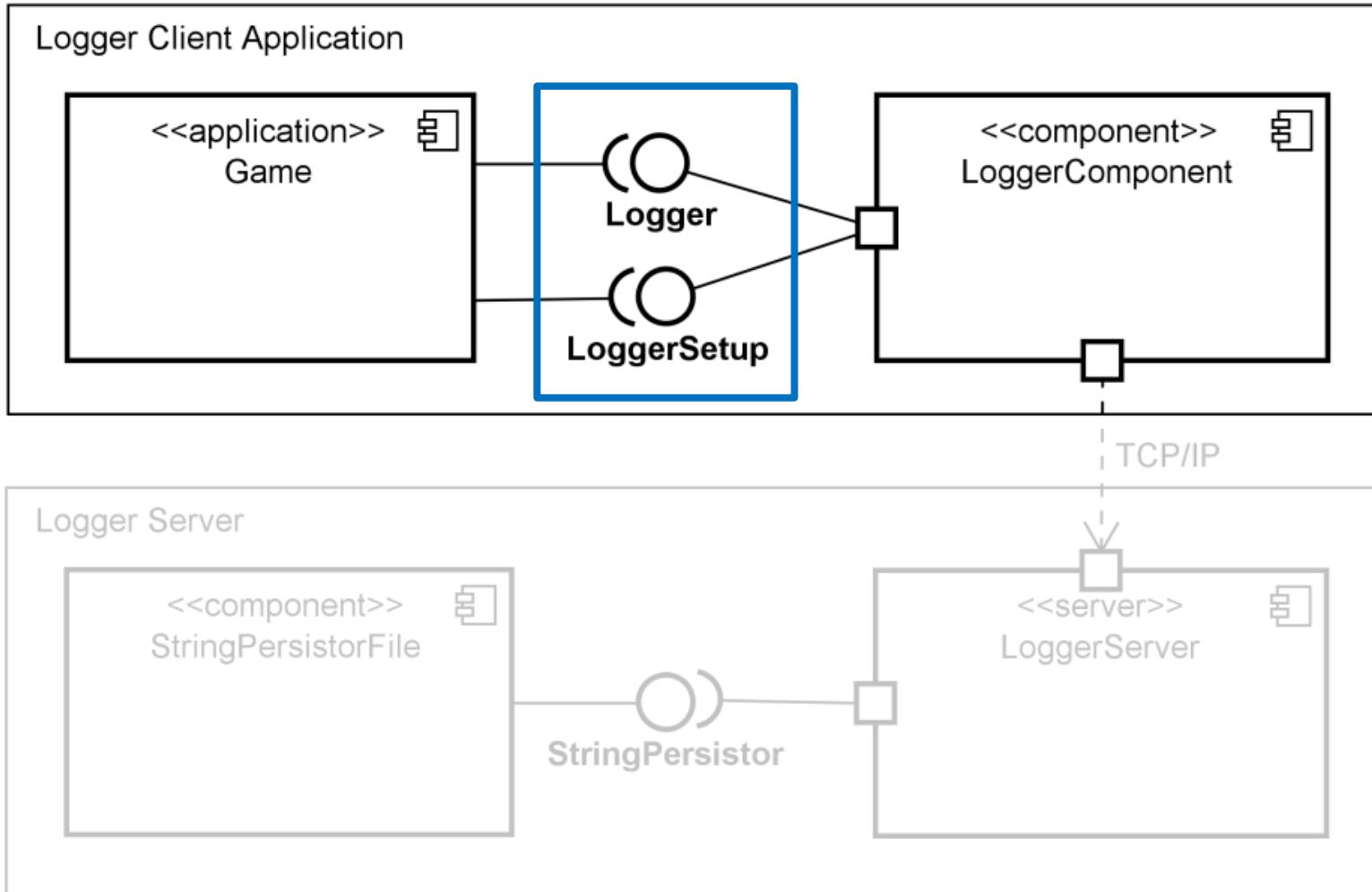
# Komponentenmodelle: Eine Zeitreise...



**Quelle:** <https://www.heise.de/hintergrund/Komponentenbasierte-Softwaretechnik-Was-ist-heute-noch-geblieben-5041855.html>

# **Minimales Komponentenmodell mit Java**

# Komponentenübersicht des Loggersystems

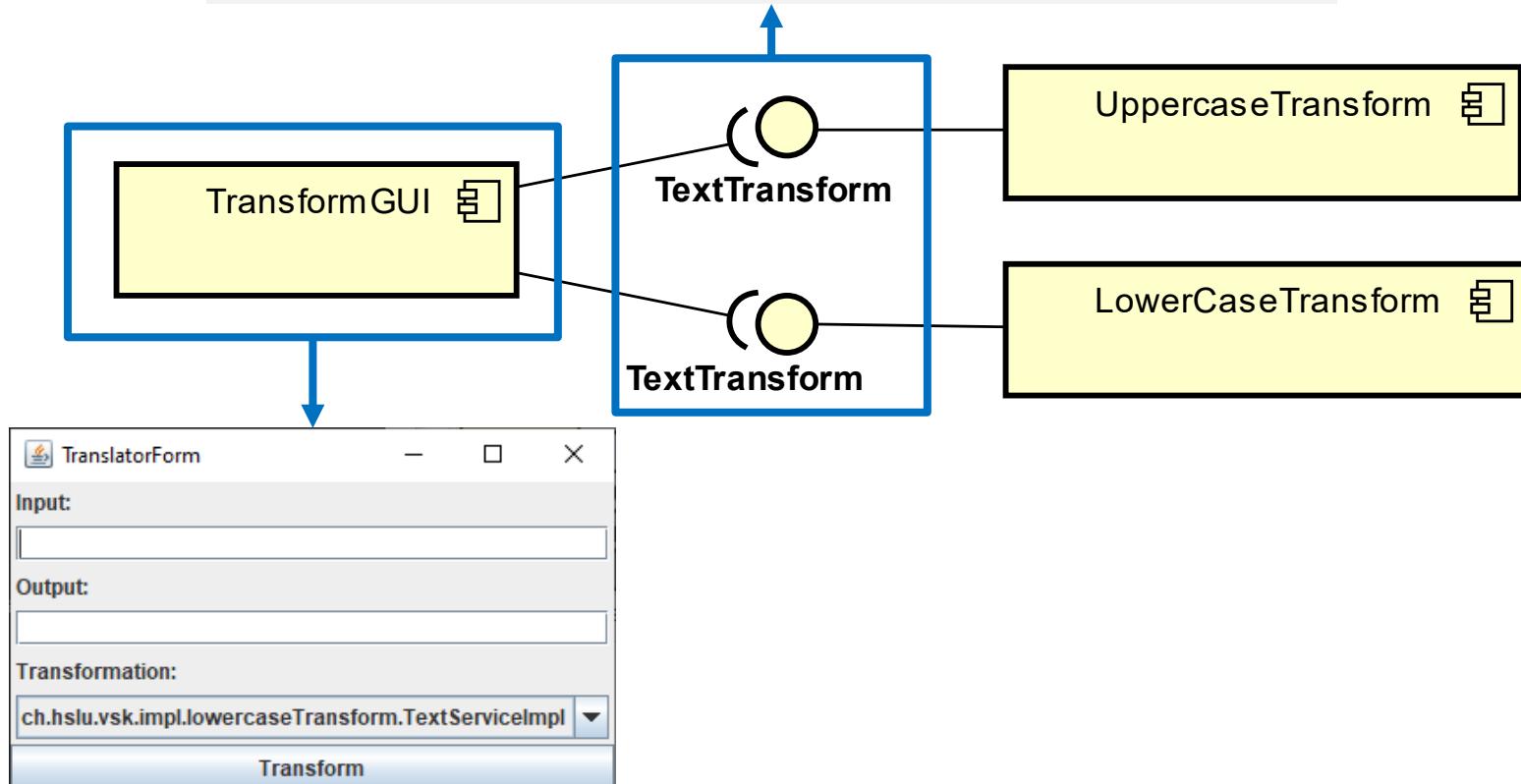


# Bestimmung der vier grundlegenden Eigenschaften

Eigenschaft	Beschreibung
<b>Schnittstellendefinition</b>	
<b>Kommunikation und Verteilung</b>	
<b>Komposition und Auffindbarkeit</b>	
<b>Deployment</b>	

# Kleine Demonstration mit drei Komponenten

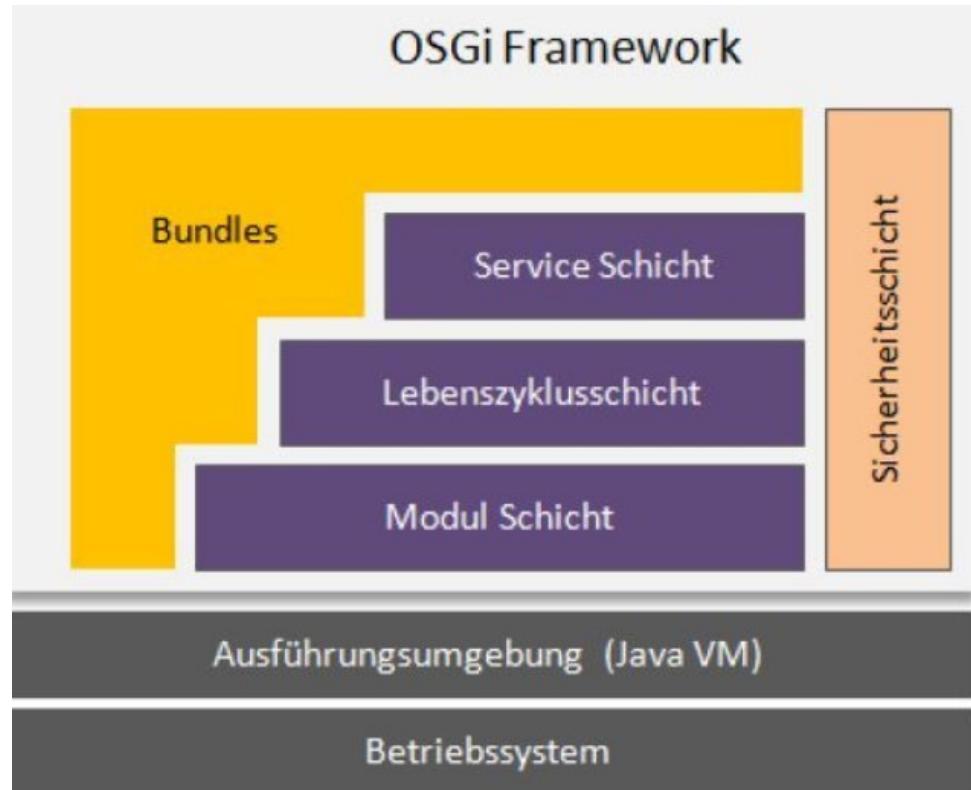
```
public interface TextService {  
    /**  
     * Returns the input string in uppercase letters.  
     * @param text to be converted.  
     * @return text in uppercase letters.  
     */  
    String translate(String text);  
}
```



# **Das OSGi-Komponentenmodell**

# OSGi: Übersicht

- Leichtgewichtiges Modul- und Servicesystem für Java.
- Standard: Implementationen verschiedener Hersteller verfügbar.
- Definiert ein vollständiges Komponentenmodell.



Quelle: Modularisierung mit Java 9, Guido Oelmann

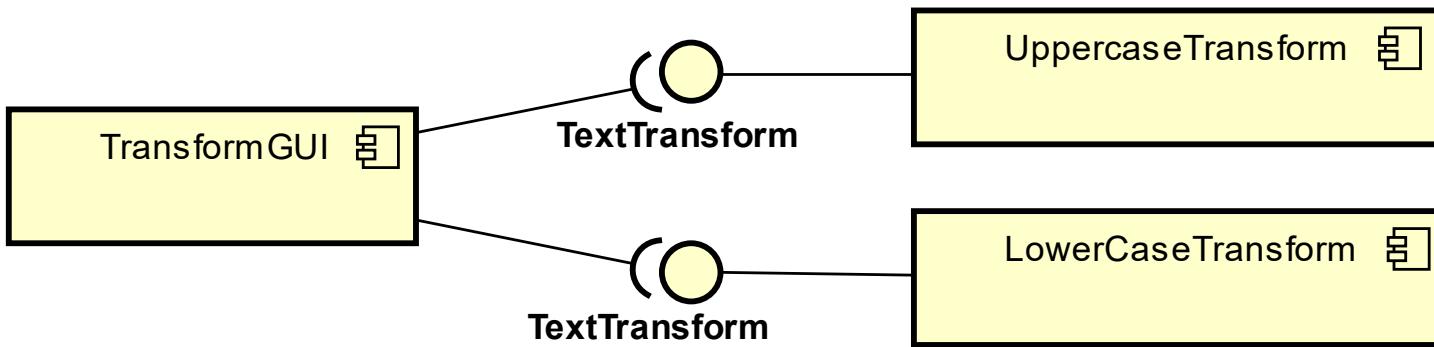
# Anbieter und Anwender von OSGi

- Anbieter (Auszug):
  - Apache Felix (Basis für Apache Karaf)
  - Gemini / Equinox OSGi (Spring/Eclipse)
  - Concierge OSGi (smallfoot print OSGi for embedded devices)
  - Knopflerfish (hat eine kommerzielle Pro-Version)
  - Diverse kommerzielle Anbieter (Samsung, Hitachi, etc.)
  - Diverse Applikationsserver (WebSphere, Glassfish, WildFly, etc.)
- Einige bekannte Anwender von OSGi:
  - NetBeans
  - Confluence und JIRA
  - Eclipse
  - IntelliJ
  - usw.

# Komponenten und Schnittstellen in OSGi

- Bundle ist die Bezeichnung von Komponenten bzw. Modul in OSGi.
- Sowohl Komponenten und Schnittstellen als Bundle geliefert.
- Bundle: JAR-Datei mit einem Manifest META-INF/MANIFEST.MF

## Beispiel mit drei Komponenten und einer Schnittstelle:



⇒ vier Bundles

# Beispiel eines OSGi-Bundle-Manifests

```
Manifest-Version: 1.0
Bnd-LastModified: 1616424844077
Build-Jdk-Spec: 11
Bundle-Activator: ch.hslu.vsk.textservice.ActivatorBundle
ManifestVersion: 2
Bundle-Name: Transform API (OSGi Variant)
Bundle-SymbolicName: ch.hslu.vsk.TransformApiBundle-Version:
1.0.0.SNAPSHOT
Created-By: Apache Maven Bundle Plugin
Export-Package:
ch.hslu.vsk.textservice;uses:="org.osgi.framework";version="1.0
.0"
Import-Package: org.osgi.framework;version="[1.4,2)"
Require-Capability:
osgi.ee;filter:="(&(osgi.ee=JavaSE)(version=11))"
Tool: Bnd-5.1.1.202006162103
```

# Schnittstellendefinition

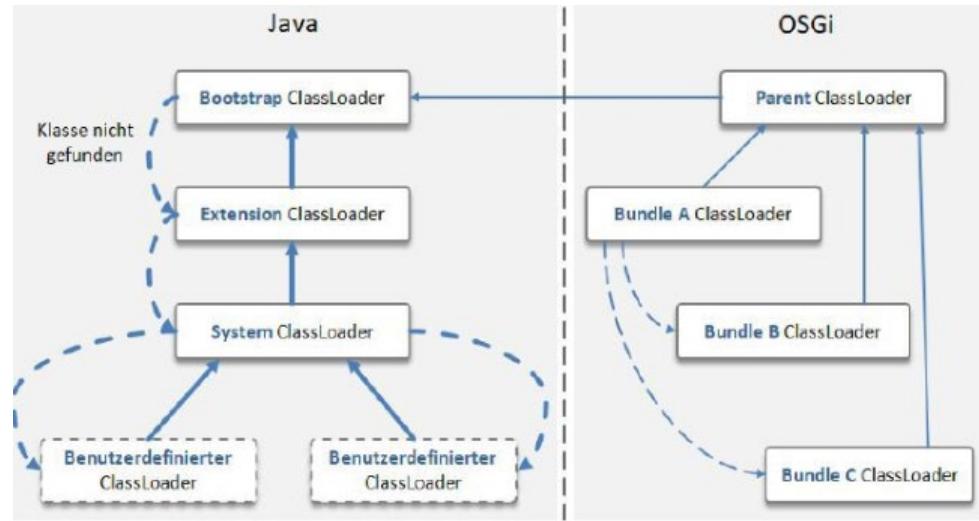
- Reguläre Java-Interfaces.
- Als separates Bundle (JAR).

```
package ch.hslu.vsk.textservice;

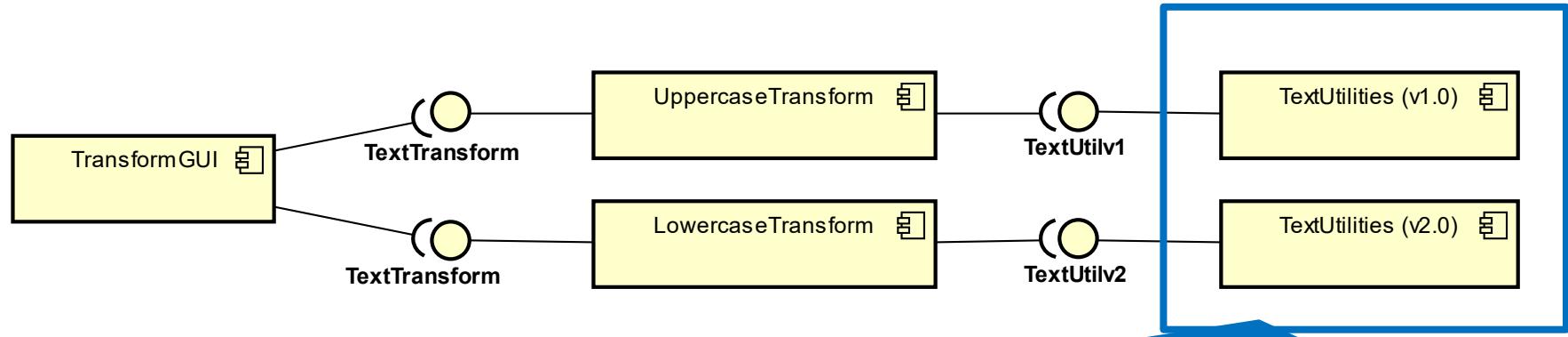
public interface TextService {
    /**
     * Transforms the input string and returns the transformation.
     * @param text to be transformed.
     * @return transformed text.
    */
    String translate(String text);
}
```

# Lösung der Versionierungsproblematik durch Isolation

- Nur exportierte Packages sind von anderen Bundles sichtbar.
- Ansatz: Verwendung von verschiedenen ClassLoader (automatisch).



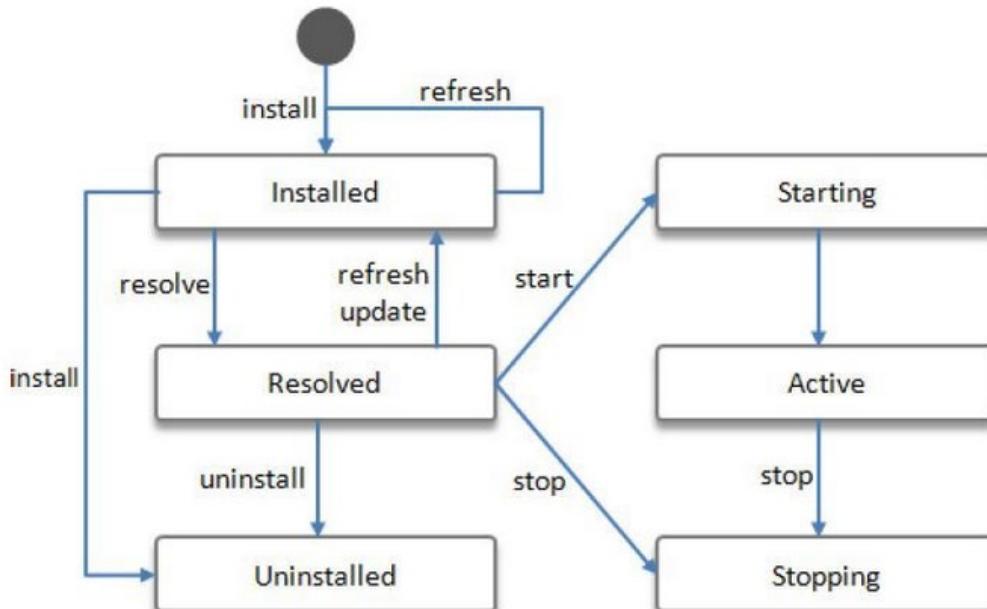
## Beispiel:



Verwenden beide Klasse: `ch.hslu.vsk.impl.util.StringUtil`

# Lebenszyklus von OSGi-Modulen

- Module werden dynamisch geladen, benutzt und entladen.
- Ideal für Plugins (Kein Neustart), aber keine garantierte Verfügbarkeit (analog zu Netzwerkdiensten).
- Bundle wird via BundleActivator (festgelegt in Bundle-Manifest) benachrichtigt, sobald es aktiviert wird.



Quelle: Modularisierung mit Java 9, Guido Oelmann

# Beispiel eines Bundle-Activators

```
public class Activator implements BundleActivator {  
    private JFrame main;
```

```
    public void start(BundleContext context) {  
        main = TransformerClient.createForm(context);  
    }
```

start(): Aufgerufen, wenn bundle aktiviert wird.

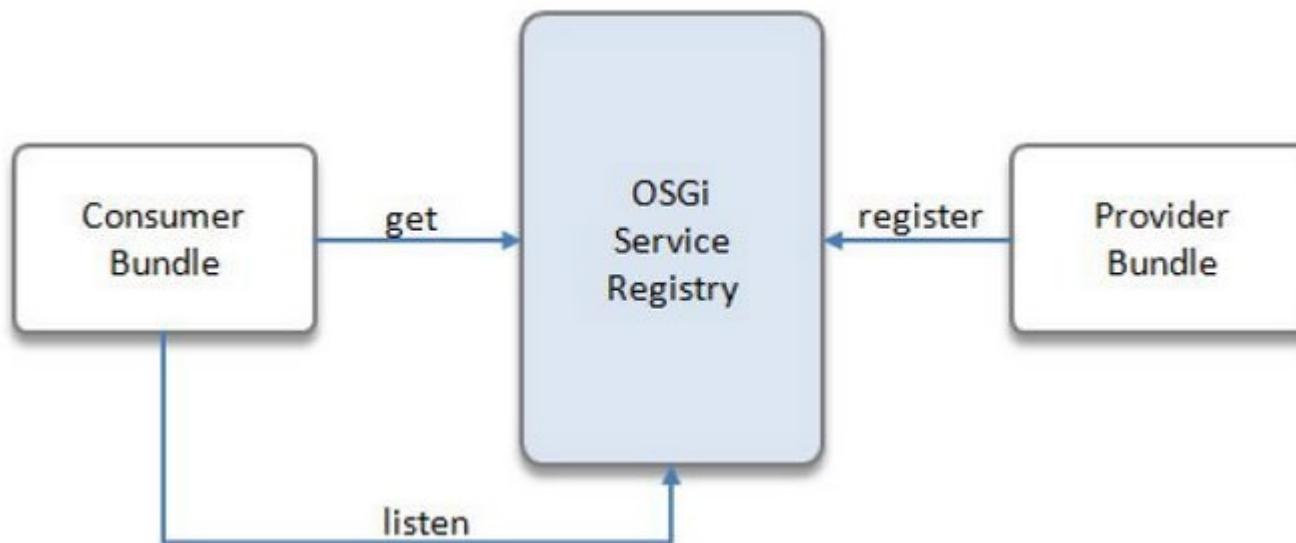
```
    public void stop(BundleContext context) {  
        main.dispose();  
    }
```

}

stop(): Aufgerufen, wenn bundle gestoppt wird.

# OSGi Services und Service Registry

- Service-Registry: Zentrales Verzeichnis über alle Services.
- Service: Objektinstanz, welche ein bestimmtes API implementiert und in der Service-Registry publiziert wird.



**Quelle:** Modularisierung mit Java 9, Guido Oelmann

# Beispiel: Registrieren und Auffinden von Services

- Registrieren einer Service-Instanz mit Zusatzinformationen:

```
Hashtable<String, String> properties = new Hashtable<>();  
properties.put("name", "Lowercase Transform");  
context.registerService(TextService.class.getName(),  
                      new TextServiceImpl(), properties);
```

- Finden einer Service-Instanz:

```
// Abrufen der Services, die das TextService API implementieren  
ServiceReference[] refs = context.getServiceReferences(  
                                         TextService.class.getName(), null);  
  
// Service pinnen  
TextService textService = (TextService) context.getService(refs[0]);  
  
// Service verwenden  
output = textService.translate(text);  
  
// Service unpinnen  
context.ungetService(refs[0]);
```

# OSGi als Komponentenmodell

Eigenschaft	Beschreibung
<b>Schnittstellendefinition</b>	<ul style="list-style-type: none"><li>▪ Java-Interfaces.</li></ul>
<b>Kommunikation (Verteilung)</b>	<ul style="list-style-type: none"><li>▪ Method-Calls.</li></ul>
<b>Komposition (Austauschbarkeit, Auffindbarkeit):</b>	<ul style="list-style-type: none"><li>▪ Import/Export von Interfaces.</li><li>▪ Service-Registry.</li><li>▪ Isolation der nicht-exportierten Packages.</li></ul>
<b>Deployment</b>	<ul style="list-style-type: none"><li>▪ Einzelne JARs (Bundles).</li><li>▪ Bundle-JAR.</li></ul>

# **Ein Komponentenmodell für Microservices**

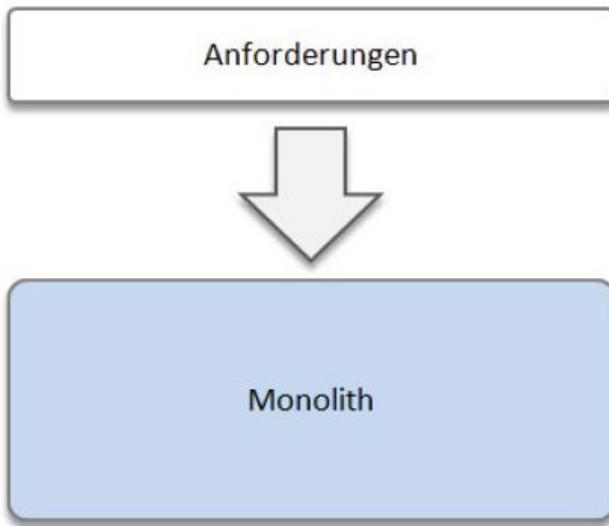
# Was sind Microservices?

"In short, the microservice architectural style is an approach to developing a single application as a **suite of small services**, each **running in its own process** and communicating with lightweight mechanisms, often an HTTP resource API. These services are **built around business capabilities** and **independently deployable** by fully automated deployment machinery. There is a **bare minimum of centralized management** of these services, which may be written in different programming languages and use different data storage technologies." (James Lewis and Martin Fowler, 2014)

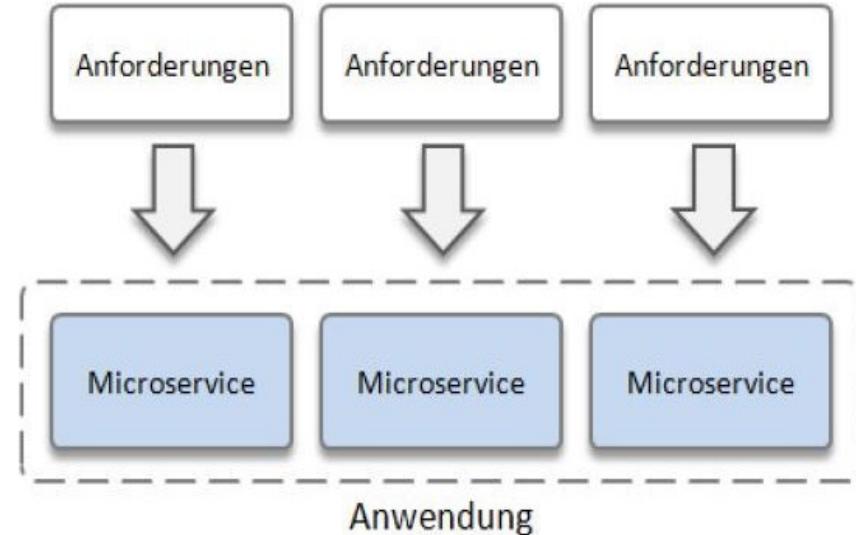
**Hinweis:** Nur Einordnung von Microservice als Komponente.  
Einführung der Microservices-Architektur erfolgt in Modul SWDA.

# Aufteilung nach "Business Capabilities"

- **Ziel:** Bessere Aufteilung auf verschiedene Entwicklungsteams:  
Konzentration auf Fachdomänen und ggf. separate Auslieferung.



**Ohne Microservices**

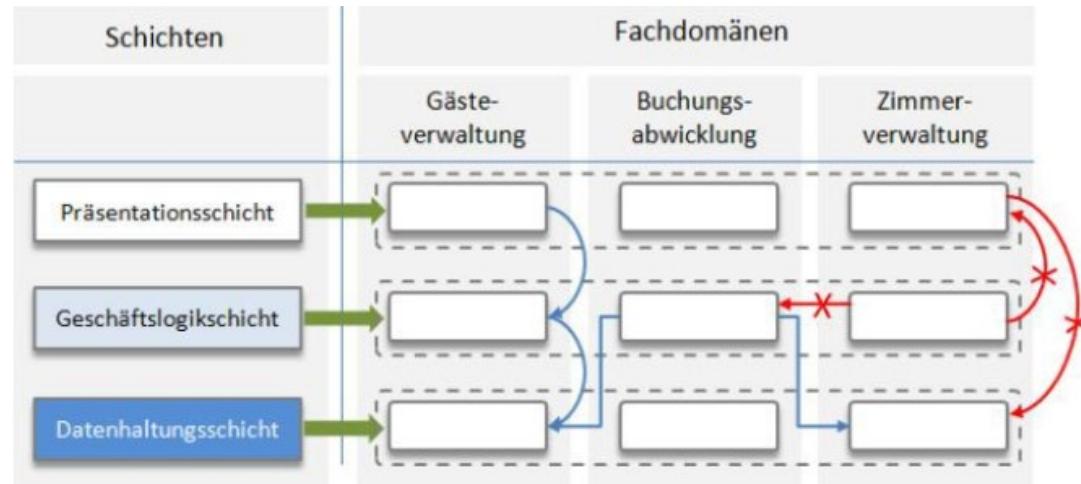


**Mit Microservices**

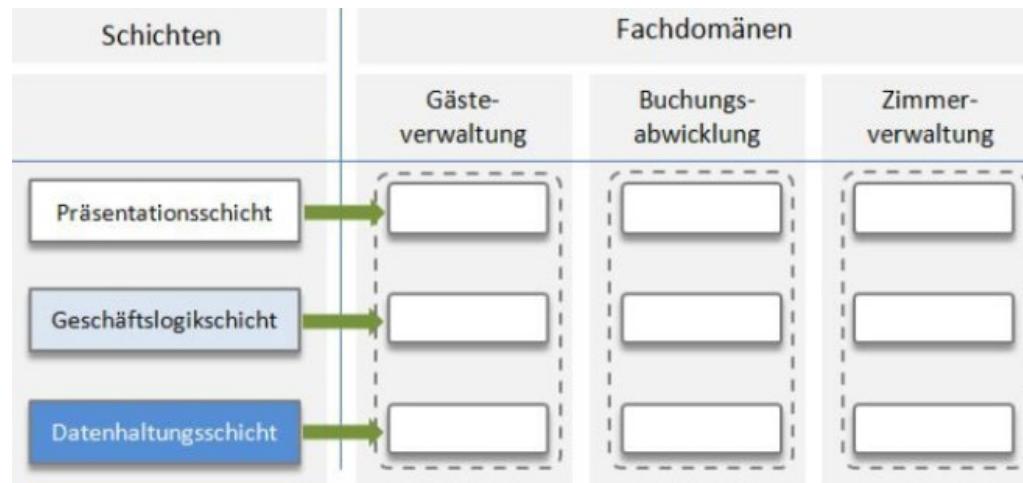
**Bildquellen:** Modularisierung mit Java 9, Guido Oelmann

# Beispiel: Hotelbuchungssystem

## Aufteilung nach Schichten:



## Aufteilung nach Fachdomänen:



Welche Aufteilung ist besser?

**Bildquellen:** Modularisierung mit Java 9, Guido Oelmann

# Eigenschaften von Microservices

- **Aufteilung in Services:** Anwendung in kleine Services unterteilt, welche als Komponenten verwendet werden.
- **Schichten übergreifend:** enthält alle Schichten einer Funktionalität / Domäne (Frontend / Businesslogik / Persistenz).
- **Unabhängiges Deployment** (der kleinen Services).
- **Kommunikation über Netzwerk:** REST, MQ, SOAP, etc.
- **Entkopplung:** Microservice nur über Schnittstellen bekannt. Ausführung als eigener Prozess.
- **Technologische Entkopplung:** pro Microservices kann Plattform, Datenbank, Programmiersprache individuell festgelegt werden.

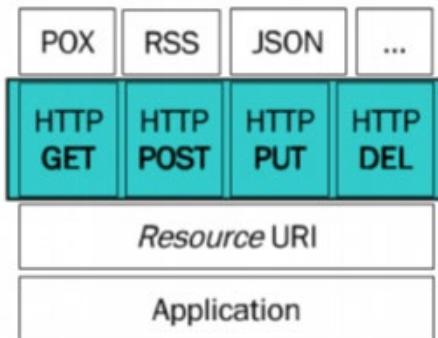
# Komponentenmodell für Microservices

Eigenschaft	Beschreibung
<b>Schnittstellendefinition</b>	???
<b>Kommunikation (Verteilung)</b>	???
<b>Komposition (Austauschbarkeit, Auffindbarkeit):</b>	???
<b>Deployment</b>	???

# Kommunikation

- Muss eine Interprozess-Kommunikation sein.
- Fowler und Lewis: "often an HTTP resource API".

## REST



URL	Method	POST Body	Result
http://example.com/entries	GET	empty	Returns all entries
http://example.com/entries	POST	JSON String	New entry created (ID)
http://example.com/entries/<id>	GET	Empty	Returns single entry
http://example.com/entries/<id>	PUT	JSON string	Updates existing entry
http://example.com/entries/<id>	DELETE	empty	Deletes existing entry

# Komponentenmodell für Microservices

Eigenschaft	Beschreibung
<b>Schnittstellendefinition</b>	???
<b>Kommunikation (Verteilung)</b>	<ul style="list-style-type: none"><li>▪ HTTP REST API</li><li>▪ Verteilung über virtuelle und physische Systeme möglich.</li></ul>
<b>Komposition (Austauschbarkeit, Auffindbarkeit):</b>	???
<b>Deployment</b>	???

# Schnittstellendefinition

- Populär ist aktuell OpenAPI (<https://www.openapis.org>).

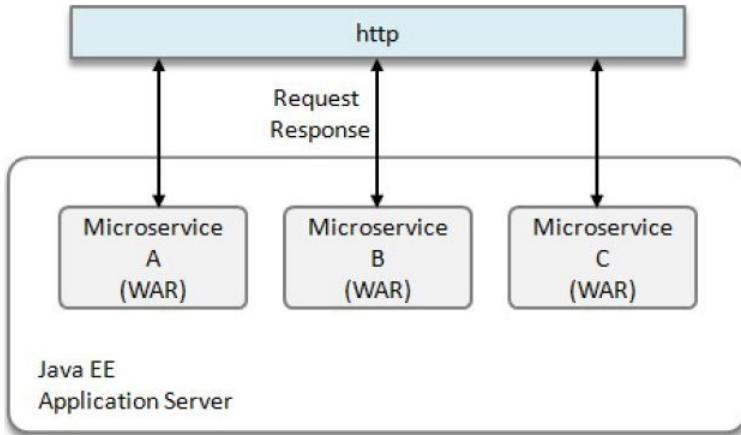
```
openapi: "3.0.2"
info:
  title: "TransformApi"
  version: '1.0'
  description: "Interface for text transformations."
  license:
    name: "Apache License 2.0"
    url: "http://www.apache.org/licenses/LICENSE-2.0"
paths:
  /transform:
    get:
      summary: "Applies transformation to input"
      description: "Applies transformation to input"
      parameters:
        - name: text
          in: query
          description: "Contains text to transform"
          schema:
            type: string
... and more ...
```

# Komponentenmodell für Microservices

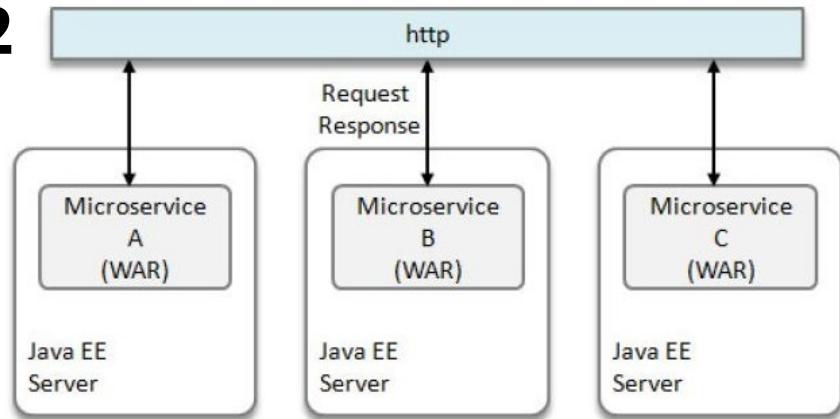
Eigenschaft	Beschreibung
<b>Schnittstellendefinition</b>	<ul style="list-style-type: none"><li>▪ OpenAPI</li></ul>
<b>Kommunikation (Verteilung)</b>	<ul style="list-style-type: none"><li>▪ HTTP REST API</li><li>▪ Verteilung über virtuelle und physische Systeme möglich.</li></ul>
<b>Komposition (Austauschbarkeit, Auffindbarkeit):</b>	???
<b>Deployment</b>	???

# Varianten des Deployments

1



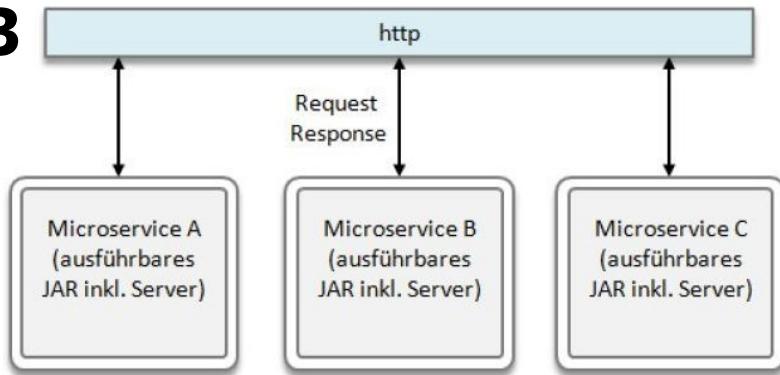
2



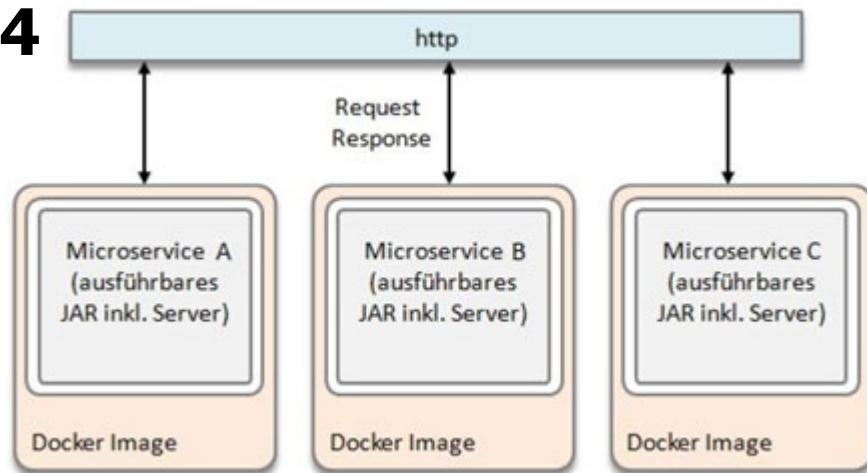
Kein eigener Prozess: Normale Webservices.

Bündeln mit vollwertigem Applikationsserver:  
Komplex (Konfiguration) / viele Ressourcen.

3



4



JAR -> Technologie neutral?

Container-Images mit allen Abhängigkeiten.

**Bildquellen:** Modularisierung mit Java 9, Guido Oelmann

# Komponentenmodell für Microservices

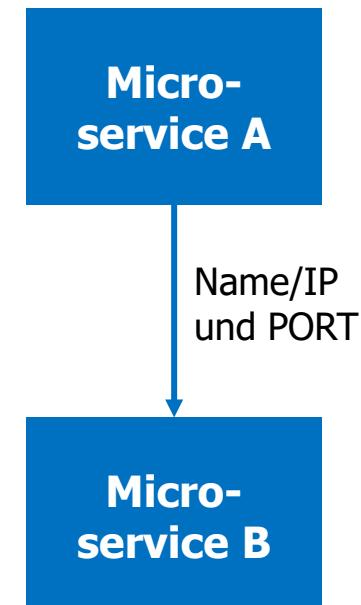
Eigenschaft	Beschreibung
<b>Schnittstellendefinition</b>	<ul style="list-style-type: none"><li>▪ OpenAPI</li></ul>
<b>Kommunikation (Verteilung)</b>	<ul style="list-style-type: none"><li>▪ HTTP REST API</li><li>▪ Verteilung über virtuelle und physische Systeme möglich.</li></ul>
<b>Komposition (Austauschbarkeit, Auffindbarkeit):</b>	???
<b>Deployment</b>	<ul style="list-style-type: none"><li>• Gebündelt mit HTTP-Server und allen Abhängigkeiten als Container-Image.</li><li>• Abgelegt in einem binär Repository.</li></ul>

# Verbinden von Services (Import / Export)

- Regulär via Hostname/IP und Port, jedoch sehr variable Umgebung:  
Viele Services, unabhängiges Deployment, Skalierung,  
Virtualisierung, Load-Balancing, Fault-Tolerance, usw.
- Statische Zuweisung von einem Microservice zum anderen z.B. via properties-File möglich, aber nicht praktikabel.

## Mögliche Variante:

- **Export:** Container-Laufzeitumgebungen vergeben oft **automatisch** Namen (DNS) an Container.
- **Import:** Beim Start eines Containers können vergebene Namen in einer Systemumgebungs-Variable oder als Parameter **automatisch** an Konsumenten übergeben werden.



# Komponentenmodell für Microservices

Eigenschaft	Beschreibung
Schnittstellendefinition	<ul style="list-style-type: none"><li>▪ OpenAPI</li></ul>
Kommunikation (Verteilung)	<ul style="list-style-type: none"><li>▪ HTTP REST API.</li><li>▪ Verteilung über virtuelle und physische Systeme möglich.</li></ul>
Komposition (Austauschbarkeit, Auffindbarkeit):	<ul style="list-style-type: none"><li>▪ Orchestrierung:<ul style="list-style-type: none"><li>▪ Automatische Vergabe von Name/IP und Port an Service-Anbieter.</li><li>▪ Automatische Übergabe von Name/IP und Port an Konsument z.B. mit Umgebungsvar.</li></ul></li><li>▪ Isolation durch unabhängige Prozesse.</li></ul>
Deployment	<ul style="list-style-type: none"><li>• Gebündelt mit HTTP-Server und allen Abhängigkeiten als Container-Image.</li><li>• Abgelegt in einem binär Repository.</li></ul>

# Zusammenfassung

- Komponentenmodelle bilden einen Rahmen innerhalb dessen zwei oder mehr Komponenten direkt zusammengesteckt werden können.
- Ein vollständiges Komponentenmodell legt mindestens folgende Eigenschaften fest: Schnittstellendefinition, Kommunikation, Komposition, Deployment.
- Der OSGi-Standard definiert ein vollständiges Komponentenmodell auf Basis von Java, welches ein dynamisches Laden von Modulen (Komponenten) innerhalb einer JVM sowie die Definition von Services ermöglicht.
- Microservices ist eine Softwarearchitektur, aber kein Komponentenmodell. Wir können jedoch ein Komponentenmodell für Microservices herleiten.

# **Fragen?**

# Literatur

- Component-Based Software Engineering von Bill Councill und George T. Heineman, Addison-Wesley, 2001.
- Modularisierung mit Java 9 von Guido Oelmann, dpunkt.verlag GmbH, 2017.
- Microservices - a definition of this new architectural term von James Lewis and Martin Fowler, [www.martinfowler.com](http://www.martinfowler.com), 2014.
- OpenAPI Specification Version 3.1.0, OpenAPI Initiative, a Linux Foundation Collaborative Project.

# Clean Code

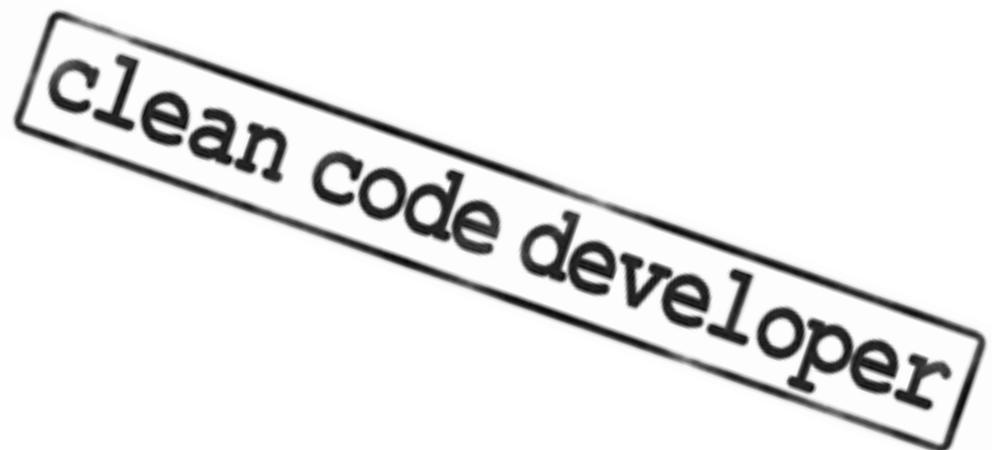
**Warum sich sauberer Code lohnt!**



JUGS-Referat von Roland Gisler

# Inhalt

- Teil 1: Clean Code
- Teil 2: Clean Code Developer
- Teil 3: Clean Code [Developer] – Praxistipps
- Fragen und Diskussion



# Teil 1: Clean Code



# Woher kommt 'Clean Code'?

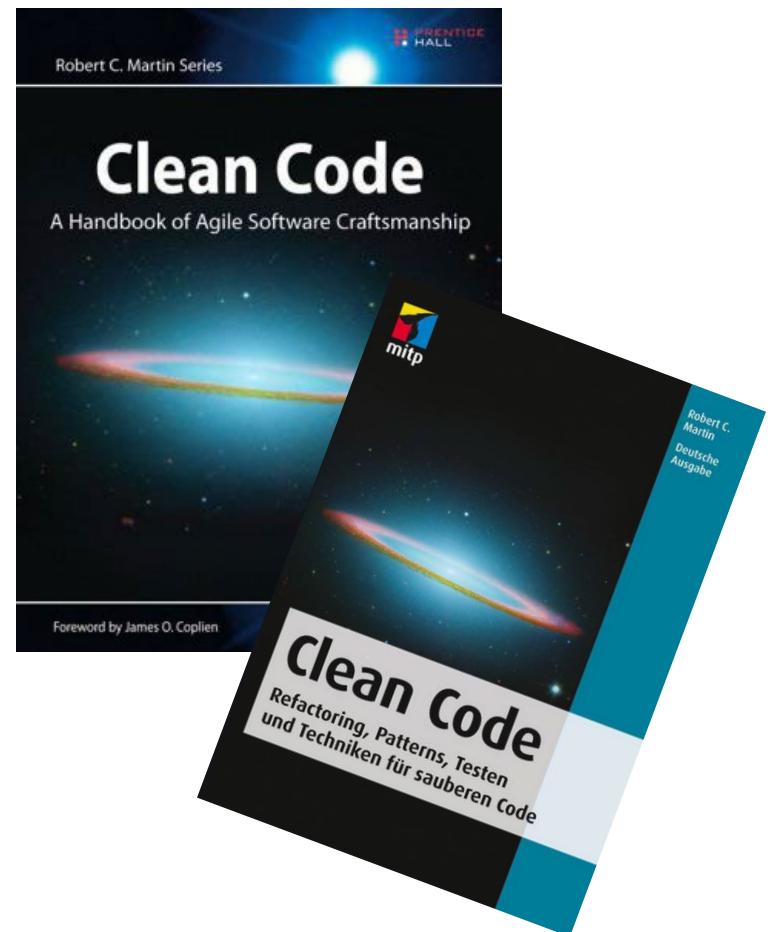


Robert C. Martin

# Clean Code – das Buch

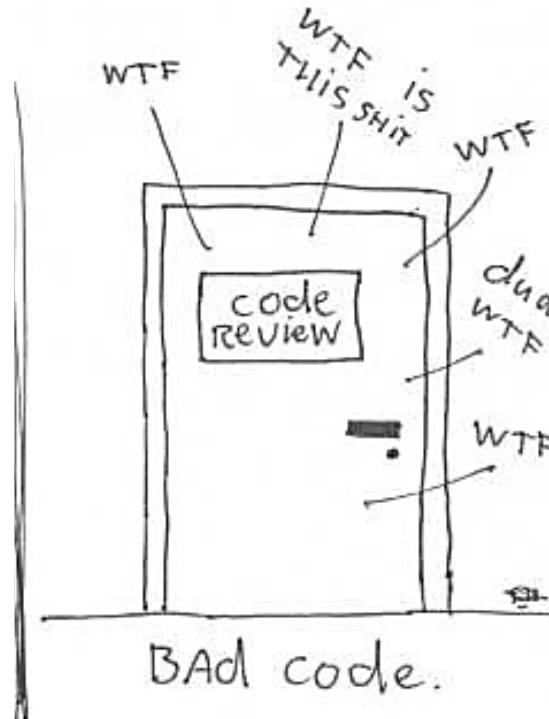
*Robert C. Martin:*  
**Clean Code**

A Handbook of Agile Software Craftsmanship  
Prentice Hall, März 2009,  
ISBN: 978-0-13-235088-4 (englisch)  
ISBN: 978-3-8266-5548-7 (deutsch)



# Code-Qualität verbessern!

The ONLY VALID measurement  
OF code QUALITY: WTFs/MINUTE

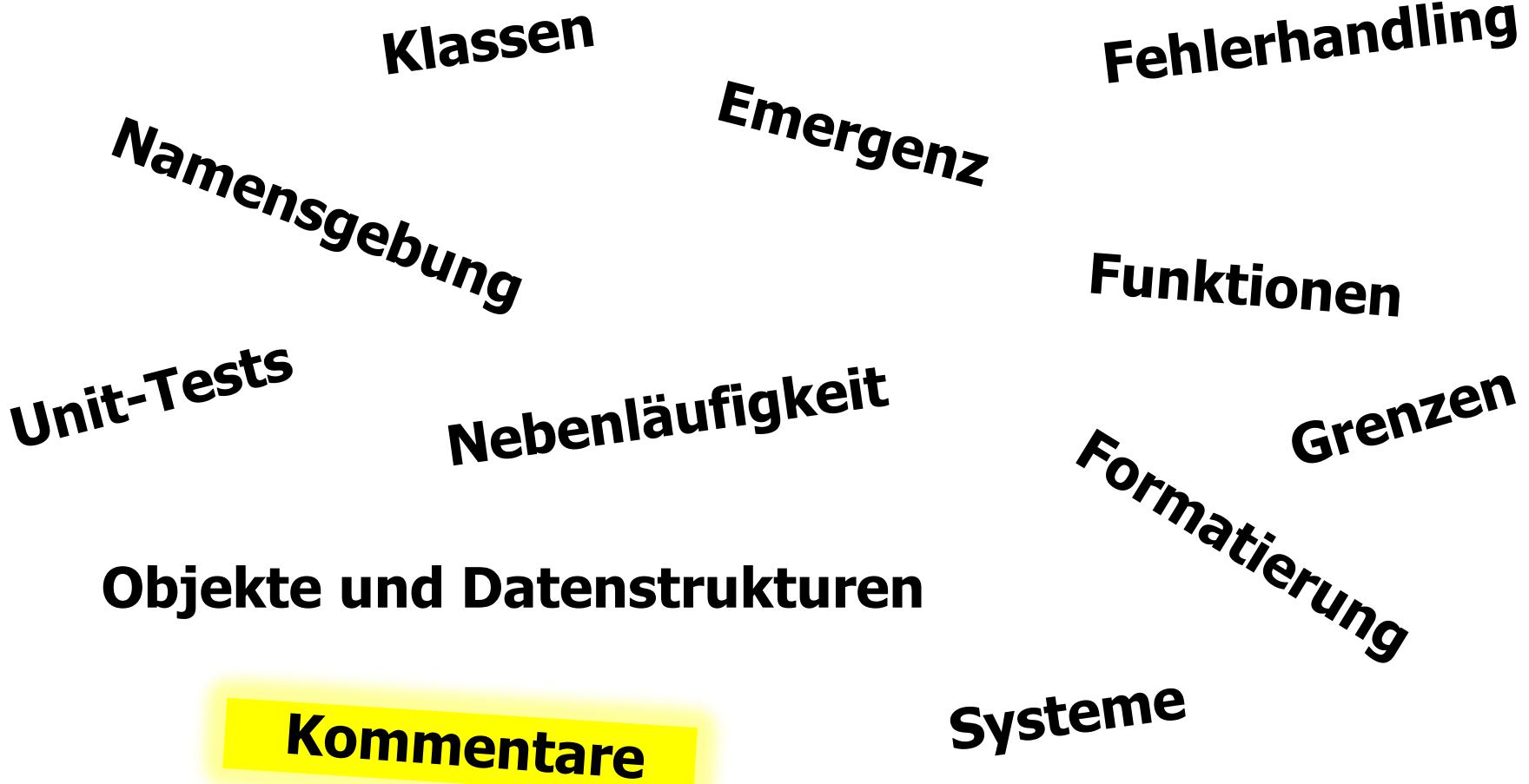


(c) 2008 Focus Shift

# SW-Entwicklung als ein «echtes» Handwerk!



# Themen von «Clean Code»



# **Soll guter Quellcode kommentiert sein?**

# Quellcode Kommentare

- Kommentare machen keinen guten Code!

**Kommentieren Sie schlechten  
Code nicht – schreiben Sie ihn um!**

*Brian W. Kernighan / P. J. Plauger*

- Kommentare (und auch Dokumentationen) lügen!
  - Nur im Code liegt die Wahrheit!
- Kommentare sind ein notwendiges Übel!
  - fundamentaler Wechsel des Paradigmas!

# Gute Kommentare

- Grundsatz: **Der beste Kommentar ist derjenige, den man gar nicht zu schreiben braucht!**
  - Energie besser in guten, selbsterklärenden Code stecken!
- Notwendige, akzeptable Kommentare können sein:
  - juristische Kommentare (Copyright etc.)
  - TODO-Kommentare (aber nur temporär!)
  - verstärkende, unterstreichende Kommentare, welche Dinge hervorheben, die sonst zu unauffällig wären
  - Kommentare zur (Er-)Klärung der übergeordneten Absicht oder zur expliziten Warnung vor Konsequenzen

# Schlechte Kommentare (1/2)

- Redundante Kommentare ([CCD:DRY](#))
  - reine Wiederholungen dessen, was schon der Code sagt
- Irreführende Kommentare
  - falsche oder unpräzise Formulierungen
- vorgeschriebene oder erzwungene Kommentare
  - sture JavaDoc Kommentare, nur damit sie da sind.
- Tagebuch- oder Changelog-Kommentare
  - heute haben wir Versionskontrollsysteme!
- Positionsbezeichner und Banner
  - zur optischen Unterteilung von grossen Quellcodedateien

## Schlechte Kommentare (2/2)

- Zuschreibungen und Nebenbemerkungen
  - Hinweise auf Autor, sinnlose Zusatzbemerkungen
- Auskommentierter Code
  - Ein Todsünde! Kurioserweise traut sich (fast) niemand diesen zu löschen. Wir haben Versionskontrollsysteme!
- HTML(-formatierte) Kommentare
  - Kommentar muss im Code direkt lesbar sein. Nicht erst in der JavaDoc.
- zu viel Kommentar / Information
  - endlose Abhandlungen über Gott und die Welt

# Beispiel: Ein guter Kommentar?

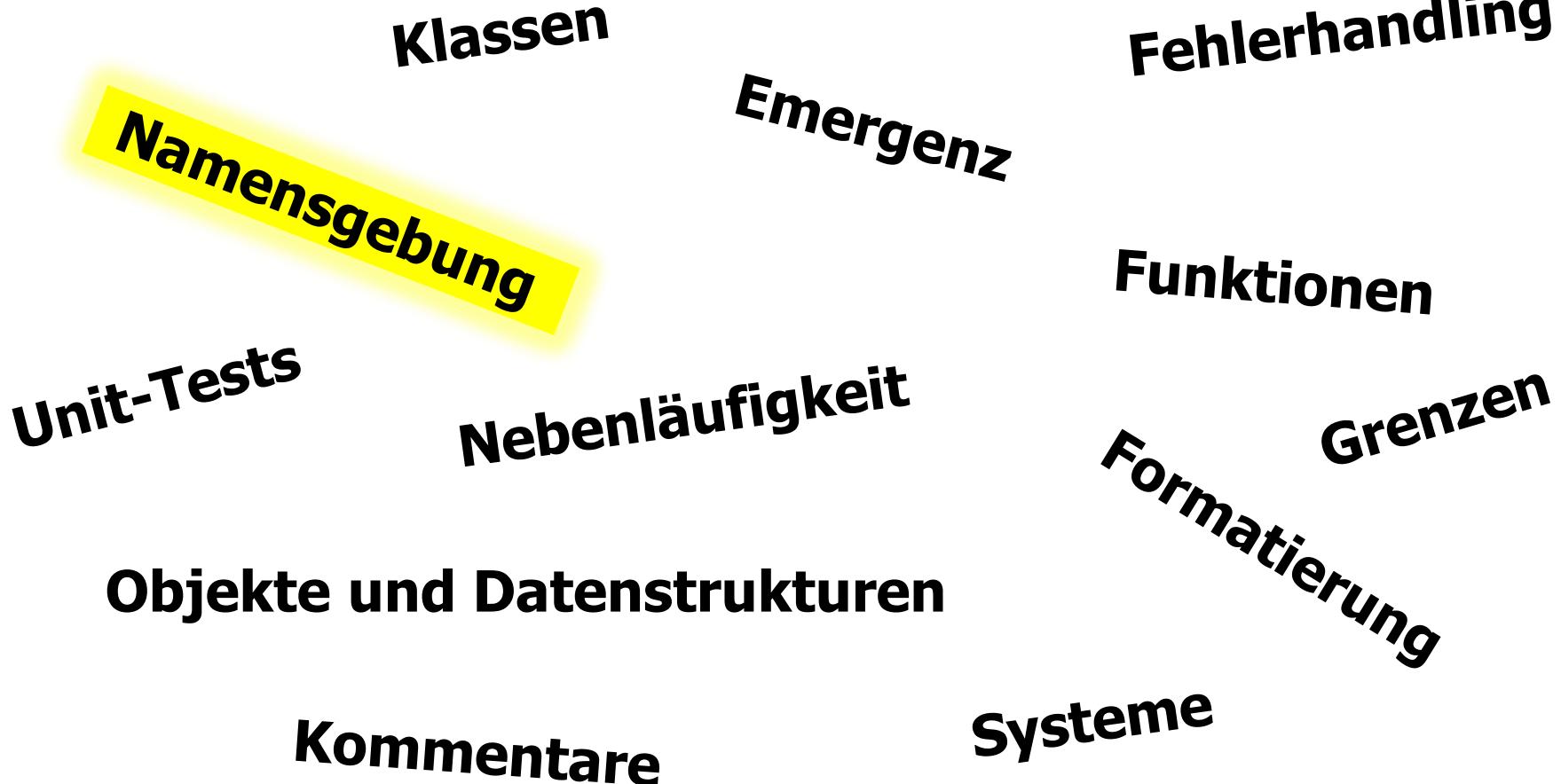
```
// Timeout in Millisekunden  
int time = 100;
```

- Der Kommentar kann entfallen, wenn der Code verbessert wird:

```
int timeoutMilliseconds = 100;
```

- Bessere Namensgebung erspart uns viele Kommentare!
- Gute Namensgebung ist aber anspruchsvoll!

# Themen von Clean Code



# Gute Namensgebung als Herausforderung

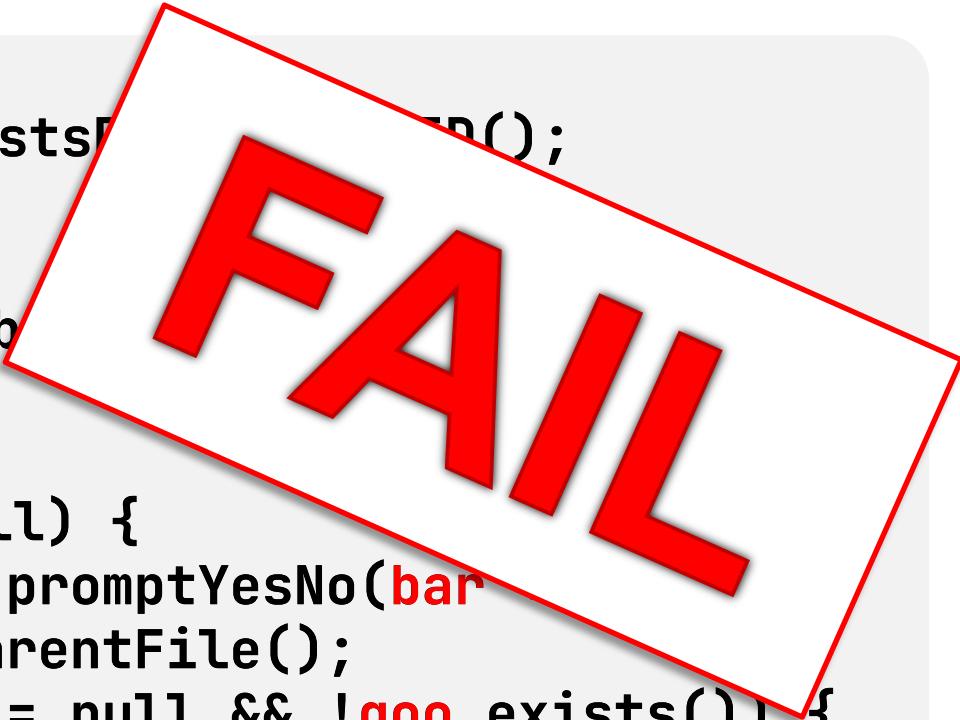
- Überlege dir den Namen einer Klasse so gut, wie den Namen eines Kindes
  - Ein Kind (und die Klasse) trägt ihn ein Leben lang!
  - Besonders heikel: Interfaces!
- Ein richtig guter Name sollte
  - absolut zweckbeschreibend sein
  - Fehlinformationen vermeiden
  - Unterschiede deutlich machen (differenzierend sein)
  - gut aussprechbar und gut suchbar sein
  - möglichst keine Codierungen enthalten

# Namensgebung: Heuristiken N1 – N7

- **N1:** Beschreibende Namen wählen
- **N2:** Namen passend zur Abstraktionsebene wählen
- **N3:** Standardnomenklatur verwenden
- **N4:** Eindeutige Namen wählen
- **N5:** Namenlänge abhängig vom Geltungsbereich
- **N6:** Codierungen vermeiden
- **N7:** Namen sollten auch Nebeneffekte beschreiben

# Beispiel 1: Gute Namensgebung?

```
...
String bar = getKnownHosts();
if(bar != null) {
    boolean foo = true;
    File goo = new File(b
    if(!goo.exists()) {
        foo = false;
        if(userinfo != null) {
            foo = userinfo.promptYesNo(bar
            goo = goo.getParentFile();
            if(foo && goo != null && !goo.exists()){
                ...
            }
        ...
    }
}
```



Quellcodeausschnitt aus JSch-Library (KnownHosts.java); Copyright (c) 2002-2012 ymnk, JCraft, Inc. All rights reserved.

## Beispiel 2: Gute Namensgebung!

```
String removeLastClosingCurlyBrace(String message) {  
    ...  
}
```

# Themen von Clean Code



# Teil 2: Clean Code Developer



# Woher kommt 'Clean Code Developer'?



Ralf Westphal und Stefan Lieser

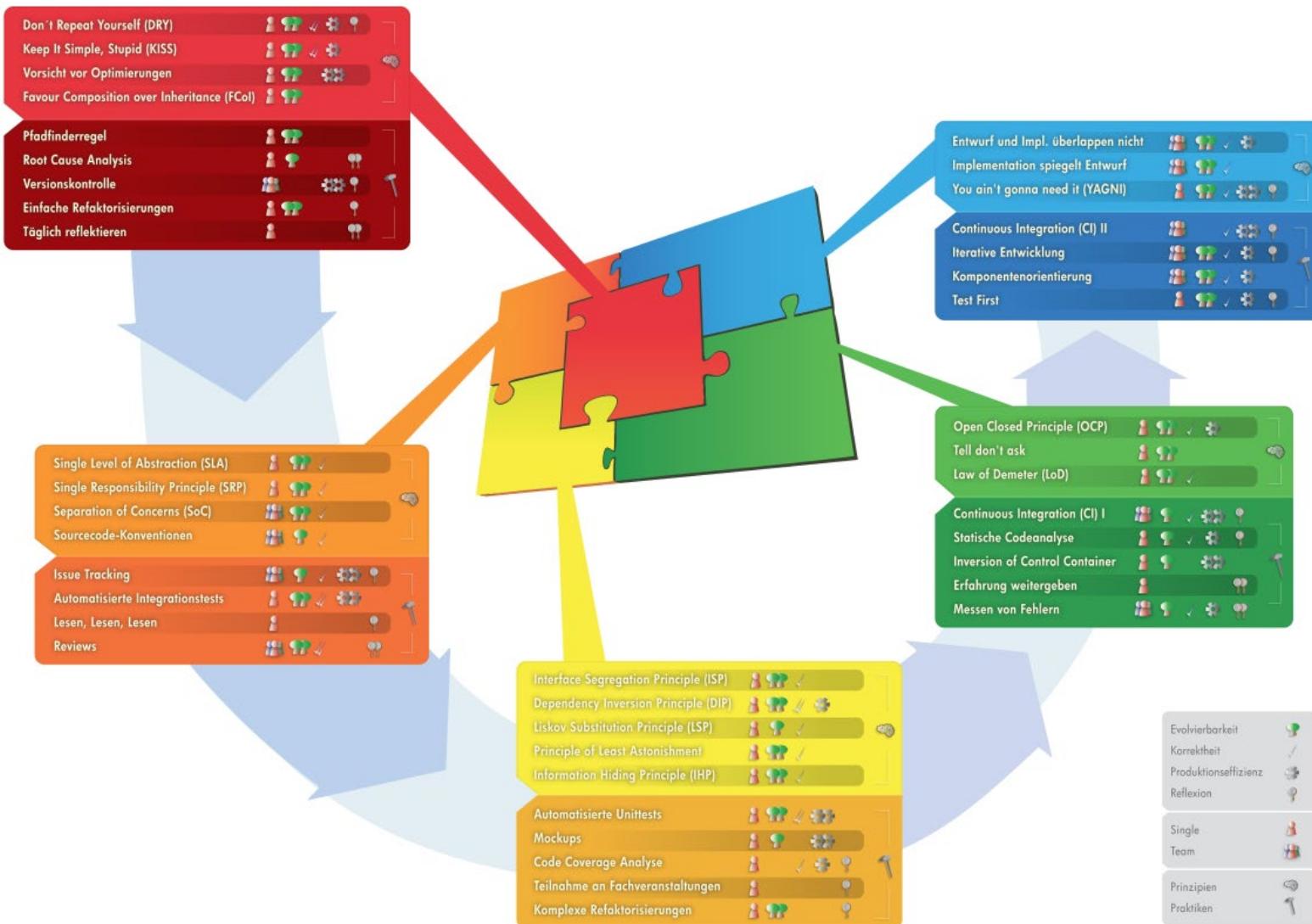
# Was ist 'Clean Code Developer'?

- Clean Code Developer ist eine wohldefinierte Auswahl von Prinzipien und Praktiken
- Basis ist ein Wertesystem:
  - Evolvierbarkeit
  - Korrektheit
  - Produktionseffizienz
  - Reflexion

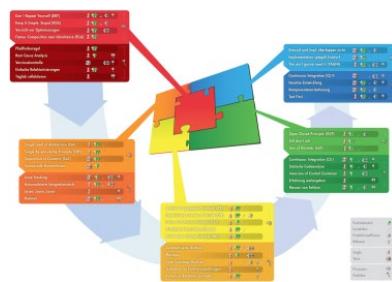
The screenshot shows two views of the 'Clean Code Developer' website. The left view is the homepage, featuring a sidebar with links like 'Das Wertesystem', 'Die Tugenden', 'Die Grade', 'Die Initiative', 'Weitere Infos', and 'Impressum'. Below this is a section titled 'Diskutieren' with links to Google Gruppe, Xing Gruppe, and Vorschlagswesen. At the bottom, it says 'Unterstützt von' and lists 'SIGS DATACOM FACHINFORMATIONEN FÜR IT-PROFESSIONALS' and 'dotnetpro'. The right view is a detailed page under 'Professionalität = Bewusstsein + Prinzipien', which discusses the need for professionals in software development and the principles behind it. It also includes a section on 'Minimale Anforderungen an Professionalität'.

<http://www.clean-code-developer.de>

# Iteration über sieben Grade

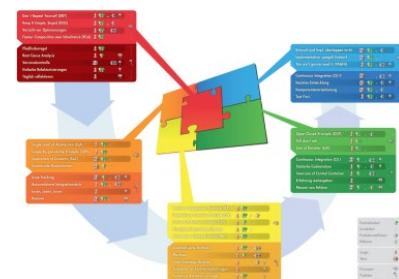


# Iteration über sieben Grade



- Grade: **Schwarz, Rot, Orange, Gelb, Grün, Blau, Weiss**
  - In Anspielung an die verschiedenen Gürtel im Judo
- Jeder einzelne Grad fokussiert auf eine relativ kleine, überschaubar ausgewählte Menge von Prinzipien und Praktiken
  - Insgesamt sind es **42** einzelne Items
  - Überblickbare Ziele und doch stetige Verbesserung.

# **Der erste Grad – Schwarz**



- Sie wissen nun bereits was Clean Code Developer ist, und haben damit den ersten, **schwarzen Grad** erreicht!

# Gratulation!

The image consists of two main text elements. At the top left, the word "ratulation!" is written in a large, bold, dark blue sans-serif font. To the right of the center, there is a large, tilted rectangular box with a black double-line border. Inside this box, the words "clean code developer" are written in a large, bold, black, sans-serif font. The entire graphic is set against a plain white background.

# Der zweite Grad – Rot

- Prinzipien
  - **Don't Repeat Yourself (DRY)**
  - **Keep it simple, stupid (KISS)**
  - Vorsicht vor Optimierungen
  - **Favour Composition over Inheritance (FCoI)**
- Praktiken
  - Die Pfadfinderregel beachten
  - **Root Cause Analysis (RCA)**
  - Ein Versionskontrollsystem einsetzen
  - Einfache Refaktorisierungsmuster anwenden
  - Täglich reflektieren



\* siehe Bloch: Effective Java

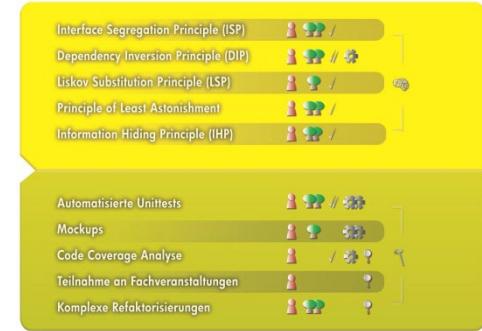
# Der dritte Grad – Orange

- Prinzipien
  - **Single Level of Abstraction (SLA)**
  - **Single Responsibility Principle (SRP)**
  - **Separation of Concerns (SoC)**
  - Source Code Konventionen: Namensregeln, Kommentare
- Praktiken
  - Issue Tracking
  - Automatisierte Integrationstests
  - Lesen, Lesen, Lesen
  - Reviews



# Der vierte Grad – Gelb

- Prinzipien
  - Interface Segregation Principle (ISP)
  - Dependency Inversion Principle (DIP)
  - Liskov Substitution Principle (LSP)
  - Principle of Least Astonishment
  - Information Hiding Principle (IHP)
- Praktiken
  - Automatisierte Unit Tests
  - Mockups (Testattrappen)
  - Code Coverage Analyse
  - Teilnahme an Fachveranstaltungen
  - Komplexe Refaktorisierungen



# Der fünfte Grad – Grün

- Prinzipien
  - **Open Closed Principle (OCP)**
  - Tell, don't ask
  - Law of Demeter
- Praktiken
  - **Continuous Integration (CI) I**
  - Statische Codeanalyse (Metriken)
  - Inversion of Control Container
  - Erfahrung weitergeben
  - Messen von Fehlern



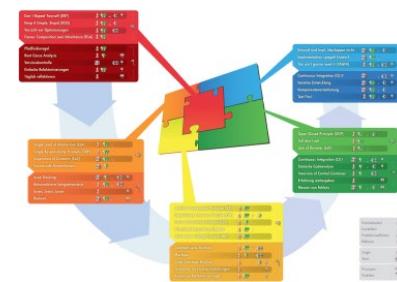
# Der sechste Grad – Blau

- Prinzipien
  - Implementation spiegelt Entwurf
  - Entwurf und Implementation überlappen nicht
  - You Ain't Gonna Need It (YAGNI)
- Praktiken
  - Continuous Integration (CI) II
  - Iterative Entwicklung
  - Komponentenorientierung
  - Test First

Entwurf und Impl. überlappen nicht   
Implementation spiegelt Entwurf   
You ain't gonna need it (YAGNI) 

Continuous Integration (CI) II   
Iterative Entwicklung   
Komponentenorientierung   
Test First 

# Der siebte Grad – Weiss



- Weisser Grad vereinigt alle Prinzipien und Praktiken der farbigen Grade
- Eine gleichschwebende Aufmerksamkeit ist jedoch sehr schwer zu halten
  - darum beginnt der Clean Code Developer im Gradesystem nach einiger Zeit wieder von vorne
- Die zyklische Wiederholung bringt stetige Verbesserung auf der Basis von überschaubaren Schwerpunkten
- **CCD wird somit zur verinnerlichten Einstellung!**

Don't Repeat Yourself (DRY)



Keep It Simple, Stupid (KISS)



Vorsicht vor Optimierungen



Favour Composition over Inheritance (FCoI)



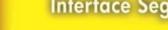
Pfadfinderregel



Root Cause Analysis



Versionskontrolle



Einfache Refaktorisierungen



Täglich reflektieren



Entwurf und Impl. überlappen nicht



Implementation spiegelt Entwurf



You ain't gonna need it (YAGNI)



Continuous Integration (CI) II



Iterative Entwicklung



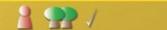
Komponentenorientierung



Test First



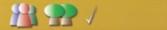
Single Level of Abstraction (SLA)



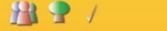
Single Responsibility Principle (SRP)



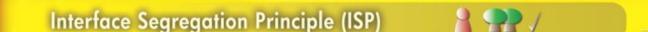
Separation of Concerns (SoC)



Sourcecode-Konventionen



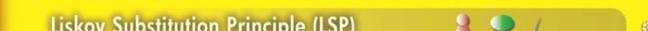
Interface Segregation Principle (ISP)



Dependency Inversion Principle (DIP)



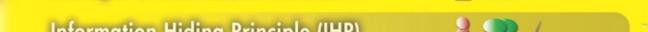
Liskov Substitution Principle (LSP)



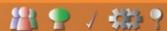
Principle of Least Astonishment



Information Hiding Principle (IHP)



Issue Tracking



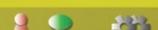
tests



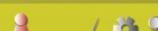
Automatisierte Unitests



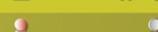
Mockups



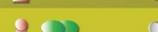
Code Coverage Analyse



Teilnahme an Fachveranstaltungen



Komplexe Refaktorisierungen



Continuous Integration (CI) I



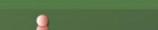
Statische Codeanalyse



Inversion of Control Container



Erfahrung weitergeben



Messen von Fehlern



# Teil 3: Praxistipps



# **Just do it!**

# Reviews. Reviews? Reviews!

- Mit Abstand am Effizientesten!
- Anfangs alleine oder in kleinen (2er-) Teams!
- Erst später mit mehr Teilnehmern (Clean Coder!)
- Wichtig: Offene, vertrauensvolle Atmosphäre!
  - Nicht als QS-Massnahme!



# Erfahrungen weitergeben

- Eigene Clean Code Erfahrungen und Erlebnisse weitergeben
- Beispiel: «Clean Code Snacks»
  - (sehr) kurze Präsentationen (5'–10')
  - z.B. unmittelbar vor oder nach der Znünipause

**T8: Hinweise von Coverage Patterns nutzen**

- Wenn ein Testfall scheitert: Codeabdeckung studieren!
  - Manchmal erkennt man aufgrund der Zeilen die (**nicht**) ausgeführt wurden sehr schnell den Fehler!
- **Failure** in JUnit-Test:

Punkt in Quadrant 2  
expected:<2> but was:<3>
- «Pattern» der Codeabdeckung:
  - **if**-Zweig wurde nicht ausgeführt!
  - Bedingung ist falsch formuliert!
- Vergleiche mit Beispiel zu T2: Nicht nur die grünen und roten Bereiche ansehen, sondern auch die zusätzlichen Informationen studieren und nutzen!

```
57. int quadrant = NO_QUADRANT;
58. if ((x != 0) && (y != 0)) {
59.     if (x > 0) {
60.         if (y > 0) {
61.             quadrant = QUADRANT_1;
62.         } else {
63.             quadrant = QUADRANT_4;
64.         }
65.     } else {
66.         if (y == 0) {
67.             quadrant = QUADRANT_2;
68.         } else {
69.             quadrant = QUADRANT_3;
70.         }
71.     }
72. }
73. return quadrant;
```

# Aufmerksamkeit für Clean Code wecken

- Clean Code Ringe tragen
  - Aufmerksamkeit und Interesse wecken
- Zettel mit einzelnen Themen aufhängen
  - Kühlschranktüre?
- Clean Code Buch
  - kaufen und verschenken
- Mausmatten
- Broschüre



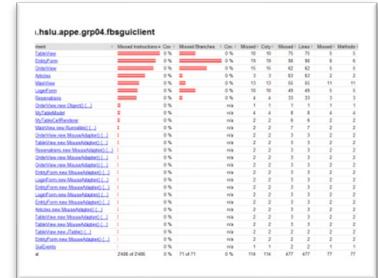
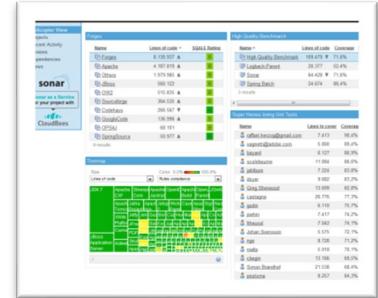
Optische Präsenz!

# Werkzeuge

- Checkstyle, PMD, Spotbugs
  - Namenslängen, Funktions- und Klassengrößen, Formatierung
  - Designprinzipien, Abhängigkeiten
  - Potentielle Programmierfehler
- SonarQube
  - Statistiken zur Verbesserung des Codes
  - Sensoren für 'gefährliche' Verletzungen
- Messung der Codeabdeckung
  - JaCoCo / EclEmma, JCobertura, Clover etc.
  - Vereinfachung der Fehlersuche!

signed in with <a href="#">Discourse</a> and <a href="#">API</a>			
File	Errors	Warnings	Info
408	0	0	0
...	...	...	...
2000	0	0	0

In 10 minutes I checked my GitHub repository for code quality and found 10 bugs!			
Severity	Error Summary	Count	Time
Info	No file or directory found for file	0	0s
Info	Nested brace import (prez song) [Layout]	4	0s
Info	JavaDoc Kommentar M&T	16	0s
Info	File contains tab character (this is the first instance)	17	0s
Info	JavaDoc Kommentar M&T	21	0s
Info	The Parameter name sofar is 'test' instead of 'sofar'	25	0s



# Schlusswort

clean code developer

- **Clean Code** und **Clean Code Developer**
  - Verinnerlichte Grundhaltung um **guten** Code zu schreiben!
- SW-Entwicklung als ein **SOLIDes Handwerk**
  - Wertschätzung und Qualitätsbewusstsein
- Pfadfinder-Regel!
  - Schon kleine Verbesserungen sind ein grosser Schritt hin zu besserer Software!



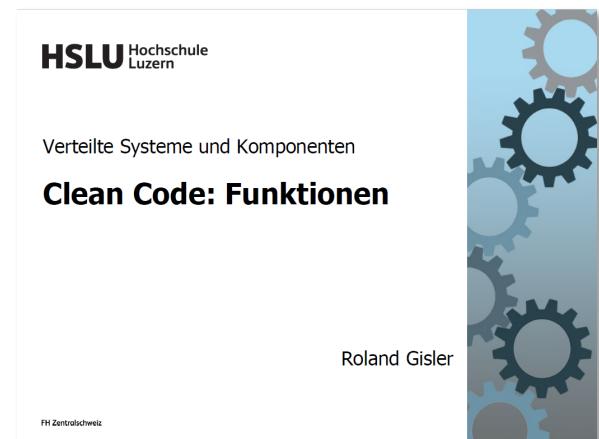
Just do it!

# Ergänzende Präsentationen zu Clean Code

- Wie implementieren wir gute (JUnit-)Tests nach Clean Code:  
**EP\_53\_CleanCodeUnitTests**

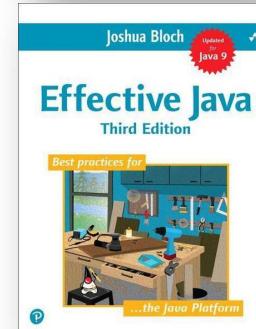
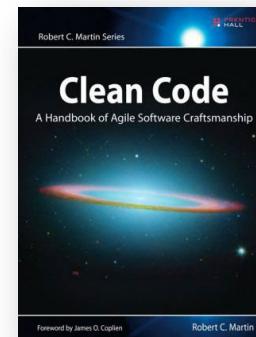


- Wie entwirft und implementiert man gute Funktionen (Methoden) nach Clean Code:  
**EP\_54\_CleanCodeFunctions**



# Literaturhinweise

- Robert C. Martin (Uncle Bob):  
<https://sites.google.com/site/unclebobconsultingllc/>
- Ralf Westphal & Stefan Lieser:  
<http://www.clean-code-developer.de>
- Robert C. Martin: **Clean Code**  
A Handbook of Agile Software Craftsmanship  
Prentice Hall, März 2009  
ISBN: 978-0-13-235088-4 (englisch)  
ISBN: 978-3-8266-5548-7 (deutsch)
- Joshua Bloch: **Effective Java**  
Best practices for the Java Plattform  
Third Edition, Juni 2008  
Pearson Addison-Wesley  
ISBN: 978-0-13-468599-1





**Besten Dank!**

# Quellen

- <https://sites.google.com/site/unclebobconsultingllc/>
- <http://www.frsweb.de/walz.htm>
- <http://www.lifehack.org/articles/productivity/how-to-do-an-ultimate-gtd-weekly-review-lifehack-lessons.html>

Verteilte Systeme und Komponenten

# **S.O.L.I.D.-Prinzipien**

**Fünf zentrale Designprinzipien**

Roland Gisler

# Inhalt

**S.O.L.I.D.** fasst **fünf** wichtige Designprinzipen zusammen:

- **S**ingle **R**esponsibility **P**rinciple (SRP)
- **O**pen **C**losed **P**rinciple (OCP)
- **L**iskov **S**ubstitution **P**rinciple (LSP)
- **I**nterface **S**egregation **P**rinciple (ISP)
- **D**ependency **I**nversion **P**rinciple (DIP)

# Lernziele

- Sie kennen die fünf grundlegenden S.O.L.I.D.-Designprinzipien.
- Sie können die Prinzipien anhand von Beispielen erklären.
- Sie können die Prinzipien in eigenen Entwürfen anwenden.

# Ziel der S.O.L.I.D.-Prinzipien

- Durch die Einhaltung der fünf fundamentalen **S.O.L.I.D.**-Prinzipien erreicht man ein qualitativ besseres und schöneres Design.
- Besseres Design heisst konkret:
  - Höhere Wiederverwendbarkeit
  - Leichtere Verständlichkeit / bessere Lesbarkeit
  - Verbesserte Testbarkeit
  - Vereinfachte Wartung
  - Verbesserte Erweiterbarkeit
  - Leichteres Refactoring

# **SRP**

## **Single Responsibility Principle**

# Single Responsibility Principle (SRP)

- Hauptziele
  - Eine Klasse\* soll nur **eine** Verantwortlichkeit haben.
  - Eine Klasse soll nur **einen** Grund zur Änderung haben.
- Einhaltung von SRP hat eine hohe Kohäsion zur Folge.
  - Es kommt und bleibt zusammen, was zusammen gehört.
- Wird SRP verletzt, ergibt sich umgekehrt eine hohe Kopplung.
  - Höhere Komplexität, schlechtere Wart- und Erweiterbarkeit.
- SRP gilt aber auch auf den Ebenen der Komponenten, Schichten, Teilsysteme: Unterschiedliche Abstraktionsebenen!

\* Beispiel für ein beliebiges Softwareartefakt, gilt sinngemäss auch für Modul, Package, Methode etc.

# **SRP ist eine fundamentale Grundlage von OOD**

- Das Single Responsibility Prinzip gilt als eines der fundamentalen Prinzipien des objektorientierten Designs.
  - Als Konzept relativ einfach, wird schnell verstanden.
- Einhaltung von SRP liefert im Design typisch viel **mehr** und dafür aber **kleinere** Klassen
  - Das ist deutlich besser als wenige, grosse Klassen!

## **Aber: SRP ist eines der am häufigsten verletzten Prinzipien!**

- Wir treffen sehr häufig auf Funktionen und Klassen, die (viel) zu viele Aufgaben auf einmal erfüllen wollen (bis hin zu «Gott»-Klassen).
- Wir nutzen Tools und Werkzeuge, die zu viel auf einmal machen (wollen) und eine sehr starke Abhängigkeit produzieren.

# Beispiel: Modem

- Ein altes Beispiel, in Anlehnung an Tom DeMarco, einem erklärten SRP-Anhänger (und Vorreiter von SA/SD):



```
interface Modem {  
    void dial(String phoneNumber);  
    void hangup();  
    void send(char data);  
    char receive();  
}
```



- Eine schmale und schlanke Schnittstelle!
- Ein schönes Beispiel für «gute» Objektorientierung!
- Ähm, oder vielleicht doch nicht?

# Modem – Beispiel

- Mögliche Gründe für eine Änderung sind:
  - Änderung des Wahlvorgangs:  
z.B. die automatische Wahlwiederholung wenn besetzt.
  - Änderungen in der Datenübertragung:  
z.B. die Vergrösserung des Datenbuffers.
- Das sind **zwei** Gründe!
- Das **Single Responsibility Principle** sagt aber:  
Es soll nur **einen** Grund für Änderungen geben!
- Und tatsächlich: Der Verbindungsauf- und -abbau hat (funktional) absolut **nichts** mit der Datenübertragung zu tun!
- Prinzip der **Single Responsibility Principle** ist hier somit **verletzt!**

# Modem – Lösung mit SRP

- Darum sollte man die Schnittstellen auftrennen:

```
interface Transmit {  
    void send(char character);  
    char receive();  
}
```



```
interface Connection {  
    void dial(String phoneNumber);  
    void hangup();  
}
```



- Schnittstellen werden schmäler, die Wiederverwendbarkeit höher.
- Wie ist das beim Logger-Projekt VSK: **Logger** und **LoggerSetup**

# SRP - Zusammenfassung

- Unix-Philosophie:  
«Tu nur ein Ding, genau ein Ding, das aber richtig!»
- Eine Klasse hat möglichst nur **eine** Zuständigkeit.
- Eine Klasse hat somit nur **einen** Grund zur Änderung.
- Änderungen oder Erweiterungen sollten sich auf möglichst wenig Klassen beschränken.
  - Die hohe Kohäsion bleibt erhalten.
- Viele kleine Klassen sind besser als wenige grosse Klassen.
- Wichtiger Nebeneffekt: Stark verbesserte Testbarkeit!

# OCP

## **O**pen **C**losed **P**rinciple

# Open Closed Principle

- Grundidee: Eine Klasse\* soll «offen» für Erweiterungen, aber «geschlossen» gegenüber Modifikationen sein.
- **Offen** für Erweiterungen:  
Design ist für eine einfache und sichere Erweiterbarkeit ausgelegt, beispielsweise durch Einsatz des Strategy-Pattern (GoF): Erweiterung durch einfaches Anlegen einer **neuen** Klasse.
- **Geschlossen** für Änderungen:  
Design ist so ausgelegt, dass **bestehende** Methoden und Klassen bei einer Erweiterung möglichst **nicht** verändert werden müssen.
- Motivation: **Reduktion des Risikos neue Fehler einzubauen.**

\* Beispiel für ein beliebiges Softwareartefakt, gilt sinngemäss auch für Modul, Package, Methode etc.

# Open Closed Principle – Beispiel

```
public double calc(Operation op, double arg1, double arg2) {  
    double result = 0.0;  
    switch (op) {  
        case Addition:  
            result = arg1 + arg2; break;  
        case Subtraktion:  
            result = arg1 - arg2; break;  
        default:  
            throw new IllegalArgumentException(); break;  
    }  
    return result;  
}
```



- Was passiert, wenn eine neue Operation ergänzt werden soll?  
→ Es besteht ein hohes Risiko bei der Erweiterung einen Fehler einzubauen. Klassiker: Das **break** zu vergessen.

## Bessere Lösung

- Auslagern der erweiterbaren Funktionen in →Strategien, anstatt die bestehende Funktion zu verändern.

```
interface Operation {  
    double calc(double arg1, double arg2);  
}
```



```
double calc(Operation op, double arg1, double arg2) {  
    return op.calc(arg1, arg2);  
}
```

- Erweiterung durch neue Strategien, welche das Interface implementieren.
  - Die Operation ist nun ein Interface.
  - Das ursprüngliche **switch**–Statement entfällt vollständig!

# OCP - Zusammenfassung

- Eine Software-Entität soll offen für Erweiterungen, aber geschlossen gegenüber Modifikationen sein.
- Mit OCP **senkt** man das **Risiko** neue Fehler einzubauen, weil man bestehenden Code **nicht** (oder weniger) ändern muss.
- Häufig über Einsatz des Strategy-Patterns (GoF) erreicht.
  - Eliminiert oder reduziert die **if/switch**-Statements:  
<http://www.industriallogic.com/xp/refactoring/conditionalWithStrategy.html>
- OCP ist ein sehr wirkungsvolles, aber anspruchsvolles Prinzip!

# LSP

## **Liskov Substitution Principle**

# Liskov Substitution Principle (LSP)

- Liskov'sches Substitutionsprinzip:  
«Eigenschaften, die anhand der Spezifikation des vermeintlichen Typs eines Objektes bewiesen werden können, sollen auch dann gelten, wenn das Objekt einer Spezialisierung dieses Typs angehört.»
- Etwas einfacher formuliert:  
«Subtypen sollten sich so verhalten wie ihre Basistypen.»
- Noch etwas einfacher formuliert:  
In spezialisierten Klassen nur Methoden ergänzen oder für Erweiterung überschreiben, aber **nie** fundamental verändern!

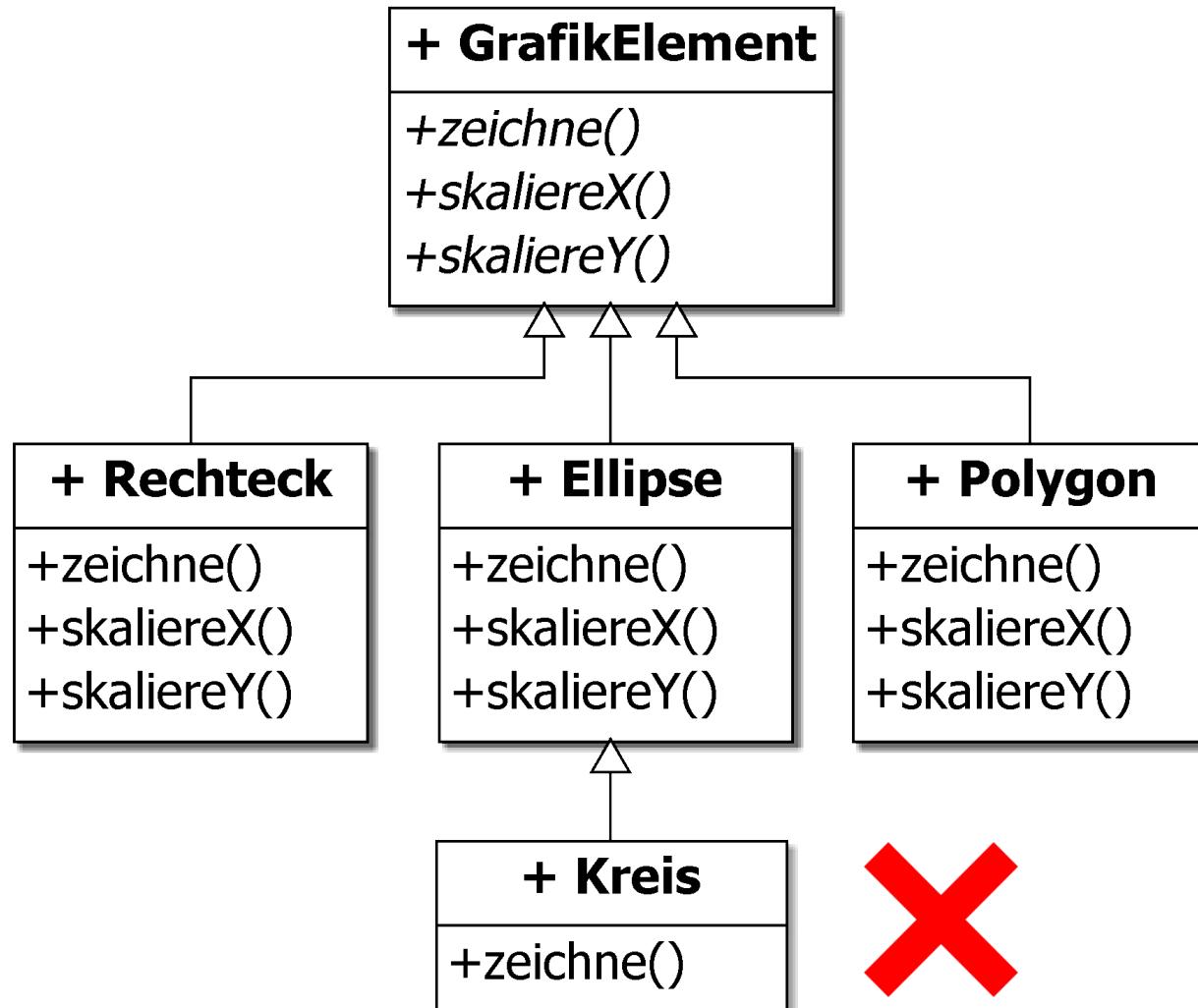
# **Barbara Jane Huberman Liskov**

- Barbara Jane Huberman Liskov
- Professorin für Elektrotechnik und Informatik am MIT.
- 1968 erhielt sie an der Stanford University als erste Frau in den USA den Titel eines Ph.D. in Informatik.
- 2008 erhielt sie als erst zweite Frau (nach Frances E. Allen) den Turing Award.
- Gemeinsam mit Jeannette Wing entwickelte sie 1993 das für die OOP bedeutsame Liskov'sche Substitutionsprinzip.



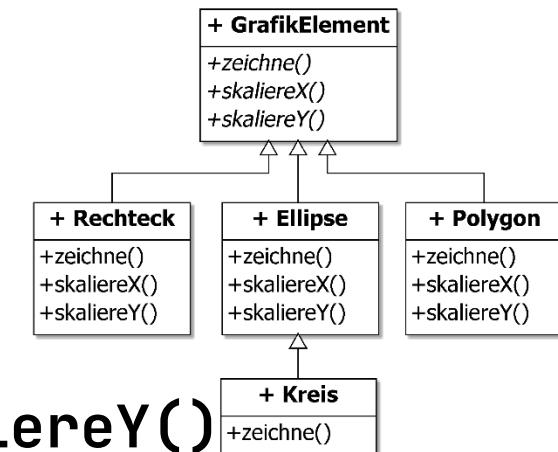
# LSP-Klassiker: Kreis-Ellipse Problem

- Annahme: Die `skaliere*`-Methoden werden später ergänzt.
  - Wo liegt das Problem?



# Kreis-Ellipse Problem

- Kreis scheint eine Spezialisierung der Ellipse mit der zusätzlichen Bedingung  $r_x=r_y$  zu sein.
- Die Ergänzung von `skaliereX()` und `skaliereY()` ist weder für Rechteck, Ellipse noch Polygon ein Problem.  
→ Aber was ist mit dem Kreis?
- Der Kreis erbt alle Methoden von der Ellipse. Bei einem Kreis dürfen  $r_x$  und  $r_y$  aber nicht mehr unabhängig voneinander skaliert werden!
- Die Anforderung: «Die Achsen können unabhängig voneinander skaliert werden.» stimmt für den Kreis nicht mehr!  
→ **Das Liskov'sche Substitutionsprinzip ist hier verletzt!**



# Das hatten wir doch auch schon mal!

+ Punkt
-x : int
-y : int
+Punkt(x,y) : Punkt
+getX() : int
+getY() : int
+equals(o : Object) : boolean
+hashCode() : int

+ FarbPunkt
-farbe : Color
+FarbPunkt(x,y,farbe) : FarbPunkt
+getFarbe() : Color
+equals(o : Object) : boolean
+hashCode() : int

- Problem bei der Einhaltung des **equals**-Contracts bei der Spezialisierung nach **FarbPunkt**.
- Anforderung der **Symmetrie**: Wenn **punkt.equals(farbpunkt)** dann ist auch **farbpunkt.equals(punkt)**.
- Verhält sich ein **farbiger** Punkt genauso wie ein normaler Punkt? **Nein**, weil er berücksichtigt eventuell noch die Farbe in seinem Verhalten!
- **LSP ist hier meistens auch verletzt!**
  - Letztlich Abhängig vom Kontext!

# LSP - Zusammenfassung

- Den Sinn von Vererbung / Spezialisierung immer kritisch verifizieren: Subtypen sollten sich so verhalten wie ihre Basistypen.
- Verifizierte Entscheide mit folgenden Sätzen:
  - Subtyp ist ein (**is-a**) Basistyp.
  - Subtyp verhält sich (**behaves-as**) wie ein Basistyp.
- **FCoI:** Meistens ist die Komposition der Vererbung vorzuziehen!
- Macht die Implementation von **equals()** Schwierigkeiten, sollte man unbedingt die Vererbung hinterfragen!
- Tipp: Vererbung konsequent verhindern (**final**)!
  - Nur wo sinnvoll und vorgesehen ein explizites Design für Spezialisierungen (Design for inheritance or else prohibit it).

# **ISP**

## **I**nterface **S**egregation **P**rinciple

# Interface Segregation Principle

- Clean Code: Interface Segregation Principle (ISP)
  - Artikel von Uncle Bob (Robert C. Martin, 1996):  
<http://www.objectmentor.com/resources/articles/isp.pdf>
  - Segregation: «Entmischung» → Trennung
- Schnittstellen strikt von Details der Implementation trennen.
- Schnittstellen sollten sauber voneinander getrennt sein.
  - Keine Überschneidungen.
  - Keine Population von Schnittstellen.
  - Keine «fat»-Schnittstellen.
- Eine Schnittstelle soll eine hohe Kohäsion aufweisen.
  - Nur Methoden, die wirklich zusammen gehören.
- Nebenbei: Konzept des «design by interface» hilft hier.

# Interface Segregation Principle

- Klienten sollten nur von Schnittstellen abhängig sein, die sie wirklich brauchen.
- Gibt es verschiedenartige Klienten eines Systems, sollte jeder Typ von Klient seine eigene Schnittstelle haben.
  - Abhängigkeiten minimieren → Kopplung minimiert.
  - Hat auch positiven Einfluss auf die Sicherheit.
- Basisklassen sollten nichts von ihren Spezialisierungen wissen.
  - Abstrakte Basisklassen als Schnittstellen.
- Schnittstellen feingranular entwerfen.
  - Viele kleine Schnittstellen sind besser als wenige grosse Schnittstellen.

# Refactorings für ISP

Refactorings zur Erreichung der Interface Segregation:

- Superklasse aus bestehender Klasse extrahieren.
  - Superklasse ist dann häufig abstrakt (Schnittstelle).
  - Gefahr der (unerwünschten) Population der Schnittstelle!
- Interface aus bestehender Klasse extrahieren.
  - Geringere Kopplung, vgl. Komposition statt Vererbung (FCoI).
  - Unabhängigkeit von Implementation.
- Bestehende Interfaces auftrennen.
  - Schmalere Schnittstellen, geringere Kopplung.
    - vgl. Separation of Concerns (**SoC**)
    - vgl. **SRP** – Beispiel: Interface für ein Modem)

# ISP - Zusammenfassung

- Schnittstelle strikt von Details der Implementation trennen.
  - Ist mit Java **einfach**: Weil wir haben Interfaces!
- Schnittstellen sollen eine hohe Kohäsion haben.
- Kopplung zwischen Komponenten soll minimal sein.
- Viele kleine (schmale) Schnittstellen sind besser als eine zu grosse (fette) Schnittstelle.
- Population von Schnittstellen vermeiden
  - möglichst **keine** Vererbung von Schnittstellen!

# DIP

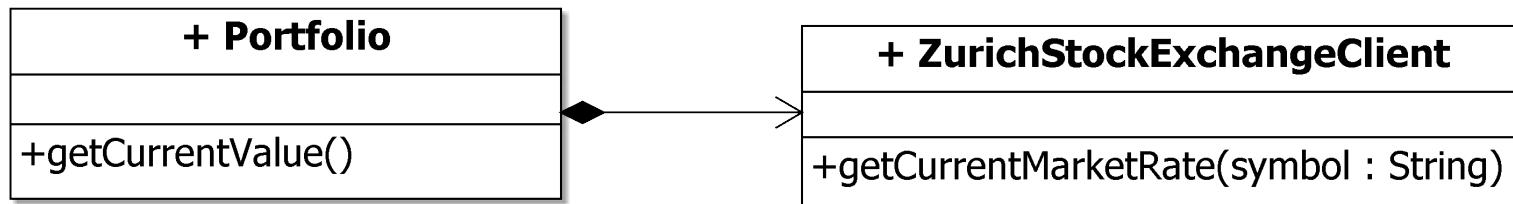
## Dependency Inversion Principle

# Dependency Inversion Principle (DIP)

- Ziel: Änderungen isolieren.
  - Artikel von Uncle Bob (Robert C. Martin, 1996):  
<http://www.objectmentor.com/resources/articles/dip.pdf>
- High-Level Klassen sollen nicht von Low-Level Klassen abhängig sein, sondern allenfalls beide von Interfaces.
  - High-Level: Hoher Abstraktionsgrad, Konzeptionell.
  - Low-Level: Konkrete, detailbehaftete Implementation.
  - Siehe auch **Single Level of Abstraction (SLA)**
- Analog: Interfaces sollen nie von Details der Implementation abhängig sein, sondern allenfalls Implementationen von Interfaces.

# Änderungen isolieren

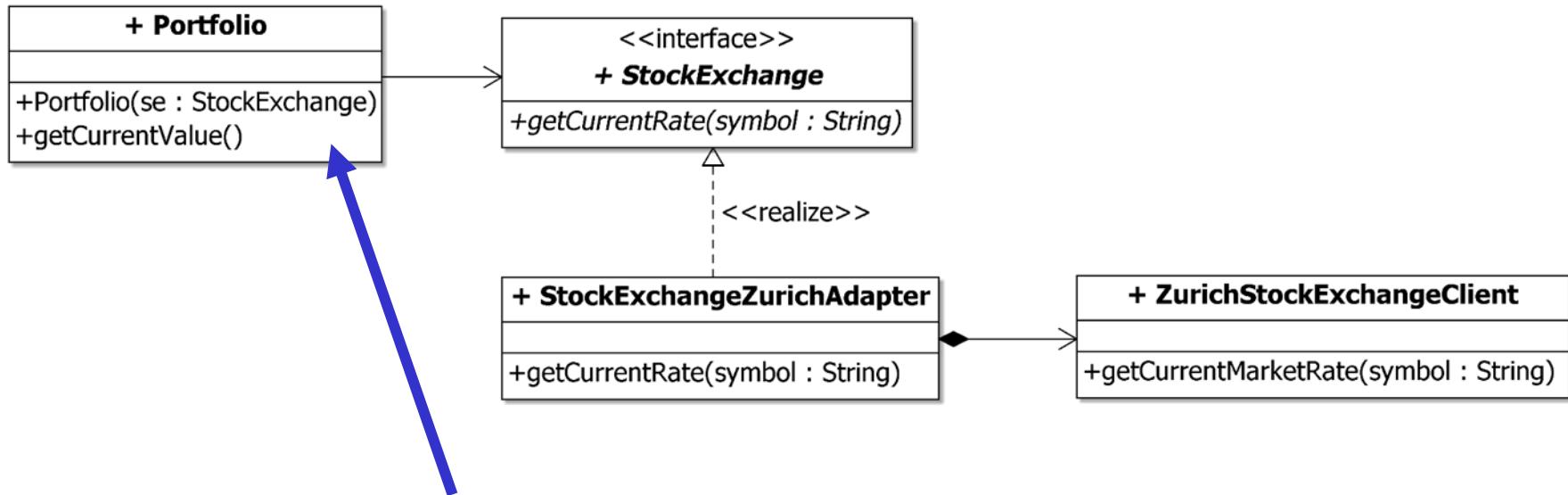
- Abstrakte Klassen und/oder Interfaces einführen, um die Auswirkungen von Implementationsdetails zu minimieren.
- Schlechtes Beispiel:
  - **Portfolio** und **ZurichStockExchangeClient**



- High-Level-Klasse **Portfolio** ist abhängig von Low-Level-Klasse **ZurichStockExchangeClient** → **Schlecht!**

# Änderungen isolieren

- Bessere Lösung: **Portfolio** unabhängig von Implementationsdetails (**ZurichStockExchangeClient**)



- Nebenbei mit Dependency Injection (DI): Konstruktor mit **StockExchange** erlaubt einfachere Testbarkeit von **Portfolio**. (z.B. durch den Einsatz von →Test Doubles).
- Nebenbei: Das entspricht dem **Adapter**-Pattern nach **GoF**!

# DIP - Zusammenfassung

- High-Level Klassen sollen nicht von Low-Level Klassen abhängig sein, sondern beide von Interfaces.
- Interfaces sollen nicht von Details abhängig sein, sondern Details von Interfaces.
- Isolation von Klassen vereinfacht/ermöglicht die Testbarkeit, ggf. auch mit Einsatz von Test Doubles.
- Auflösung von Dependencies über **Dependency Injection (DI)**.

# S.O.L.I.D.

Schreiben Sie **SOLIDen** Code!

# Vermeintlicher Angriff auf SOLID: C.U.P.I.D. - Prinzipien

- Im Jahr 2021 wurde von Daniel Terhorst-North eine 2017 entstandene «Provokation» unter dem Titel «Why every single element von SOLID is wrong» veröffentlicht.
- **C.U.P.I.D.** ist eine gute Ergänzung zu S.O.L.I.D. – aber kein Ersatz:  
**EP\_52\_CupidPrinzipien**  
(optionale Ergänzung)

**HSLU** Hochschule  
Luzern

Verteilte Systeme und Komponenten

## C.U.P.I.D.-Eigenschaften

Keine Konkurrenz, sondern eine sinnvolle  
Ergänzung zu den S.O.L.I.D.-Prinzipien

Roland Gisler



# **Zusammenfassung – S.O.L.I.D.**

- **Single Responsibility Principle (SRP)**
  - Spezialisierung von Separation of Concerns (SOC).
  - Ein Element soll nur einen Grund für Änderungen haben.
- **Open Closed Principle (OCP)**
  - Ein Element soll offen für Erweiterungen sein, aber geschlossen gegen Modifikationen.
  - Neue Funktionalitäten können ergänzt werden, ohne dass bestehender Code geändert werden muss.
- **Liskov Substitution Principle (LSP)**
  - Eine Spezialisierung verhält sich immer wie sein Basistyp.
  - Kann somit jederzeit den Platz des Basistyps einnehmen.

# **Zusammenfassung – S.O.L.I.D.**

- **Interface Segregation Principle (ISP)**
  - Clients sollen nicht mit Details belastet werden, die sie nicht benötigen.
  - Bewirkt eine lose Kopplung und eine hohe Kohäsion.
- **Dependency Inversion Principle (DIP)**
  - Highlevel Klassen sollen nicht von Lowlevel Klassen abhängig sein, sondern beide von Interfaces.
  - Interfaces sollen nicht von Details abhängig sein, sondern Details von Interfaces.

**Fragen?**

Verteilte Systeme und Komponenten

# **C.U.P.I.D.-Eigenschaften**

**Keine Konkurrenz, sondern eine sinnvolle  
Ergänzung zu den S.O.L.I.D.-Prinzipien**

Roland Gisler

# Inhalt

**C.U.P.I.D.** fasst **fünf** wichtige Designprinzipen zusammen:

- **C**omposable – lässt sich gut mit anderem nutzen.
- **U**nix philosophy – kümmert sich genau um eine Sache.
- **P**redictable – macht das, was man erwartet.
- **I**diomatic – Nutzung fühlt sich «natürlich» an.
- **D**omain-based – sowohl in Sprache als auch Struktur.

# Lernziele

- Sie kennen die fünf grundlegenden C.U.P.I.D.-Designprinzipien.
- Sie können die Prinzipien anhand von Beispielen erklären.
- Sie können die Prinzipien in eigenen Entwürfen anwenden.

# Idee hinter den C.U.P.I.D.-Eigenschaften

- Entwickelt als Ergänzung zu S.O.L.I.D. von Daniel Terhorst-North und 2021 veröffentlicht.
  - Seine «Provokation» von 2017 wurde oft missverstanden:  
«Why every single element of SOLID is wrong».
- Motivation für C.U.P.I.D.:  
Nicht strenge Regeln und Richtlinien (die nicht eingehalten werden) definieren, sondern positive Eigenschaften, die mehr oder weniger erfüllt werden können.
- Wichtig: C.U.P.I.D. ist kein Ersatz oder Gegenplot zu S.O.L.I.D., sondern eine sinnvolle Ergänzung!

# Composable

Plays well with others.

## **Composable** - lässt sich gut mit anderem nutzen

- Guter Code hat eine kleine «Oberfläche»: Von Aussen sind nur jene Teile sichtbar, welche für die Entwickler\*innen relevant sind.
  - (Wieder-)Verwendung des Codes möglichst einfach halten.
  - Es kann weniger schiefgehen und es entstehen weniger Konflikte, etwa bei Änderungen und Verschiebungen.
  - Hohe Kohäsion, schmale und einfache Schnittstelle.
- Mit eindeutig definierten Schnittstellen ist er einfacher zu verstehen und leichter zu verwenden.
  - Die Intention des Codes soll offensichtlich sein.

# Unix Philosophy

Does one thing well.

# **Unix philosophy** - kümmert sich genau um eine Sache

- Guter Code sorgt dafür, dass eine Softwareeinheit (Programm, Modul, Klasse oder Methode) „nur“ genau eine Aufgabe erledigt, diese dafür aber richtig gut.
- Klassisches UNIX-Konzept: Viele, kleine Tools, die sich vielseitig kombinieren lassen.
- Erinnert das zufällig an SRP (Single Responsibility Principle)?
  - Exakt das ist gemeint!

# Predictable

Does what you expect.

## **Predictable** - macht das, was man erwartet

- Der Code verhält sich wie erwartet, ohne unliebsame, unerwartete Überraschungen.
  - Fachlicher Formuliert: Keine Nebeneffekte.
- Der Code ist deterministisch in der Ausführung und er ist im technischen Sinne beobachtbar (keine Magie).
  - Vereinfacht sowohl das Debugging als auch die Fehlersuche.
- Der interne Status einer Softwareeinheit kann von den Ausgaben zwar abgeleitet werden, ist aber nicht veränderbar.
  - Datenkapselung

# Idiomatic

Feels natural.

## **Idiomatic** - Nutzung fühlt sich «natürlich» an

- Wenn der Code gut ist, dann fühlt sich dessen Nutzung und die Bearbeitung des Codes «natürlich» an.
  - Leicht verständlich.
- Es werden die Idiome des Kontextes (Domäne, Firma, Team) sowie des Ökosystems der Sprache, in der er geschrieben ist, verwendet.
  - Namenskonventionen einhalten!

# Domain-based

In language and structure.

## **Domain-based** - sowohl in Sprache als auch Struktur

- Für guten Code wird die Domänensprache und deren Struktur verwendet.
- Die Struktur des Codes soll die Lösung widerspiegeln, und nicht das darunterliegende Framework und/oder die eingesetzten Konzepte.
  - Etwas was in vielen Libraries/Frameworks verletzt wird!
- Anstatt technischer Begriffe/Konzepte (Models, Views, Controllers) sollen Domänenbegriffe (Dokumente, Zahlungen) verwendet werden.
  - Die Domänengrenzen sollen als Modulgrenzen und Deployment-Units beachtet werden.
  - Architektur: Vertikale vs. horizontale Trennung.

# C.U.P.I.D.

Schreiben Sie **liebevollen\*** Code!

\*engl. «cupid» = amor, Liebesgott in der römischen Mythologie

# Zusammenfassung – C.U.P.I.D.

- Wird eher durch «positive, bewertbare Eigenschaften» denn als (vermeintlich) «harte Regeln» definiert.
  - Somit für viele Entwickler\*innen angenehmer im Umgang?
  - «Kuschel»-Kommunikation?
- Letztlich verfolgt C.U.P.I.D ziemlich exakt die gleichen Ziele wie S.O.L.I.D. – einfach ein bisschen moderner und zeit(geist)gemässer ausformuliert.
- Gute Ergänzung zu S.O.L.I.D. – welche noch immer Gültigkeit hat und absolut unbestritten ist.
  - C.U.P.I.D. ist **kein** Angriff auf S.O.L.I.D.!

**Fragen?**

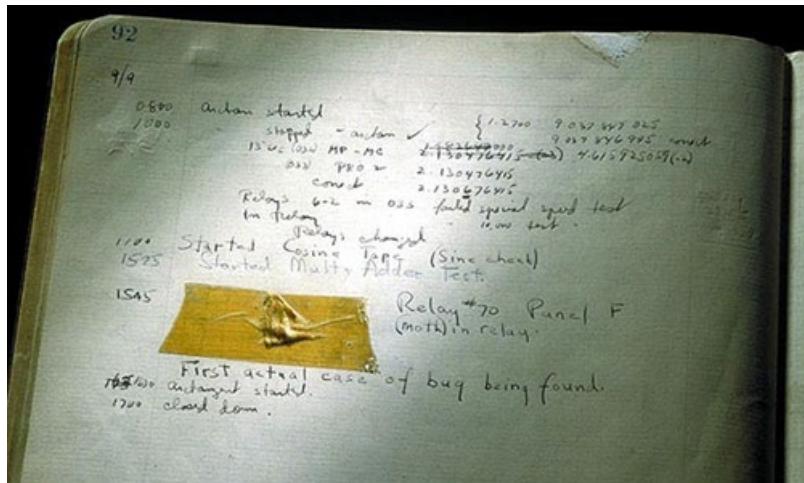
Verteilte Systeme und Komponenten

# **Clean Code: Unit Tests**

Roland Gisler

# Unit-Tests nach Clean Code

- Clean Code: Kapitel 9 – Unit-Tests; Heuristiken T1 – T9
- Unit-Tests sind bei den meisten Entwickler\*innen gut etabliert, es gibt aber noch viel Potential! (siehe [Test-First](#))
- Qualität des Testcodes ist wichtig!
  - Testcode nicht als «Wegwerfcode» behandeln!
- Dank Unit-Tests finden wir Bugs schneller und früher!



4. April 1945 um 15:15 im Naval Weapons Center, Virginia:

Admiral Grace Murray Hopper protokolliert im Tagesjournal den ersten Bug in einem Programm das auf dem Harvard Mark II Aiken Relay Calculator life (so die Legende).



[https://de.wikipedia.org/wiki/Grace\\_Hopper](https://de.wikipedia.org/wiki/Grace_Hopper)

# Motivation

- Gute und umfassende Tests (egal welcher Art) sind die fundamentale Basis für alle weiteren Bemühungen zur Verbesserung der (Code-)Qualität.
  - Qualität in Form von :  
Reliability, Changeability, Efficiency, Security,  
Maintainability, Portability, Reusability etc.
- Gute Unit Tests sind die erste, schnellste, einfachste Teststufe.
  - Schnelles erstes Feedback «ob es funktioniert».
  - Regression, Basis für jedes Refactoring.
- Wichtig für Continuous Integration (siehe CI-Input).

# Was ist ein Unit Test?

- Nicht alles was JUnit verwendet ist auch ein Unit Test!
  - JUnit kann zur Automatisation von beliebigen Testarten verwendet werden!
- Definition eines Unit Tests von Roy Osherove (Author von «The Art of Unit Testing»):

«A unit test is an automated piece of code that invokes a unit of work in the system and then checks a single assumption about the behavior of that unit of work.»

# Test Driven Development (TDD)

Die drei Gesetze des TDD:

- §1:** Produktiver Code darf erst implementiert werden, wenn es dafür einen Unit-Test gibt.
- §2:** Dieser Unit-Test darf nur gerade so viel Code enthalten, dass er fehlerfrei kompiliert, aber als Test scheitert.
- §3:** Man ergänzt jeweils nur gerade so viel produktiven Code, dass der scheiternde Test besteht.

- Der Zyklus dieses Ablaufes liegt dabei im Bereich von **Sekunden bis Minuten!**
  - Tests und Produktivcode werden praktisch zeitgleich geschrieben; Tests eilen nur wenig voraus.

# Tests sauber halten!

- Für Testcode sollen die identischen Qualitätsstandards gelten wie für produktiven Code!
  - Gute Namensgebung, gute Struktur, verständlich → CCD!
- Wer denkt: «Besser schmutzige Tests als gar keine Tests!», der ist auf dem Holzweg!
- Begründung: Testcode lebt länger als produktiver Code!
  - Produktiver Code wird z.B. refactored!
  - Wenn Testcode erodiert, dann haben wir doppelt verloren!
- Testcode «dokumentiert» den produktiven Code.
  - Muss umso mehr wart- und erweiterbar sein!
- Es ist ein weiter Weg von Unit Tests zu **guten** Unit Tests!

# Was ist ein sauberer Unit Test?

- Drei Dinge machen einen sauberen Unit Test aus:
  1. Lesbarkeit durch **Klarheit**.
  2. Lesbarkeit durch **Einfachheit**.
  3. Lesbarkeit durch **Ausdrucksdichte**.
- Lesbarkeit (Klarheit, Einfachheit und Ausdrucksdichte) ist bei Testcode noch wichtiger als bei Produktivcode.
  - Mit möglichst wenig Code möglichst viel aussagen.
- Jeder Testfall nutzt das «**Build-Operate-Check**»-Pattern:
  1. Erstellen der Testdaten (**Build**).
  2. Manipulieren der Testdaten (**Operate**).
  3. Verifikation der Ergebnisse (**Check**).
- Vergleiche auch: «**Triple A**»-Pattern: **Arrange**, **Act**, **Assert**.

# Domänenspezifische Testsprache (DS[T]L)

- Nicht so schlimm wie es klingt, wir bleiben bei der Sprache unserer Wahl!
- Aber: Wir schreiben uns ein domänenspezifisches Set von (typisch statischen) Utility-Methoden, welche...
  - den Testcode kompakter und aussagekräftiger machen.
  - ihrerseits natürlich die API des Testkandidaten verwenden.
- Beispiel für eigene **assert**-Methoden:

**assertResponseIsXML()**

**assertResponseContainsElement(...)**



- Identischer Ansatz wie bei AssertJ- oder Hamcrest-Library:  
Möglichst aussagekräftige und leichtverständliche Ausdrücke.

# Nur ein assert pro Test

- Eine wirklich sehr drakonische Forderung!
  - Nicht immer sinnvoll umsetzbar, aber wenn: Viel besser!
- Weniger (bzw. keine) **assert**-Messages notwendig, weil die Testfälle selber schon sehr selektiv sind.
  - Einzelne Testmethoden werden überschaubarer und kleiner.

```
...  
assertEquals(soll, ist, "Person verglei...");  
...
```



```
@Test  
testPersonEquals() {  
    assertEquals(soll, ist);  
}
```



# Nur ein Konzept pro Test

- Auch bei Tests soll man SLA, SRP und SOC einhalten!
    - **Single Level of Abstraction (SLA).**
    - **Single Responsibility Principle (SRP).**
    - **Separation Of Concerns (SoC).**
  - Ein untrügliches Zeichen für Verletzung:  
Eine Testmethode wird in mehrere Abschnitte gegliedert, vielleicht sogar noch mit Kommentarblöcken unterteilt.
  - Zu grosse und lange Testmethoden sind mühsam!
    - Viel weniger selektive **failure**-Meldungen.
    - Weniger gezielte Wiederholung des Tests möglich.
- Kurz: Alle Nachteile von **zu grossen** Methoden!

# F.I.R.S.T. – Prinzip

- **Fast** – Tests sollen schnell sein, damit man sie jederzeit und regelmässig ausführt.
- **Independent** – Tests sollen voneinander unabhängig sein, damit sie in beliebiger Reihenfolge und einzeln ausgeführt werden können.
- **Repeatable** – Tests sollten in/auf jeder Umgebung lauffähig sein, egal wo und wann.
- **Self-Validating** – Tests sollen mit einem einfachen boolschen Resultat zeigen ob sie ok sind oder nicht\*.
- **Timely** – Tests sollten rechtzeitig, d.h. vor dem produktiven Code geschrieben werden → [Test-First](#), TDD.

\* Beim Einsatz zeitgemässer Unit-Test Frameworks selbstverständlich

# Uncle Bob's Unit-Tests Heuristiken

- **T1:** Unzureichende Tests.
- **T2:** Coverage-Werkzeug verwenden.
- **T3:** Triviale Tests nicht überspringen.
- **T4:** Ignorierte Tests zeigen Mehrdeutigkeit auf.
- **T5:** Grenzbedingungen testen.
- **T6:** Bei Fehlern die Nachbarschaft gründliche testen.
- **T7:** Muster des Scheiterns zur Diagnose nutzen.
- **T8:** Hinweise durch Coverage Patterns beachten.
- **T9:** Tests sollen schnell sein.

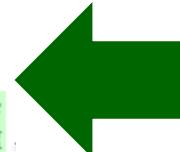
# T1: Unzureichende Tests vermeiden

- Meistens wird nur bis «zum Gefühl dass es reicht» getestet.
- Clean Code fordert:  
Es wird alles getestet, was schief gehen kann!
  - Nach Murphy's Law geht ja auch alles schief...
- Man schreibt so lange Tests...
  - wie es Bedingungen gibt, die noch nicht geprüft werden.
  - Berechnungen stattfinden, die nicht validiert werden.
- Faktisch bedeutet das eine 100%-ige Testabdeckung!
  - Ist das eine realistische Forderung? Nein, nicht überall!
- Aber sicher ein gutes Ziel um Fortschritte zu machen... ☺

## T2: Coverage-Werkzeug verwenden

- Coverage-Werkzeuge decken Lücken in den Tests auf.
  - Machen es (sehr) leicht, ungetesteten Code aufzudecken.
- Statement- und Decision-Coverage beachten!
- Beispiel:

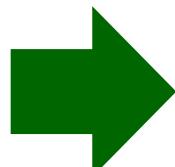
```
57.         int quadrant = NO_QUADRANT;
58.     if ((x != 0) && (y != 0)) {
59.         if (x > 0) {
60.             if (y > 0) {
61.                 quadrant = QUADRANT_1;
62.             } else {
63.                 quadrant = QUADRANT_4;
64.             }
65.         } else {
66.             if (y > 0) {
67.                 quadrant = QUADRANT_2;
68.             } else {
69.                 quadrant = QUADRANT_3;
70.             }
71.         }
72.     }
73.     return quadrant;
```



### Testfälle:

```
assertEquals(1, Geometrie.getQuadrant( 1,  1));
assertEquals(2, Geometrie.getQuadrant(-1,  1));
assertEquals(3, Geometrie.getQuadrant(-1, -1));
assertEquals(4, Geometrie.getQuadrant( 1, -1)); // B1

assertEquals(0, Geometrie.getQuadrant( 0,  1)); // B2
assertEquals(0, Geometrie.getQuadrant( 1,  0)); // B3
assertEquals(0, Geometrie.getQuadrant( 0,  0)); // B4
```



## T3: Triviale Tests nicht überspringen

- Manchmal gibt es Tests, die man nicht macht, weil man der Meinung ist, sie seien zu trivial.
  - Setter/Getter, Konstruktoren, einfache Berechnungen...
- Clean Code sagt:  
Dokumentarischer Wert der Testfälle übersteigt die Produktionskosten → es lohnt sich **doch!**
  - Sie vergrössern die Abdeckung.
  - Sind ein Motivationsfaktor, weil einfach zu schreiben.

«Keine Klasse, keine Funktion ist zu klein,  
um nicht automatisch getestet zu sein!»

## T4: Deaktivierte Tests zeigen Mehrdeutigkeit auf

- Test auskommentiert oder (besser) deaktiviert:

```
@Disabled("Gibt es eigentlich einen Nullpunkt?")
@Test
void testGetQuadrantNullpunkt() {
    assertEquals(0, Geometrie.getQuadrant(0, 0));
}
```

- Wenn ein Testfall deaktiviert ist, deutet das häufig auf eine Unklarheit in den Anforderungen hin!
- Deaktivierte Testfälle sind somit ein **Warnsignal!**
  - Temporär ok, aber **nie** als «Providurium».
- Hinweis: Vor JUnit 5 hiess die Annotation **@Ignore(...)**

# T5: Grenzbedingungen testen

- Testen mit Grenzwerten ist wichtig!
  - Meist wird «die Mitte» eines Algorithmus richtig implementiert, aber seine Grenzen falsch beurteilt.
- Beispiel:

```
public long addition(int sum1, int sum2) {  
    long result = (long) sum1 + sum2;  
    System.out.println("Addition ergibt: " + result);  
    return result;  
}
```

- Und der Testfall: PASSED  

```
assertEquals(2L * Integer.MAX_VALUE,  
            t.addition(Integer.MAX_VALUE, Integer.MAX_VALUE));
```
- Konsolenausgabe: Addition ergibt: 4294967294

## T6: Fehler-Nachbarschaft gründlich testen

- Fehler treten oftmals gehäuft auf!
- Findet man in einer Klasse / Funktion einen Fehler sollte man diese erschöpfend testen.
  - Es besteht eine hohe Wahrscheinlichkeit, dass darin noch weitere Fehler gefunden werden.

*Murphy's Law sagt:*

- Meist ist ein kleiner Fehler nur dazu da, dass sich dahinter ein viel grösserer Fehler verstecken kann.
- Findet man hingegen einen grossen Fehler, wird sich dahinter ein anderer grosser Fehler verstecken.

## T7: Muster des Scheiterns zur Diagnose nutzen

- Wenn man genügend (und mit hoher Codeabdeckung) testet, kann man in scheiternden Tests manchmal Muster erkennen!
- Beispiele:
  - Alle Tests scheitern, bei welchen ein String eine bestimmte Länge überschreitet.
  - Alle Tests scheitern, bei welchen ein bestimmtes Argument negative Werte erhält.
- Es soll schon Fälle gegeben haben, wo das deutliche **rot/grün**-Muster im Testreport zu «**Aha**»-Erlebnissen geführt hat!

# T8: Hinweise von Coverage Patterns nutzen

- Wenn ein Testfall scheitert: Codeabdeckung studieren!
  - Manchmal erkennt man aufgrund der Zeilen die (**nicht**) ausgeführt wurden sehr schnell den Fehler!
- Beispiel: **Failure** in JUnit-Test:

Punkt in Quadrant 2  
expected:<2> but was:<3>

- «Pattern» der Codeabdeckung:
  - **if-Zweig wurde nicht ausgeführt!**
  - **Bedingung ist falsch formuliert!**
- Vergleiche mit Beispiel zu T2: Nicht nur die grünen und roten Bereiche ansehen, sondern auch die zusätzlichen Informationen studieren und nutzen!

```
57.     int quadrant = NO_QUADRANT;
58.     if ((x != 0) && (y != 0)) {
59.         if (x > 0) {
60.             if (y > 0) {
61.                 quadrant = QUADRANT_1;
62.             } else {
63.                 quadrant = QUADRANT_4;
64.             }
65.         } else {
66.             if (y == 0) {
67.                 quadrant = QUADRANT_2;
68.             } else {
69.                 quadrant = QUADRANT_3;
70.             }
71.         }
72.     }
73.     return quadrant;
```

## T9: Tests sollen schnell sein

- Langsame Test werden selten oder gar nicht ausgeführt.
  - Ein Test, der nicht ausgeführt wird, ist nichts Wert.
- Unit-Tests sollten schnell sein, damit man sie immer und jederzeit ausführt.
  - Darum sollten die Tests auch nicht zu gross sein.
- Man soll alles Erforderliche tun, um die Tests zu beschleunigen!
  - *Need for speed!*



# Zusammenfassung – 1/2

- **Qualität** des Testcodes ist **noch wichtiger** als diejenige vom produktivem Code!
  - Nur dank Testcode getrauen wir uns etwas im produktiven Code zu verändern.
  - Testcode unbedingt «sauber» halten.
- «Build-Operate-Check» - Pattern einhalten.
- Domänen spezifische Hilfsmethoden ergänzen, um die Lesbarkeit zu erhöhen.
- Wenn möglich nur ein **assert** pro Testfall.
- Nur ein Konzept (SoC) pro Testfall.
- F.I.R.S.T. – Prinzip!

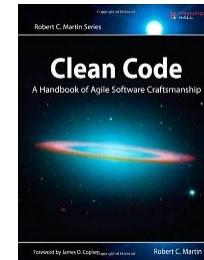
## Zusammenfassung – 2/2

- Wir scheiben viele, kleine, selektive und auch triviale Tests, welche wir schnell, jederzeit und überall ausführen können!
- Wir verwenden Coverage-Werkzeuge um die Testabdeckung zu prüfen und laufend zu erhöhen.
  - Liefern auch bei Test-Failures nützliche Informationen!
- Wenn wir Fehler finden, kontrollieren wir die Nachbarschaft umso kritischer.
  - Fehler treten meist lokal gehäuft auf!
- Wenn wir einen Fehler «einfach so» (ohne Unit-Test) finden, dann schreiben wir noch **vor** dessen Behebung einen Unit-Test, welcher diesen Fehler zuverlässig aufdeckt!
  - Einen Fehler **nie** zweimal machen (oder ausliefern)!

# Literaturhinweise / Quellen

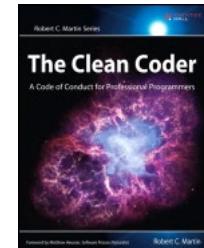
- Robert C. Martin: **Clean Code**

A Handbook of Agile Software Craftsmanship  
Prentice Hall, März 2009  
ISBN: 0132350882 / EAN: 9780132350884



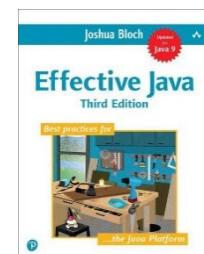
- Robert C. Martin: **The Clean Coder**

A Code of Conduct for Professional Programmers  
Prentice Hall International, Mai 2011  
ISBN: 0137081073 / EAN: 9780137081073



- Joshua Bloch: **Effective Java**

Best practices for the Java Plattform  
Pearson Addison-Wesley, Third Edition, Dezember 2017  
ISBN: 978-0-13-468599-1



**Fragen?**

Verteilte Systeme und Komponenten

# **Clean Code: Funktionen**

Roland Gisler

# Gute Funktionen nach Clean Code

- Clean Code: Kapitel 3 – Funktionen; Smells: F1 – F4
- Sehr viele Anforderungen an «gute» Funktionen:
  - Klein sein und nur eine Aufgabe erfüllen.
  - Nur eine Abstraktionsebene enthalten.
  - Geringe Einrücktiefen, möglichst kein **switch** enthalten.
  - Einen guten Namen haben.
  - Möglichst wenig (keine!) Funktionsargumente haben.
  - Keine Flag-Argumente verwenden.
  - Keine Nebeneffekte aufweisen.
  - Mit Exceptions (statt Fehlercodes) arbeiten.
  - u.v.a.m.



# Funktionen sollen klein sein

- Grundregel 1: Funktionen sollen **klein** sein!
- Grundregel 2: Funktionen sollen **noch kleiner** sein!
- Eine Funktion sollte auf einen Blick überschaubar sein.
  - Auch ohne **AGB-Schriftgrösse** und ohne Scrollen.
- Vorteil: Kleine Funktionen sind (einzeln) viel besser und schneller verständlich (**→** man muss weniger lesen!)
- Konsequenz von kleinen Funktionen:
  - Es gibt insgesamt viel mehr Funktionen.
  - Da eine Klasse nicht zu viele Methoden haben soll, gibt es auch viel mehr (aber kleinere) Klassen.
  - Beides ist **positiv**, nur schon z.B. bezüglich Testbarkeit!

# Pro Funktion nur eine Aufgabe

- Eine Funktion sollte
  - **Eine** Aufgabe erledigen.
  - Diese Aufgabe **gut** erledigen.
  - **Nur** diese Aufgabe erledigen.
- siehe OOP: **Single Responsibility Principle (SRP)**
- Wie erkennen wir was genau eine Aufgabe ist?  
Trick: Einen «**to**»-Absatz («um zu») bilden!
  - Entlehnt aus der Sprache **LOGO**  
(in welcher alle Funktionsnamen mit einem «to» beginnen!)
- Beispiel:
  - **T0 caculateSumme()** → addiere die zwei Summanden.
- Abschnitte in Funktionen: Hinweis auf Verletzung!
  - Vermutlich mehrere Aufgaben in einer Funktion enthalten.

Douglas McIlroy:  
„Mache nur eine Sache  
und mache sie gut.“  
Bekannt als die «Unix-Philosophie»

# Nur eine Abstraktionsebene pro Funktion

- Single Level of Abstraction (**SLA**)
  - Schwierig zu erlernen und zu erreichen, braucht Disziplin!
- Beispiel:

```
...
buffer.append(getPageHeader());
buffer.append(getPageContent());
buffer.append('\n');
X
buffer.append(getPageFooter());
...
```

- Grosse Gefahr: Wird **SLA** in Funktionen nicht eingehalten, wirken diese wie Magnete für den weiteren Zerfall.
  - Codeerosion im wahrsten Sinn des Wortes!

# Generell: switch-Anweisungen vermeiden!

- Mit **switch**-Anweisungen...
  - werden Funktionen gross (und werden weiter wachsen).
  - erfüllt eine Funktion typisch mehr als eine Aufgabe.
- Mit einem **switch** verletzt man viele Clean Code-Regeln:
  - **Single Responsibility Prinzip** (siehe OOP: SRP)
    - Es gibt mehr als einen Grund für Änderungen (jeder **case**).
  - **Open Closed Prinzip (OCP)**
    - **switch**-Anweisung muss für Erweiterung geändert werden.
  - **Don't repeat yourself** (siehe OOP: DRY)
    - Ein **switch**-Konstrukt kann sich im Code beliebig oft wiederholen.  
→ Eine **subtile** Form der Duplikation.
- Lösung: **switch** durch polymorphe Konstrukte (z.B. Strategy-Pattern nach GoF) ersetzen.

# Anzahl der Funktionsargumente minimieren

- Funktionsargumente sind «schwer» (im Sinn von gewichtig).
- Aus der Perspektive des Aufrufers (Nutzer) betrachten!
  - Bereits ab zwei Argumenten kann man diese vertauschen!
  - Beispiel aus JUnit-Framework:

```
assertEquals(int expected, int actual);
```



- Mal ehrlich: Wer hat die beiden Parameter noch nie versehentlich verwechselt? (Und hat es bemerkt?)
- Bei mehr als drei Argumenten hört der Spass bereits auf!
  - Ohne Konsultation der Dokumentation kaum mehr nutzbar.
  - Aufrufe werden mühsam, lang und unübersichtlich!
  - Horizontales Scrolling oder Zeilenumbrüche? ☹

# Empfehlungen aus Clean Code

- **Niladische** Funktionen - **keine** Argumente.
  - Ideal! Funktion ist sehr einfach aufzurufen.
  - Man kann praktisch keine Fehler machen.
- **Monadische** Funktionen – **ein** einziges Argument.
  - Ist ok, kann man machen wenn es nötig ist.
- **Dyadische** Funktionen - **zwei** Argumente\*.
  - Wenn es gute Gründe gibt wird es akzeptiert.
- **Triadische** Funktionen - **drei** Argumente.
  - Sollte man wenn immer möglich vermeiden!
- **Polyadische** Funktionen - **mehr** als drei Argumente.
  - Selbst bei sehr guter Begründung: **Nein!** Nicht machen!

# Reduktion der Argumente

- Grundsätzlich: Mehr und kleinere Klassen, mit Attributen!
- Häufig können dyadische Funktionen ganz einfach auf monadische Funktionen zurückgeführt werden!
  - Kann schöneres Design zur Folge haben (muss aber nicht).
- Beispiel: Dyadische (reine) Funktion:  
**summe = addiere(summand1, summand2);**
- Reduziert auf objektorientierte, monadische Form:  
**summe = summand1.addiere(summand2);**
- Reduktion verbessert auch die Lesbarkeit / Verständlichkeit.
  - Beispiel: AssertJ-Library (mit Fluent-API) für JUnit-Tests.

# Keine Flag-Argumente (formale Parameter)

- Reine Flag-Argumente sollte man möglichst vermeiden!
  - Besser die Methode in verschiedene, aber eindeutig benannte Varianten aufteilen.  
→ Auch monadische Funktionen sind nicht immer gut!
- Beispiel:

```
writeFile(myFile, true);
```



- **myFile** ist klar, aber für was ist das **boolean**-Flag?
- Besser zwei alternative Methoden:

```
writeFileOverwrite(myFile);  
writeFileAppend(myFile);
```



# Keine Nebeneffekte einbauen

- Beispiel einer Funktion:

```
boolean checkPassword(String username, String password) {  
    final User user = UserDirectory.findByName(username);  
    if (user != User.NULL) {  
        final String hash = user.getPasswordHash();  
        final String input = Cryptographer.hash(password);  
        if (input.equals(hash)) {  
            Session.initialize();  
            return true;  
        }  
    }  
    return false;  
}
```



- Gibt es darin einen unerwünschten Nebeneffekt?

# Nebeneffekte vermeiden

- Nebeneffekte sind Lügen!
  - Funktionen versprechen eine Aufgabe zu erfüllen, aber sie erledigen noch andere, verborgene Aufgaben.
- Führen zu seltsamen Kopplungen und Abhängigkeiten.
  - Im Fehlerfall nur schwer aufzudecken.
- Möglichkeiten zur Verbesserung:
  - 1. Nebeneffekt entfernen:  
Aufruf von **Session.initialize()** entfernen.
  - 2. Der Funktion einen «ehrlichen» Namen geben:  
Beispiel: **checkPasswordAndInitializeSession(...)**

# Output-Argumente vermeiden

- Output Argumente (call by reference) sind in OO-Sprachen weitgehend überflüssig geworden.
  - Wenn **ein** Rückgabewert/-objekt nicht ausreicht, ist es die Aufgabe von **this** als Output zu agieren!
- Beispiel:

```
appendFooter(s);
```


  - Wie ist das jetzt gemeint? Ist **s** ein Input oder Output?
  - Ohne Dokumentation nicht verständlich!
- Erkenntnis: Funktionsargumente werden am natürlichensten als **Input** einer Funktion interpretiert.
  - Darum: Auch immer **nur als Input** verwenden!
  - Nebenbei: Java kennt **kein** «call by reference»!

# Anweisungen und Abfragen trennen

- Funktionen sollten entweder...
  - etwas **tun** und damit den Zustand eines Objekts ändern.
  - oder **antworten** und Informationen eines Objekts liefern.
- Aber **nie** beides, weil das verwirrt!
- Beispiel einer Funktion welche diese Forderung verletzt:

```
/*
 * Setzt den Wert eines benannten Attributes.
 * Liefert true zurück wenn es geklappt hat.
 * Liefert false zurück wenn Attribut nicht existiert.
 */
public boolean set(String attribute, String value) {
    ...
}
```



# Anweisungen und Abfragen trennen

- Rückgabewerte einer Funktion «provozieren» den Einsatz in Bedingungen, wo es leicht zu Fehlinterpretationen kommen kann:

```
if (set("username", "Uncle Bob")) { ... }
```

- Wie interpretiert man nun diesen Code?
  - Prüft ob das Attribut **username** auf "Uncle Bob" gesetzt werden konnte *oder* ob dieses existiert?
  - Rein sprachliches Verständnis:  
Prüft ob **username** auf "Uncle Bob" gesetzt ist...?
- Lösung: Aufbrechen in zwei Methoden, Abfrage und Anweisung auftrennen:
  - **attributeExists(...)** und **setAttribute(...)**

# Keine Fehlercodes, besser Exceptions!

- Grundsätzlich: Fehlercodes sind eine Verletzungen der Anweisungs-/Abfrage-Trennung!
  - Fördern den Einsatz von Funktionen in **if**-Anweisungen.
- Die Fehlercodes müssen sofort ausgewertet werden.
  - Führt zu tief verschachtelten (eingerückten) Strukturen.
  - Sehr unübersichtlich und wartungsunfreundlich!
  - Gefahr der Code-Duplikationen beim Fehlerhandling.
- **Fehlercodes sind Abhängigkeitsmagneten!**
- Darum: Fehlerbehandlung besser mit Exceptions!
  - Ist auch nicht «wirklich» schön, macht den Code aber übersichtlicher und besser wartbar!

# Fehlerverarbeitung ist eine Aufgabe

- Nochmal: Eine Funktion sollte nur **eine** Aufgabe haben.
- Fehlerverarbeitung und -behandlung **ist eine** Aufgabe.
- Darum: Fehlerbehandlung in getrennte Methoden auslagern.
  - **try/catch-Konstrukte sind nicht wirklich «schön».**
- Empfehlung: Wenn eine Funktion ein **try/catch-Konstrukt** enthält dann sollte...
  - sie sonst **nichts** anderes machen.
  - das **try** immer das erste Statement sein.
  - der **catch-** (bzw. **finally-**)Block der letzte Block sein.
- So bleibt der Blick immer auf das Wesentliche fokussiert.

# Gute Namensgebung von Funktionen

- Gute Namensgebung ist und bleibt eine Kunst!
  - Tendenziell werden die Namen länger und sprechender...
  - ...und bleiben trotzdem kurz weil es mehr Packages gibt!
- Man kann es so machen:

```
package org.framework42;
public class Universal {
    public static Object doIt(Object arg1, Object arg2)
    { ... }
}
```

```
// Hier berechnen wir etwas!
x = Double.getValue(Universal.doIt(new Double(230),
                                    new Double(3.1)));
```

- Was macht dieser Code?

# Namensgebung und Grenzwerte

- Oder vielleicht besser so?

```
public class Elektrotechnik {  
    public double berechneLeistung(double spannung,  
                                    double strom) {...}  
}
```



```
leistung = berechneLeistung(230, 3.1);
```

- Empfohlene Grenzwerte für Funktionen:
  - Zeilenlänge maximal 150 Zeichen.
  - Weniger als 20 Zeilen Code (im Schnitt).
  - Keine Funktion sollte jemals länger als 100 Zeilen sein.
  - Möglichst geringe Verschachtelung, maximal zwei Ebenen!

# Uncle Bobs Funktions Smells F1 – F4

- **F1:** Zu viele Argumente.
  - Zu viele Argumente verderben den Brei!
  - 0 sind ideal, 1 ist ok, 2 in Ausnahmen, ab 3 sind es **zu viel**.
- **F2:** Output-Argumente.
  - Widersprechen der Intuition / dem was der Leser erwartet.
  - Stattdessen den Zustand des Objektes verändern!
- **F3:** Flag-Argumente.
  - Flag-Argumente zur Steuerung beweisen, dass in einer Funktion mehr als eine Aufgabe realisiert wird.
- **F4:** Tote Funktionen
  - Methoden, die nie aufgerufen werden löschen!
  - Wir haben doch alles im Versionskontrollsystem...

# Zusammenfassung – 1/3

- Funktionen sollten möglichst klein sein.
  - Wenige Zeilen, keine Abschnitte, auf einen Blick erfassbar.
- Nur eine einzige Aufgabe (Konzept) pro Funktion.
  - Unix-Philosophie: "Mach nur eine Sache und mach sie gut!"
  - Trick: «To»-Formulierung verwenden.
- Nur eine Abstraktionsebene pro Funktion (SLA).
  - Alles andere macht sie zu einem Magnet für Codeerosion.
- **switch**-Anweisungen wenn immer möglich vermeiden.
  - Potentielle Verletzung von **SRP**, **OCP** und **DRY**.

## Zusammenfassung – 2/3

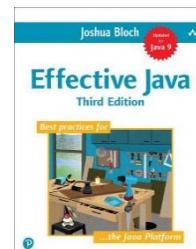
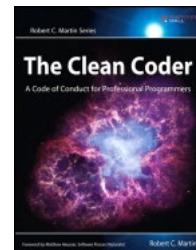
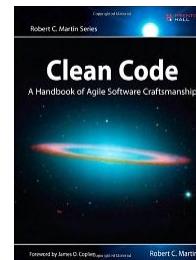
- Anzahl der Funktionsargumente möglichst reduzieren.
  - Bessere Verständlichkeit, weniger Fehler.
- Möglichst auf Flag-Argumente verzichten.
  - Weil sie sind sehr selten selbsterklärend.
- Keine unerwarteten Nebeneffekte einbauen.
  - Separation of Concerns (SoC).
- In objektorientierten Sprachen möglichst auf Output-Argumente verzichten.
- **Allgemein:** Alles Vermeiden was den Nutzer einer Funktion dazu «zwingt» eine Deklaration nachschlagen zu müssen!
  - Kognitive Unterbrechung, stört die Konzentration.

## Zusammenfassung – 3/3

- Eine Funktion sollte genau nur eine Aufgabe haben.
- Fehlerhandling ist auch eine Aufgabe!
  - Darum Fehlerhandling in eigene Methoden separieren.
- Möglichst keine Fehlercodes verwenden (besser Exceptions).
- Anweisungen und Abfragen strikte trennen.
- Gute Namensgebung von Methoden.
- Funktion Smells F1 bis F4.
- Perspektive Wechseln: Eine Funktion nicht als AutorIn schreiben, sondern als zukünftige(r) NutzerIn!
  - TDD / Test-First hilft sehr gut diesen Ansatz zu Verfolgen.

# Literaturhinweise / Quellen

- Robert C. Martin: **Clean Code**  
A Handbook of Agile Software Craftsmanship  
Prentice Hall, März 2009  
ISBN: 0132350882 / EAN: 9780132350884
- Robert C. Martin: **The Clean Coder**  
A Code of Conduct for Professional Programmers  
Prentice Hall International, Mai 2011  
ISBN: 0137081073 / EAN: 9780137081073
- Joshua Bloch: **Effective Java**  
Best practices for the Java Plattform  
Pearson Addison-Wesley, Third Edition, Dezember 2017  
ISBN: 978-0-13-468599-1



**Fragen?**

Verteilte Systeme und Komponenten

# Koordination verteilter Systeme

Martin Bättig  
(basierend auf Material von Roger Diehl)

Letzte Aktualisierung: 14. Dezember 2022

FH Zentralschweiz



# Inhalt

- Physische Zeit
- Logische Zeit
- Lamport-Zeitstempel
- Vektor-Zeitstempel

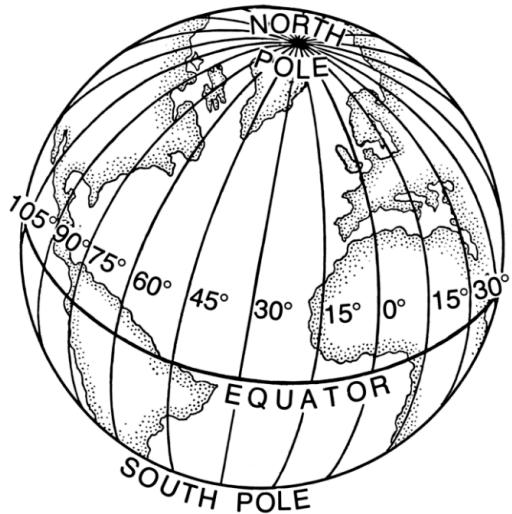
# Lernziele

- Sie kennen zwei verschiedene Algorithmen zur Synchronisation von physischen Uhren.
- Sie wissen was eine logische Uhr ist.
- Sie kennen die Happened-Before-Relation.
- Sie kennen die Algorithmen des Lamport-Zeitstempels und des Vektor-Zeitstempels zur Synchronisation von logischen Uhren.
- Sie können die Algorithmen zur Synchronisation von logischen Uhren in eigenen Programmen implementieren.

# **Physische Zeit**

# Bedeutung von Zeit

- Bestimmung der Zeit und deren Messung unverzichtbar zur Koordination menschlicher Aktivitäten.
- Koordination erreicht durch Synchronisation von Zeitmessern (Uhren).
- Synchronisation der Uhren mittels Kirchturmuhren, Telegraphie, Radio,...
- Uhrensynchronisation ermöglichte in der Schifffahrt erst die Längengradbestimmung.
- Die Existenz einer globalen Zeit haben wir verinnerlicht.

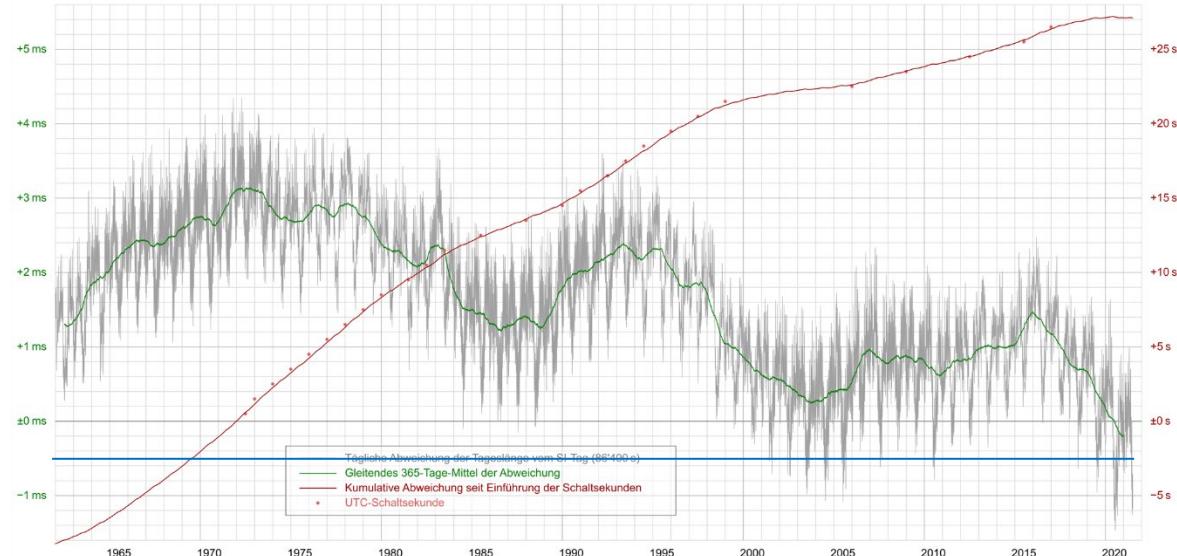


Astronomische Uhr  
Zyturm Zug

# Was ist Zeit?

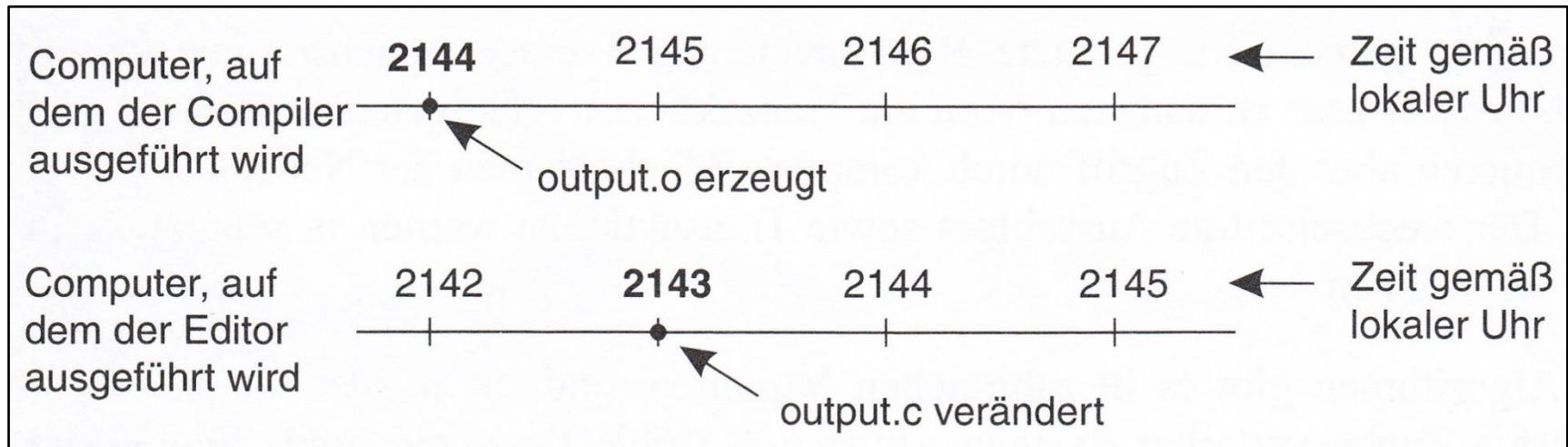
- Im 16. Jahrhundert - Einführung Gregorianischer Kalender.
- Im 17. Jahrhundert - Durchgang der Sonne im Zenit.
  - 1 Sonnentag = Zeit zwischen zwei Zenit Durchgängen.
  - 1 Sonnensekunde =  $1/86400$  eines Sonnentages.
- TAI - International Atomic Time stellt seit 1.1.1958 Anzahl Ticks der Cäsium 133-Uhren zur Verfügung.

- seit Einführung sind 86400 TAI Sek. im Mittel 2ms kürzer als ein Sonnentag.



# Was passieren kann...

- Falls es keine globale Einigung auf die Zeit gibt ist folgendes Szenario denkbar:



- Konsequenz: `output.c` wurde scheinbar zu einem früheren Zeitpunkt erstellt, deshalb wird nicht neu kompiliert!
  - Es entsteht eine Mischung aus alten und neuen Dateien.

Ist es möglich alle Uhren in einem verteilten System zu synchronisieren?

# Voraussetzung für Uhren-Synchronisierung

**Timer:** Schaltung in Computern, welche die Zeit verwaltet.

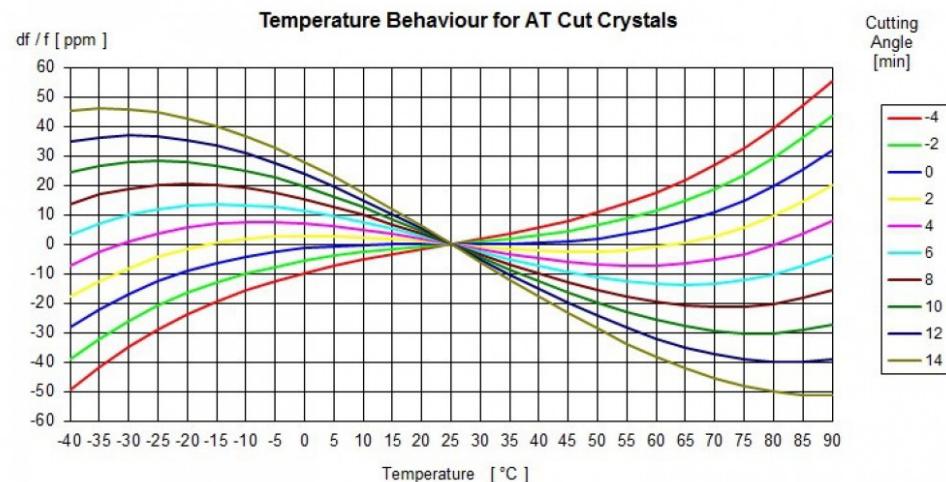
- Quarzkristall unter Spannung schwingt mit bestimmter Frequenz.
- Zählerregister zählen Schwingungen mit und erzeugen Interrupts in bestimmten Intervallen.



**Tick:** Durch den Timer erzeugter Interrupt.

**Uhrasymmetrie:** Unterschiede von Zeitwerten verschiedener Uhren, auch wenn diese ursprüngliche synchronisiert waren.

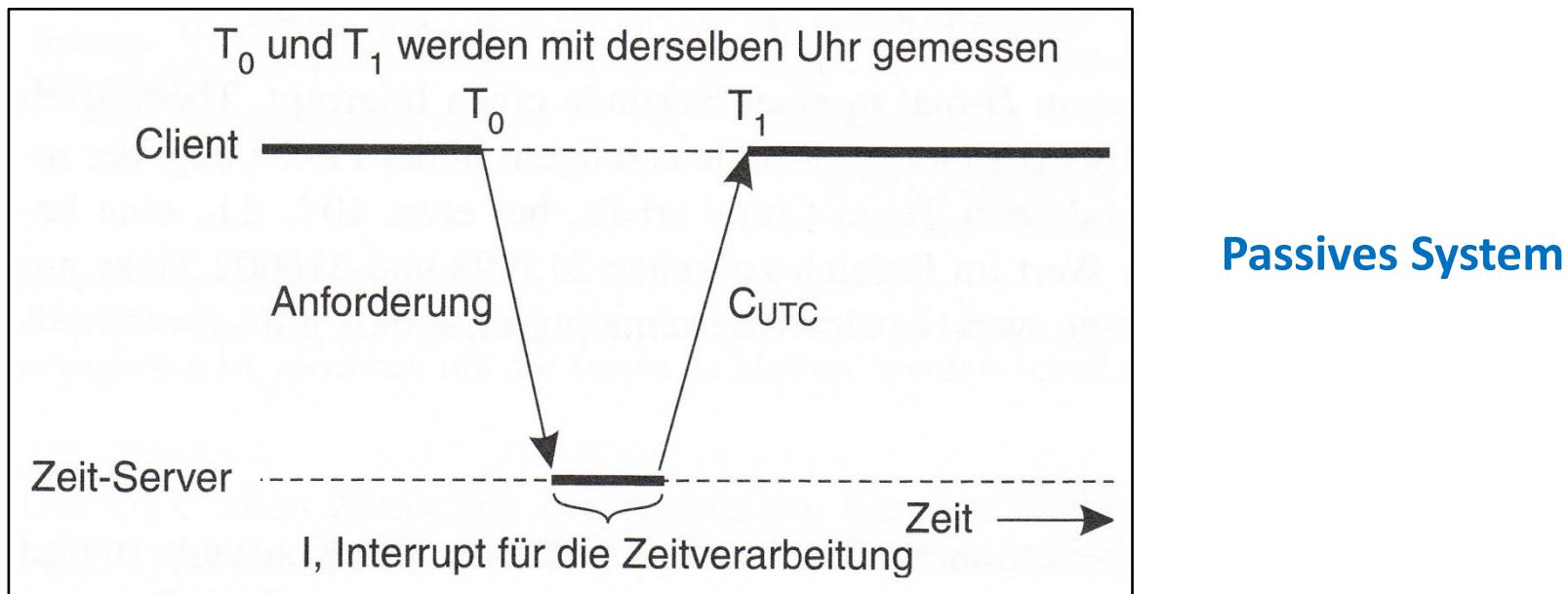
- Zeitwerte laufen auseinander, weil Quarzkristalle mit unterschiedlicher Qualität verwendet werden und deshalb mit unterschiedlichen Frequenzen schwingen.



# Algorithmus von Cristian

**Zeitserver:** Maschine mit Zeitzeichenempfänger\*, mit diesem Server werden alle anderen Maschinen synchronisiert.

- Zeitzeichensender sendet am Anfang jeder UTC-Sekunde einen kurzen Impuls.
- UTC – Universal Coordinated Time: Zeitmessung in Beziehung mit dem Sonnenstand mit Schaltsekunden.



- z.B. Langwellensender DCF77: <https://de.wikipedia.org/wiki/DCF77>

F. Cristian: *Probabilistic clock synchronization*. In: *Distributed Computing*. Volume 3, Issue 3, 1989, S. 146–158.

# Algorithmus von Cristian

1. Client P erfragt die Zeit von Zeit-Server S zum Zeitpunkt  $t_0$ .
2. Die Anfrage wird von S verarbeitet – dies benötigt eine Zeitspanne I.
3. Die Antwort  $C_{UTC}(t_1)$  wird von P zum Zeitpunkt  $t_1$  empfangen.
4. P wird auf die Zeit  $C_{UTC}(t_1) + RTT/2$  gesetzt, d.h. die vom Server gemeldete Zeit plus die Rücklaufzeit des Pakets.
  - die Round Trip Time (RTT) wird dabei berechnet durch  $RTT = t_1 - t_0$ .
  - ist die Zeitspanne I bekannt, kann die Berechnung verbessert werden  $RTT = t_1 - t_0 - I$ .
5. Für genauere Werte wird die Laufzeit öfters gemessen, Messungen ausserhalb eines Bereiches werden verworfen und eine Mittelung der restlichen Werte durchgeführt.

# Algorithmus von Cristian – Probleme

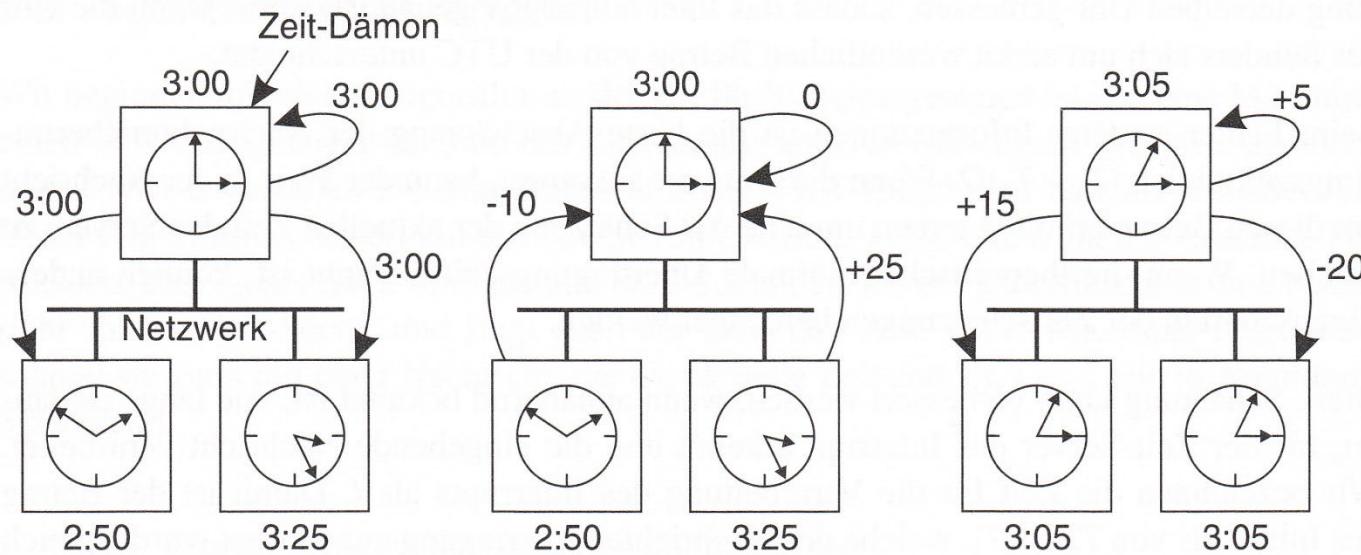
**Grosses Problem:** Zeit kann nicht rückwärts laufen.

- Zeit vom Zeitserver liegt in der Vergangenheit der lokalen Zeit.
- Zeit kann nicht einfach zurück gedreht werden, da inkonsistente Zustände im System entstehen könnten.
- **Lösung:** Verlangsamung der lokalen Zeit, bis Zeitdifferenz ausgeglichen.

**Kleines Problem:** Antwort des Zeitservers braucht Zeit.

- Laufzeit der Anfrage kann nicht genau bestimmt werden, abhängig von Netzwerklast.
- Kompensation durch mehrfache Messung der Dauer der Anfrage und Adaption des vom Zeitservers gelieferten Wert.

# Berkeley-Algorithmus



Aktives System

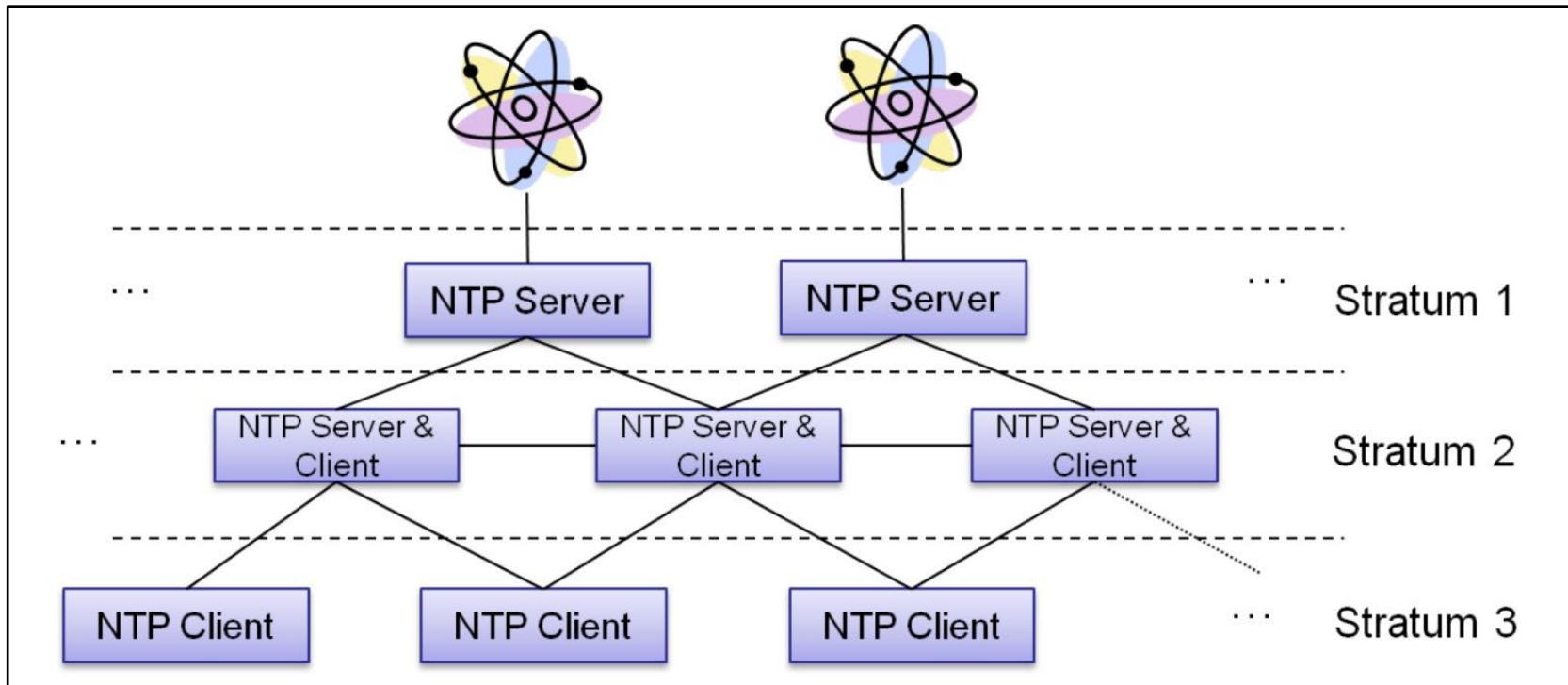
- Keine Maschine hat einen Zeitzeichenempfänger.
- Der Zeitserver (Zeit-Dämon) fragt in regelmässigen Abständen die lokale Zeit von allen teilnehmenden Clients ab.
- Basierend auf den Antworten berechnet der Zeitserver eine Durchschnittszeit und weist alle Maschinen an, ihre Uhren der neuen Zeit anzupassen.

# Network Time Protocol - NTP

- Entwickelt seit 1982 (NTP v1, RFC 1059) unter Leitung von David Mills; Aktuelle Version NTP v4, seit 1994.
- Zweck: Synchronisierung von Rechneruhren im Internet.
- NTP-Dämon auf fast allen Rechnerplattformen verfügbar, von PCs bis Crays; Unix, Windows, VMS, eingebettete Systeme.
- Erreichbare Genauigkeiten von ca. 10 ms in WANs und kleiner als 1ms in LANs.
- Fehlertolerant.
- Ausführliche Informationen zu NTP:
  - <http://www.ntp.org> ("Offizielle" NTP-Homepage).
  - <https://www.eecis.udel.edu/~mills> (Homepage David Mills).
  - <http://www.ntpclient.com> (Infos zu NTP Client Software).

# NTP - Struktur

- **Stratum 1:** primärer Zeitgeber, über Funk oder Standleitungen an amtliche Zeitstandards angebunden.
- **Stratum >1:** synchronisiert mit Zeitgeber des Stratum N - 1.
- Stratum kann dynamisch wechseln, z.B. bei Unterhalt oder Ausfall der Verbindung.



# NTP - Datenpaket

NTPv4: <https://tools.ietf.org/html/rfc5905>

	LI	VN	Mode	Strat	Poll	Prec	
Cryptosum							Root Delay
							Root Dispersion
							Reference Identifier
							Reference Timestamp Seconds (32), Fraction (32)
							Originate Timestamp Seconds (32), Fraction (32)
							Receive Timestamp Seconds (32), Fraction (32)
							Transmit Timestamp Seconds (32), Fraction (32)
							Ext. Field 1 Key Identifier (optional)
							Ext. Field 2 Message Digest (optional)
	Authenticator (Optional)						Key/Algorithm Identifier
							Message Hash (64 or 128)

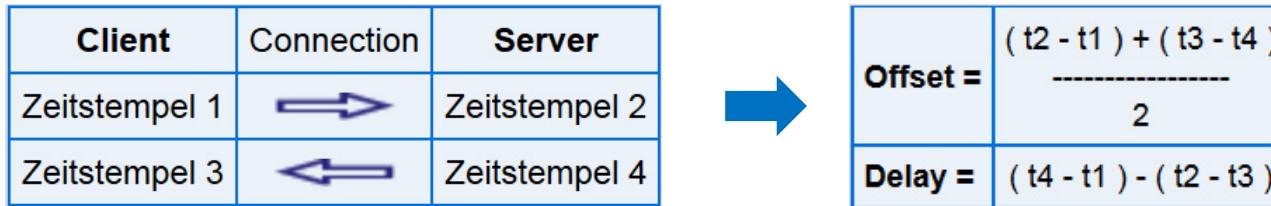
**LI** = leap indicator  
**VN** = version number  
**Strat** = Stratum (0-15)  
**Poll** = poll intervall  
**Prec** = Precision

Seconds (32-bit): Anzahl  
Sekunden seit 1.1.1900

Quelle: <https://www.meinberg.de/german/info/ntp-packet.htm>

# Prinzipieller Ablauf

1. Client sendet eine NTP-Message an den Timeserver.
2. Server verarbeitet Paket (passt IP-Addressen, Timestamps, weitere Felder an).
3. Server sendet Paket zurück.
4. Client hat nun vier Zeitstempel ( $t_1-t_4$ ) und leitet davon Offset und Delay ab:



- **Offset:** Zeitdifferenz der Rechneruhren (gemittelt).  
*Um diesen Wert wird die Zeit geändert, falls die Qualität der Messung gut ist.*
- **Delay:** Zeit während der das Paket unterwegs war.  
*Mass für die Qualität. Ggf. werden mehrere Pakete versandt und dasjenige mit dem geringsten Delay der letzten acht Pakete verwendet.*

# **Logische Zeit**

# Logische Zeit

- 1978 zeigte Leslie Lamport (\*), dass es ausreichend ist, wenn sich alle Maschinen über dieselbe Zeit einig sind.
- Eine Übereinstimmung mit der Zeit ausserhalb des Systems ist nicht notwendig (keine physische Zeit nötig).
- Logische Zeit findet vor allem in Bereichen Anwendung, in denen Kausalität und Verlässlichkeit eine grosse Rolle spielen.
- Allerdings sind die Verfahren zur Synchronisation von logischen Uhren in grossen Systemen im Allgemeinen ineffizient.

(\*) Leslie Lamport: US-amerikanischer Mathematiker, Informatiker und Programmierer. 2013 erhielt er den Turing Award für seine Beiträge zur Theorie und Praxis verteilter und nebenläufiger Systeme.

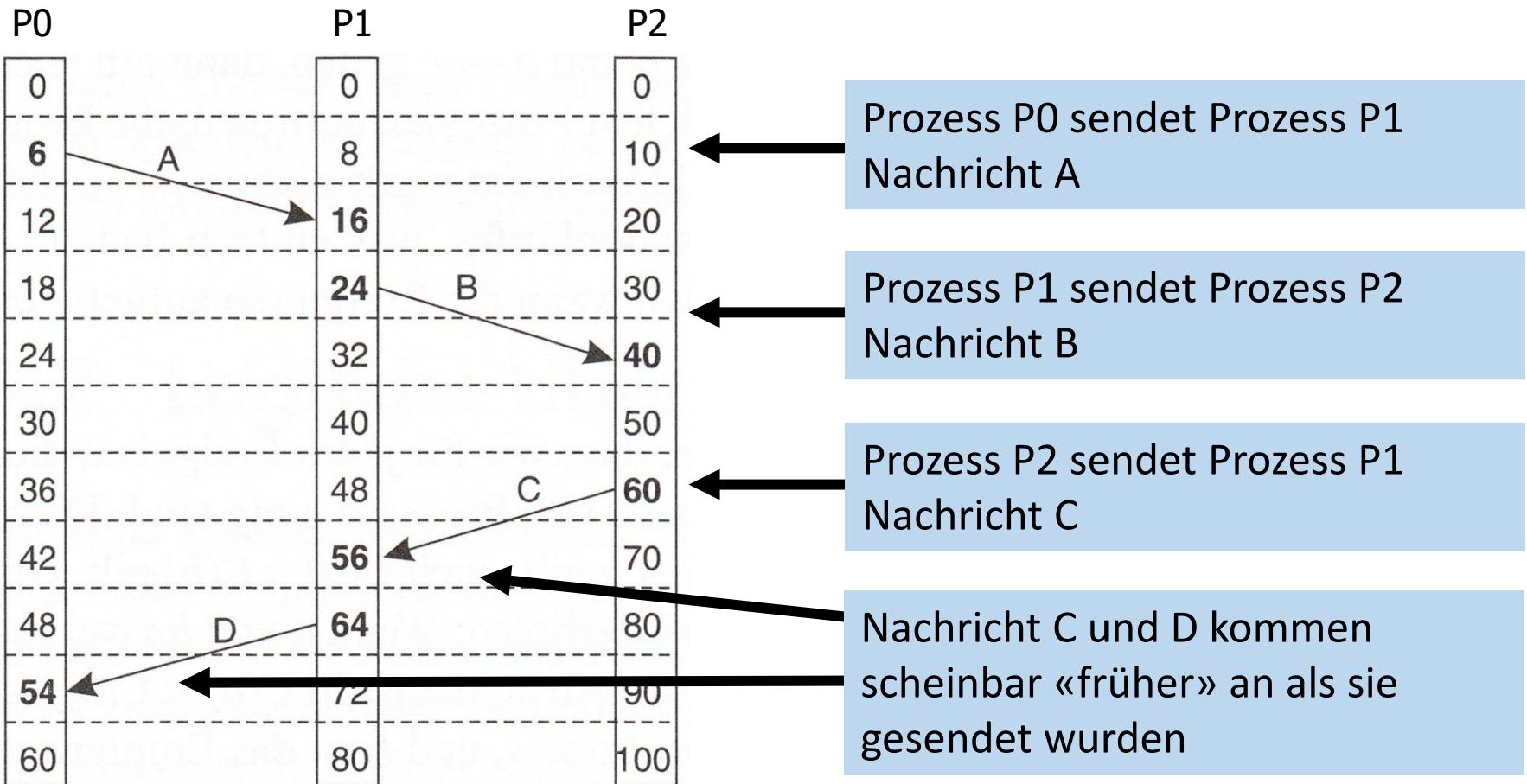
# Happened-Before-Relation von Lamport

- Der Ausdruck  $a \rightarrow b$  wird gelesen als «a passiert vor b»
  - bedeutet, dass sich **alle Prozesse einig sind**, dass
  - **zuerst das Ereignis a stattfindet und dann das Ereignis b.**
- Direkte Beobachtung der Relation in zwei Situationen:
  1. wenn a und b Ereignisse im selben Prozess sind, und a vor b auftritt, gilt  $a \rightarrow b$ .
  2. wenn a das Senden einer Nachricht bei einem Prozess und b das Empfangen derselben Nachricht bei einem anderen Prozess ist, dann gilt  $a \rightarrow b$ .
- Zwei Ereignisse  $a \neq b$  sind **kausal unabhängig**, geschrieben als  $a \parallel b$ , wenn weder  $a \rightarrow b$  noch  $b \rightarrow a$  sind
- Happened-Before-Relation ist **transitiv**: Wenn  $a \rightarrow b$  und  $b \rightarrow c$  gelten, dann gilt auch  $a \rightarrow c$

# **Lamport-Zeitstempel**

# Ausgangslage

Jede Maschine hat eine eigene Zeit mit konstanten aber unterschiedlichen Geschwindigkeiten.

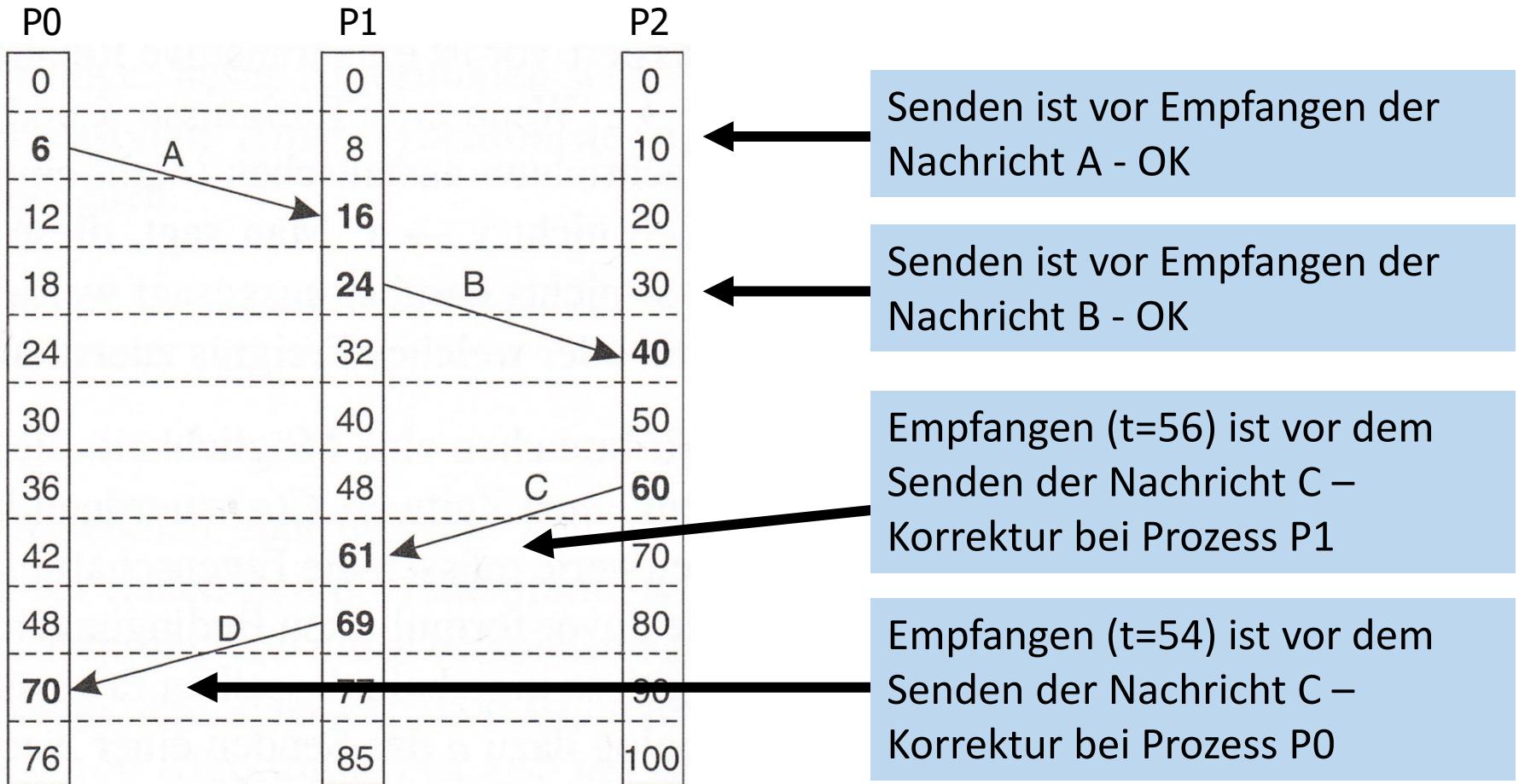


# Lamport-Zeitstempel

- Ein Prozess sendet eine Nachricht mit Zeitstempel (eigene Zeit) an einen anderen Prozess.
- Einem Ereignis a wird ein Zeitwert C(a) zugeordnet.
  - alle Prozesse sind sich über den Zeitwert einig.
  - wenn  $a \rightarrow b$  gilt auch  $C(a) < C(b)$ .
- Ein Prozess sendet eine Nachricht mit Zeitstempel a (eigene Zeit) an einen anderen Prozess, welcher die Nachricht zur eigenen Zeit b empfängt, dann müssen C(a) und C(b) so zugewiesen werden, dass  $C(a) < C(b)$  ist.
- Die Zeit C muss **immer vorwärts laufen**.
  - ansteigende Werte.
  - Korrekturen können durch **Addition von positiven Werten** vorgenommen werden.

# Lösung

Zwischen zwei Ereignissen muss die lokale Zeit **mindestens einmal ticken** – empfangene Zeit + 1.



Quelle: <https://de.wikipedia.org/wiki/Lamport-Uhr>

# Lamport-Zeitstempel zusätzliche Forderung

Zwei Ereignisse dürfen nie zu genau der selben (logischen) Zeit auftreten.

**Lösung:** Zeitstempel um Prozessnummer ergänzen.

Damit kann allen Ereignissen in einem verteilten System eine Zeit zugewiesen werden, die folgenden Bedingungen erfüllt:

1. wenn a im selben Prozess vor b auftritt, gilt  $C(a) < C(b)$ .
2. wenn a und b das Senden und Empfangen einer Nachricht darstellen, gilt  $C(a) < C(b)$ .
3. für alle anderen Ereignisse a und b, gilt  $C(a) \neq C(b)$ .

# Lamport-Zeit – Eigenschaften

- Lamports Uhren erfüllen die Uhrenbedingung:  $a \rightarrow b \Rightarrow C(a) < C(b)$ .  
Wenn Ereignis a vor Ereignis b stattfindet, dann ist der Zeitstempel von  $C(a)$  kleiner als der von  $C(b)$ .
- Die logischen Lamport-Zeitstempel definieren daher eine partielle Ordnung auf der Menge der Ereignisse, die den kausalen Zusammenhang zwischen Ereignissen erhält.
- Ergänzung zu einer totalen Ordnung ist wieder möglich.

**Einschränkung:** Anhand der Zeitstempel lässt sich nicht immer sicher sagen, ob zwei Ereignisse kausal voneinander abhängen.

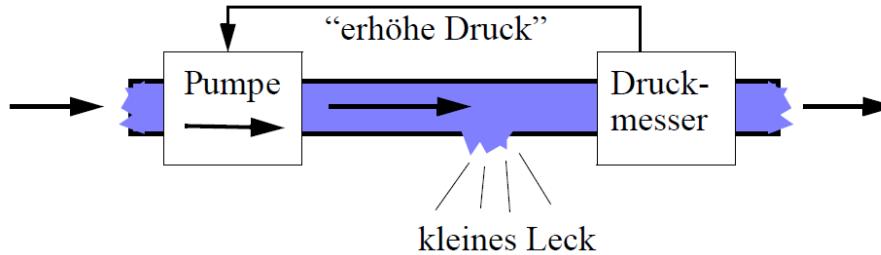
- hierfür müsste auch die Umkehrung der Uhrenbedingung gelten, aber es gilt lediglich  $C(a) < C(b) \Rightarrow a \rightarrow b \vee a \parallel b$

# **Vektor-Zeitstempel**

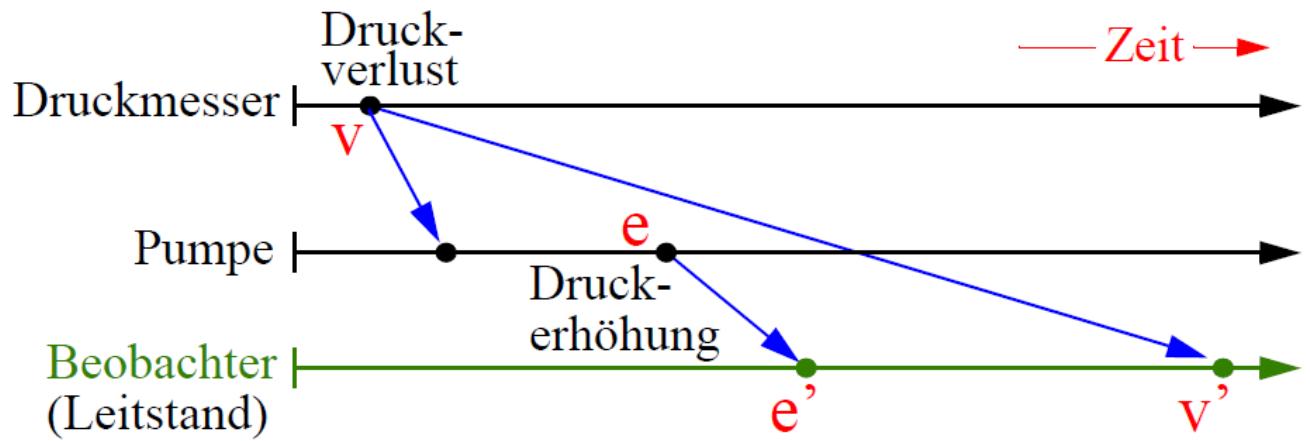
# Beispiel: (nicht) kausaltreue Beobachtungen

**Gewünscht:** Eine Ursache stets vor ihrer (u.U. indirekter) Wirkung beobachten.

Was passiert ist:



Eingehende Nachrichten:



**Falsche Schlussfolgerung des Beobachters:**

Es erhöhte sich der Druck (aufgrund einer Aktivität der Pumpe), es kam zu einem Leck, was durch den abfallenden Druck angezeigt wird.

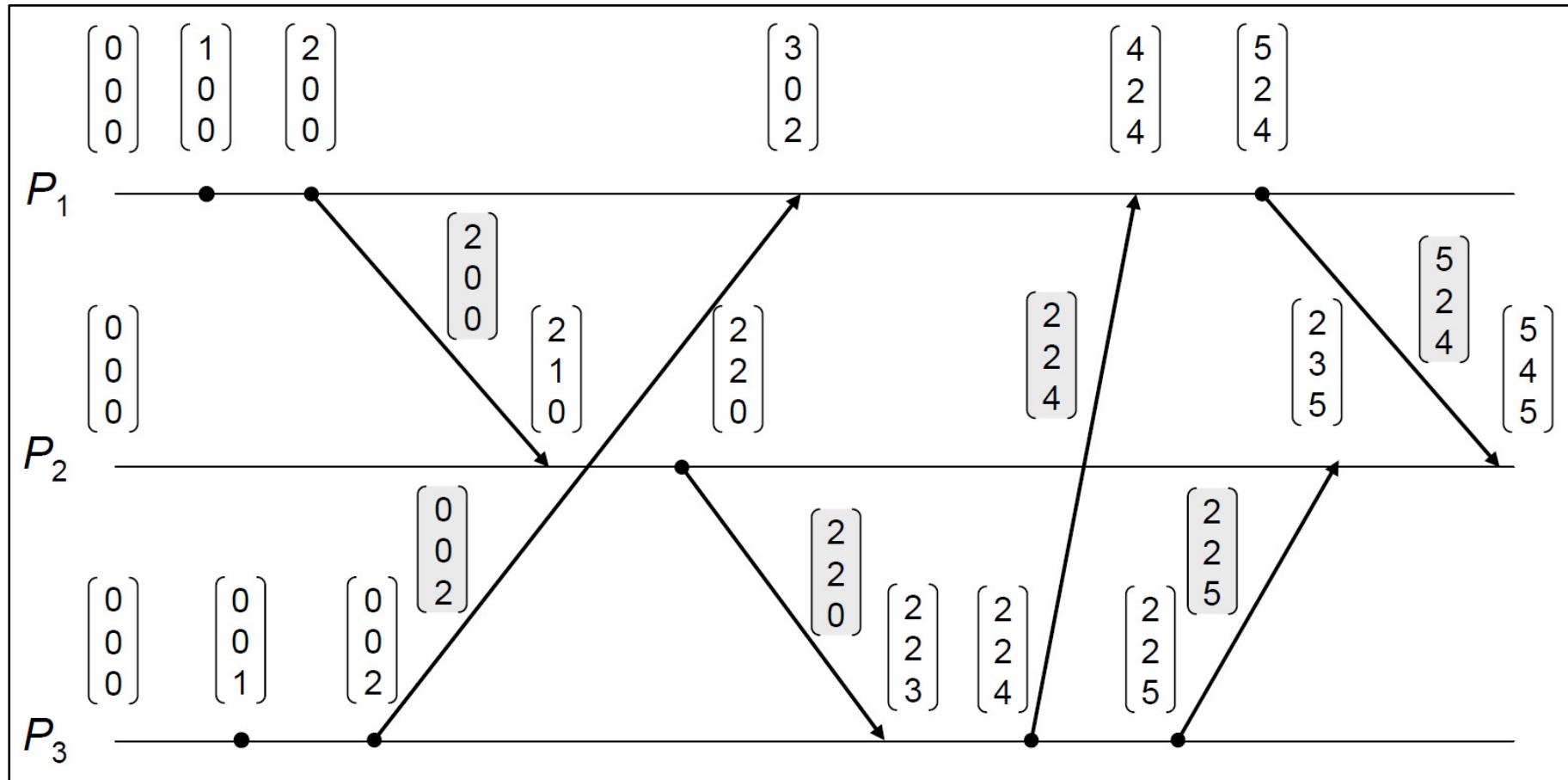
# Definition

- Ein Vektor-Zeitstempel  $VT(a)$ , der einem Ereignis a zugewiesen wurde, hat die Eigenschaft, dass Ereignis a dem Ereignis b kausal vorausgeht, wenn  $VT(a) < VT(b)$  für ein Ereignis b gilt.
- Jeder Prozess  $P_i$  besitzt einen Vektor  $V_i$ , der für jeden Prozess im System die Anzahl der Ereignisse enthält mit den Eigenschaften:
  - $V_i[i]$  ist die Anzahl der Ereignisse, die bisher in  $P_i$  aufgetreten sind
  - wenn gilt  $V_i[j] = k$ , erkennt  $P_i$ , dass in  $P_j$  k Ereignisse aufgetreten sind.
- Der Vektor  $V_i$  wird den gesendeten Nachrichten mitgegeben.

# Algorithmus Vektor Zeitstempel

- Jeder Prozess  $P_i$  hält einen Vektor  $V_i$  bestehend aus  $n$  Zählern ( $n = \text{Anzahl der Prozesse im System}$ ).
- Initial ist der Vektor Zeitstempel jedes Prozesses der Nullvektor.
- Tritt bei Prozess  $P_i$  ein Ereignis auf, so inkrementiert er die  $i$ -te Komponente seines Vektor.
- Sendet  $P_i$  eine Nachricht, so wird die neue Version von  $V_i$  mitgeschickt.
- Empfängt  $P_i$  eine Nachricht mit Vektor Zeitstempel  $VT$ , so bildet er das **komponentenweise Maximum** von der neuen Version von  $V_i$  und von  $VT$ .

# Beispiel Vektor-Uhren



grau: gesendeter Vektor Zeitstempel

# Informationen des Vektor-Zeitstempel

Der Vektor-Zeitstempel in der Nachricht informiert Empfänger über

- die Anzahl Ereignisse die in  $P_i$  aufgetreten sind,
- wie viele Ereignisse in anderen Prozessen der Nachricht vorausgegangen sind,
- wie viele vorangegangene Ereignisse möglicherweise kausal abhängig sind.

# Kausaler Zusammenhang zwischen zwei Ereignissen

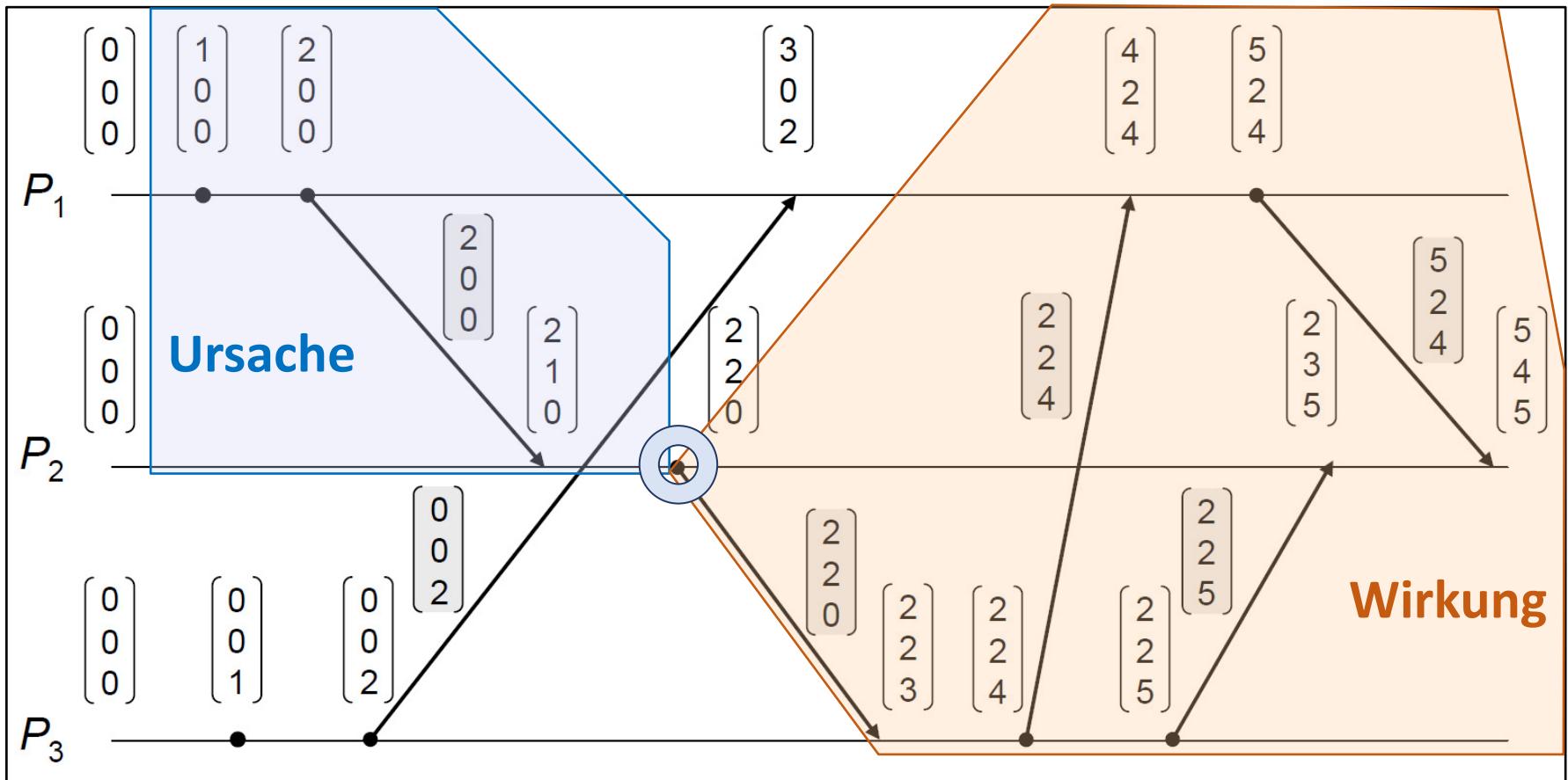
Ereignis A ist eine Ursache von Ereignis B:

- wenn der Zähler für jeden Prozess im Zeitstempel VT(A) kleiner oder gleich dem Zähler im Zeitstempel VT(B) für den korrespondierenden Prozess
- und für mindestens einen dieser Zähler kleiner ist.

**Beispiele:**

- $\begin{pmatrix} 2 \\ 0 \\ 0 \end{pmatrix}$  ist Ursache für  $\begin{pmatrix} 4 \\ 0 \\ 1 \end{pmatrix}$
- $\begin{pmatrix} 1 \\ 1 \\ 2 \end{pmatrix}$  ist Ursache für  $\begin{pmatrix} 1 \\ 2 \\ 3 \end{pmatrix}$
- $\begin{pmatrix} 2 \\ 1 \\ 3 \end{pmatrix}$  ist keine Ursache für  $\begin{pmatrix} 1 \\ 3 \\ 4 \end{pmatrix}$

# Beispiel: Kausaler Zusammenhang von Ereignissen



# Klassenraumübung: Logische Zeit

Stellen Sie zum verteilten Logging-System folgende Überlegungen an:

- a) Wo könnte logische Zeit zum Einsatz kommen? Begründen Sie in jedem Fall Ihre Antwort,
  - warum Sie logische Zeit einsetzen oder
  - warum Sie logische Zeit nicht einsetzen.
- b) Welchen Mehrwert ergäbe die logischen Zeit im Projekt?
- c) Welche logische Zeit (mit Lamport-Zeitstempel oder Vektor-Zeitstempel) ist sinnvoll, bezüglich des Mehrwerts vs. Aufwand?

# Zusammenfassung

- Zeitbestimmung und Messung von Zeitdauern ist unverzichtbar zur Koordination von Aktivitäten.
- Zeitwerte verschiedener Uhren laufen auseinander, auch wenn diese synchronisiert waren (Uhrasymmetrie). Zwei Algorithmen, Cristian und Berkeley, sind zur Synchronisierung möglich.
- Lamport sagt, dass es ausreichend ist, wenn sich alle Maschinen über dieselbe Zeit einig sind. Eine Übereinstimmung mit der Zeit ausserhalb des Systems ist nicht notwendig.
- Die Happened-Before-Relation besagt, dass eine Nachricht nicht empfangen werden kann, bevor sie gesendet wurde.
- Beim Lamport-Zeitstempel wird einer Nachricht die Uhrzeit des sendenden Prozesses mitgegeben. Der empfangende Prozess richtete seine Uhrzeit nach dem Zeitstempel + 1 (mindestens).
- Die Uhrzeit muss immer vorwärts laufen.

# Literatur

- Distributed Systems (3rd Edition), Maarten van Steen, Andrew S. Tanenbaum, Verleger: Maarten van Steen (ehemals Pearson Education Inc.), 2017.

# **Fragen?**

Verteilte Systeme und Komponenten

# Sicherheit in verteilten Systemen

Martin Bättig

Letzte Aktualisierung: 15. Dezember 2022

FH Zentralschweiz



# Inhalt

- Übersicht
- Transportlayer-Security (TLS)
- Sessions
- Authentifizierung und Autorisierung

# Lernziele

- Sie kennen die notwendigen Massnahmen für eine sichere Kommunikation in verteilten Systemen.
- Sie kennen die Grundlagen des TLS-Protokolls und wie eine TLS-Verbindung auf Socketebene erstellt wird.
- Sie wissen, wie die Erstellung von Zertifikaten in den Grundzügen funktioniert.
- Sie kennen das Prinzip einer Session und können diese in Relation zu einer TCP-Connection setzen.
- Sie kennen den Ablauf einer Authentifizierung mittels Passwort und wie Passwörter mittels Java sicher gespeichert werden können.
- Sie wissen, was eine Autorisierung ist und wie eine einfache Autorisierung mittels Java realisiert werden kann.

# **Sicherheit in verteilten Systemen**

# Cyber-Security vs. Betriebssicherheit



## Cyber-Security:

Schutz kritischer Systeme und sensibler Informationen vor digitalen Angriffen.

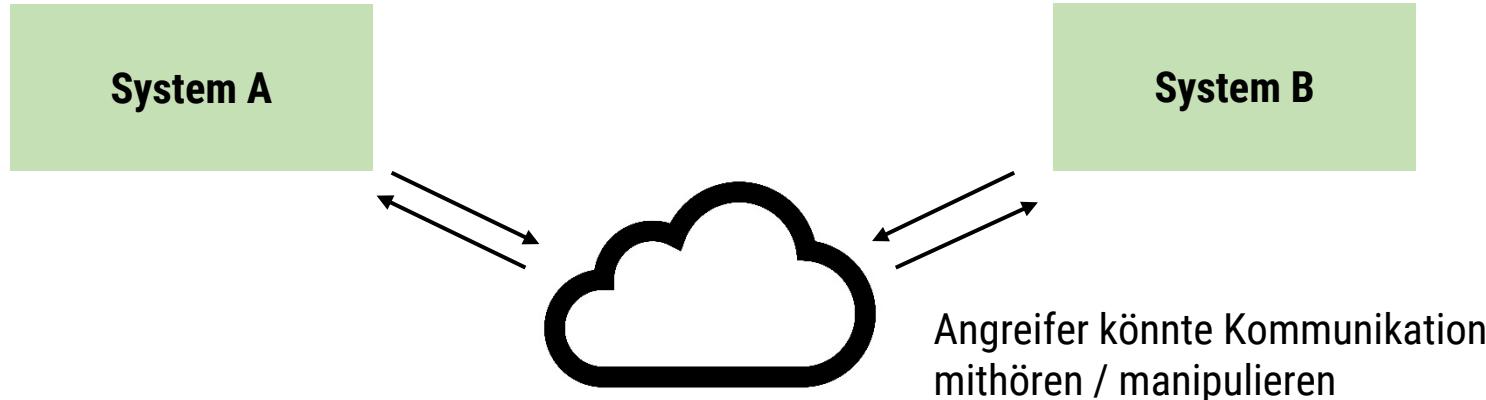


## Betriebssicherheit:

Fehlfunktionen treten nicht auf oder verursachen keine kritischen Schäden an Mensch und Maschine.

# Fokus: Sichere Kommunikation zwischen verteilten Systemen

**Situation:** A und B sollen sicher kommunizieren. Daten gehen über Drittsysteme, eine lange Verbindung oder werden drahtlos übertragen:



## Sichere Kommunikation:

1. A muss sicherstellen, dass B das korrekte System ist.
2. B muss sicherstellen, dass A ein berechtigtes System ist.
3. Kein Drittsystem C soll die Kommunikation mithören.
4. Kein Drittsystem C soll die Kommunikation manipulieren.

Dies während der gesamten Dauer der Kommunikation (Sitzung).

# Cyber-Security - Massnahmen

- **Zugriffsschutz:** Nur berechtige Benutzer und Systeme können auf unserer System regulär zugreifen.  
-> [Authentifizierung und Autorisierung.](#)
- **Manipulationssicherheit:** Gesendete Daten können nicht manipuliert werden.  
-> [Signaturen.](#)
- **Abhörsicherheit:** Keine geschützte Informationen gelangen nach aussen.  
-> [Verschlüsselung.](#)
- **Nachvollziehbarkeit:** Wissen darüber wie und von wem das System verwendet wurde.  
-> [Logs / Audit-Trails.](#)

# Welche Informationen verbergen?

Tatsache, dass Kommunikation stattfindet, lässt sich nur schwer verbergen  
(Achtung: Seitenkanäle).

**Beispiel:** Visible Light Communication.

Kommunikation durch Variation der Frequenz.

Sieht aus wie Licht: Kommunikation für menschliches Auge zunächst verborgen.

Mittels Technik aber erkennbar.



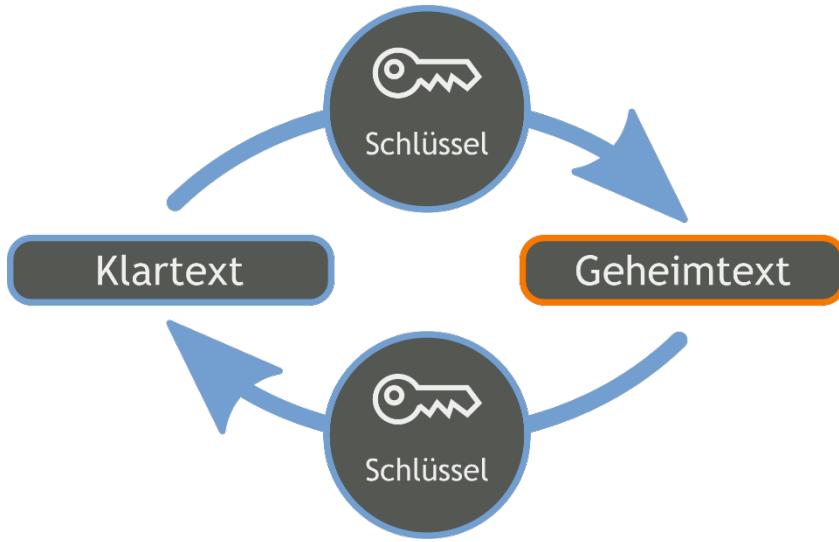
Quelle: Disney Research

## Pragmatisches Vorgehen:

- Wenn es einfach ist: **Kommunikationsinhalt** verbergen (verschlüsseln).
- Wenn es kompliziert wird, gut überlegen, was Sinn ergibt.
- Datenverschlüsselung ist Basis für sichere Datenübertragung.  
⇒ Unterscheidung: **symmetrische** und **asymmetrische** Verschlüsselung.

# Symmetrische Verschlüsselung

Gleicher Schlüssel zum Ver- und Entschlüsseln verwendet:



- Effizienter als asymmetrische Verschlüsselung.
- I.d.R. eingesetzt zum Verschlüsseln von Datenströmen.

# Asymmetrische Verschlüsselung

## Erzeugung eines Schlüsselpaars:

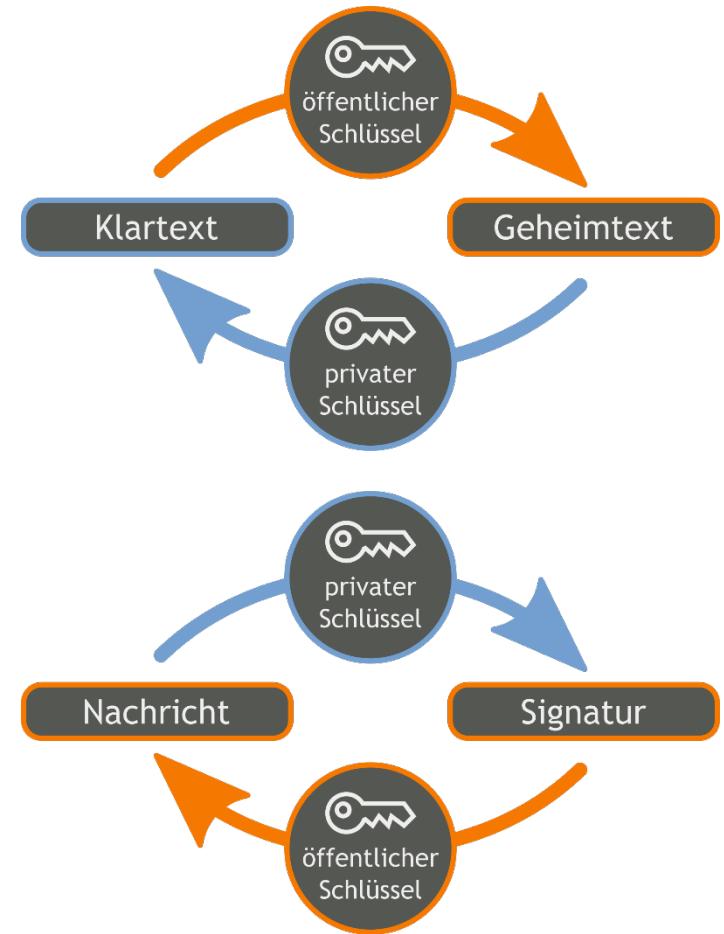
- **Privater Schlüssel** (nur Erzeuger A bekannt).
- **Öffentlicher Schlüssel** (allen bekannt, z.B. B).

## Einsatz zum Verschlüsseln:

- B verschlüsselt mittels **öffentlichen Schlüssel**.
- Nur A kann mittels **privatem Schlüssel** entschlüsseln.

## Einsatz um Signieren und Zertifizieren:

- A verschlüsselt Hash einer Information I mittels **privatem Schlüssel** => Signatur.
- B entschlüsselt Signatur mit **öffentlichen Schlüssel**. Entspricht diese dem Hashwert von I hat A die Signatur erstellt, da nur A den **privaten Schlüssel** kennt.

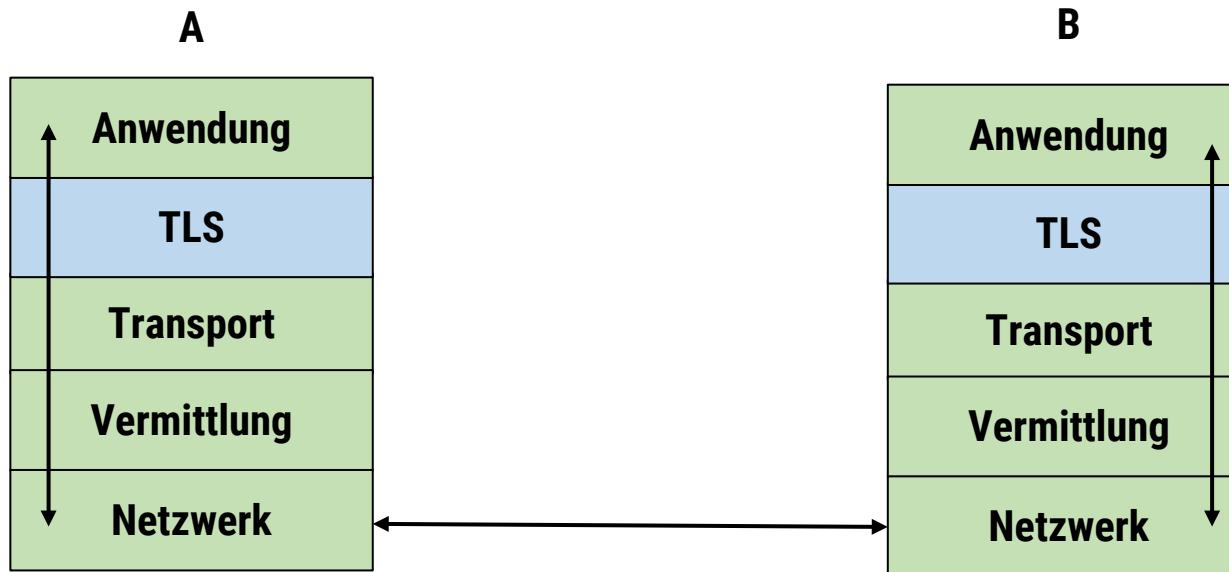


# **Transportlayer Security (TLS)**

# Grundlagen

- Transparente Verschlüsselung der **Anwendungskommunikation**.
- Von Konzept her eine Transportschicht (z.B. TCP).
- Implementiert als Protokoll in der Anwendungsschicht.

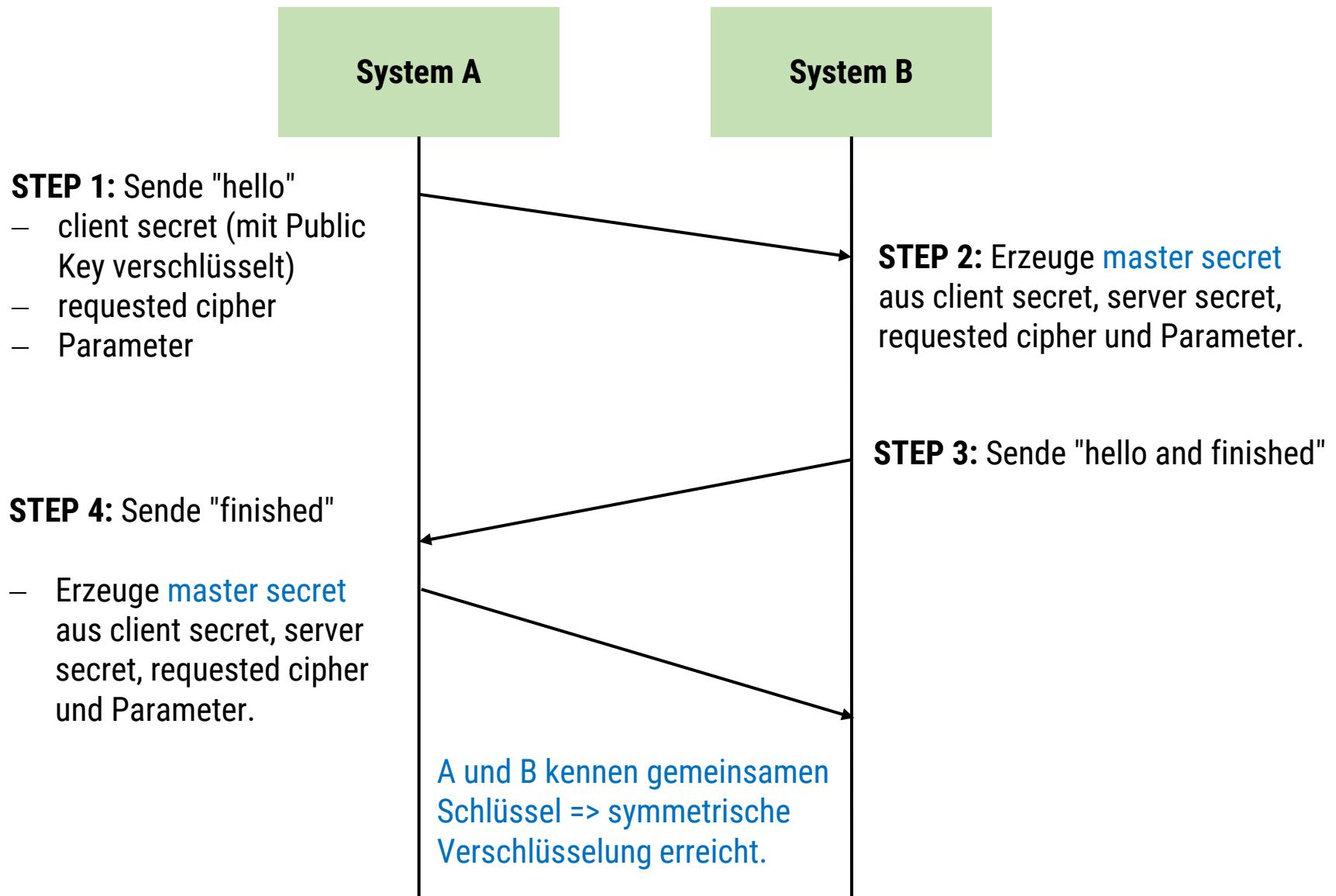
Verortung von TLS innerhalb der Internetschichtenmodelle:



# Versionen des TLS-Protokolls

Version	Einführung	Info
SSL 1.0	1994	Nicht mehr in Gebrauch, unsicher
SSL 2.0	1995	Nicht mehr in Gebrauch, unsicher
SSL 3.0	1996	Nicht mehr in Gebrauch, unsicher
TLS 1.0	1999	Nicht mehr in Gebrauch, unsicher
TLS 1.1	2006	Nicht mehr in Gebrauch, unsicher
TLS 1.2	2008	In Gebrauch, noch sicher
TLS 1.3	2018	Aktuelle Version, wenn möglich verwenden (RFC 8446)

# Verbindungsauftbau (TLS 1.3 Handshake)

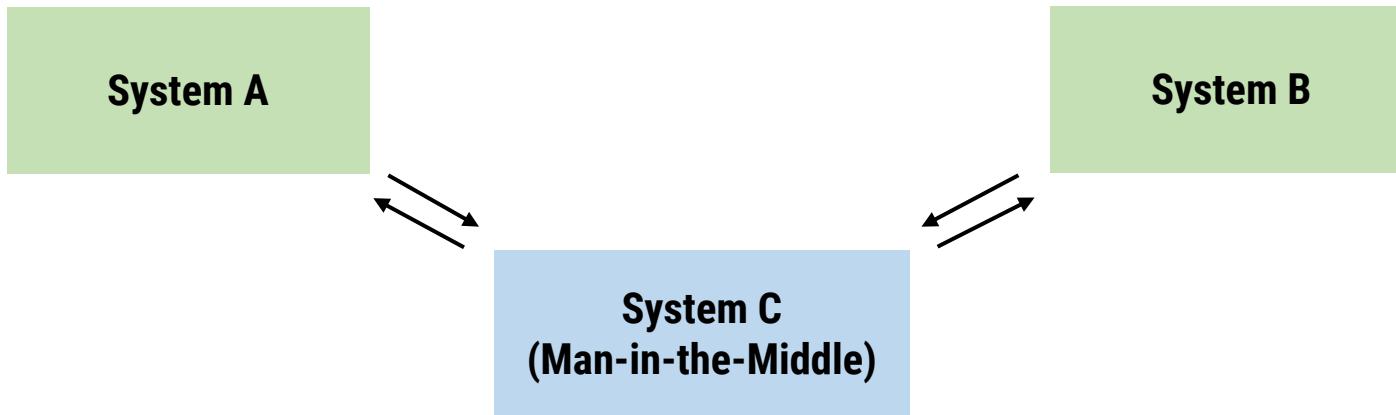


# TLS gekoppelt mit Zertifikatsprüfung

- TLS verschlüsselt in Kombination mit Authentifizierung der Gegenstelle.
- Authentifizierung mittels [X.509 Zertifikaten](#).
- Notwendig zur Verhinderung von "Man-in-the-Middle"-Attacken.

**Annahme es gäbe keine Authentifizierung:**

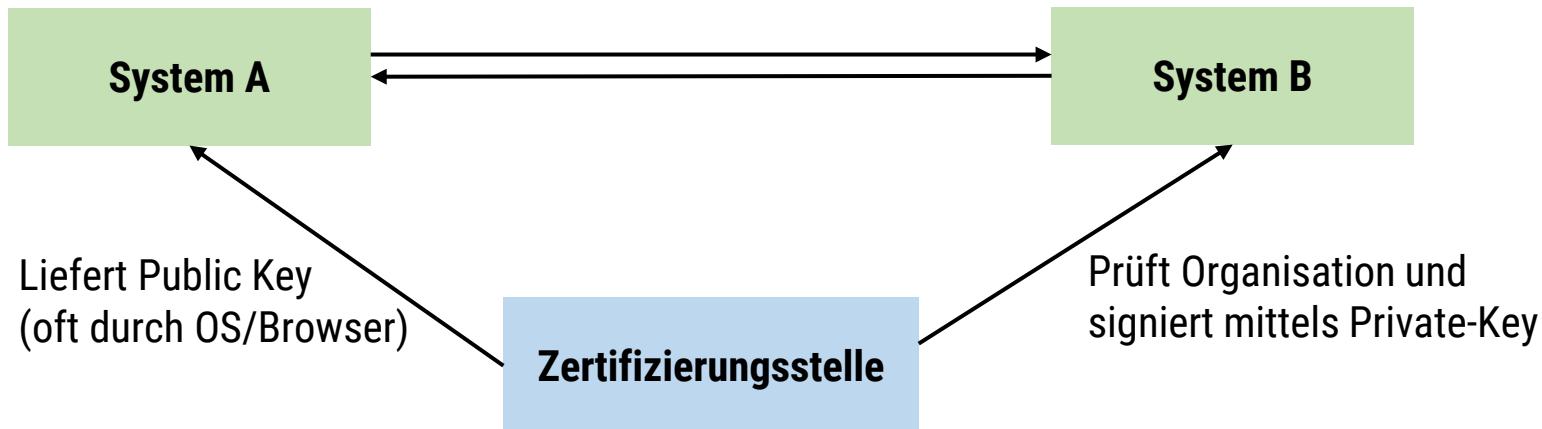
- A stellt Verbindung mit B her und wird via C geroutet:



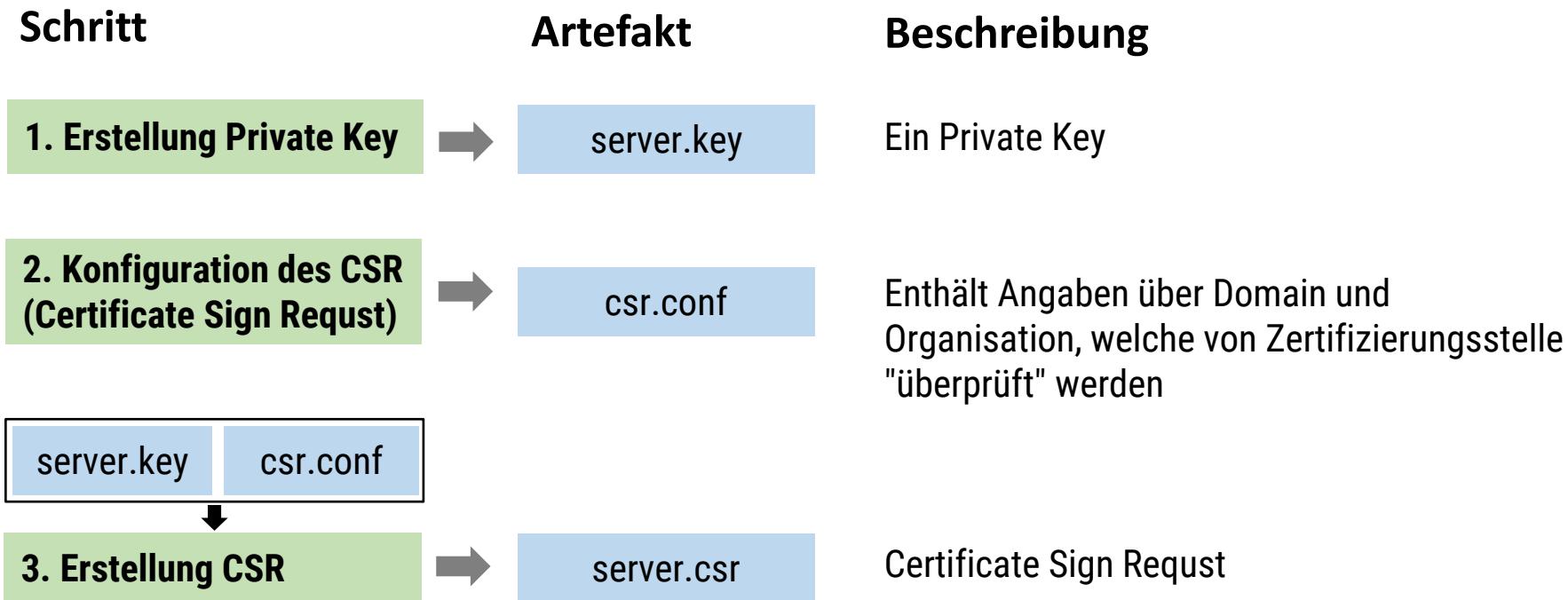
- C könnte beim Aufbau der Verbindung A und B [eigene Schlüssel](#) senden und Klartextkommunikation lesen.  
(d.h.: C gibt sich gegenüber A als B aus und gegenüber B als A).

## X.509 Zertifikate

- binden eine Identität mittels digitaler Signatur zu öffentlichem Schlüssel.
- sind zertifiziert durch **Zertifizierungsstelle** oder **selbstzertifiziert**.
- bedingen Vertrauen in Zertifizierungsstelle (wurde Identität geprüft?).
- unterteilt in zwei Kategorien:
  - **Zertifizierungsstelle (CA)**: Kann weitere Zertifikate erteilen. Mehrere Hierarchiestufen. Vertrauen in oberste Hierarchie (Root-CA) benötigt.
  - **Endstelle**: Identifizierte eine Entität (Domain, Person, Organisation, etc.)



# Erstellung eines Certificate Sign Request



# Ausstellung Zertifikat durch Zertifizierungsstelle



# Zertifikatsausstellung (Selbstzertifiziert)

Schritt	Artefakt	Beschreibung
1. Erstellung eigener Zertifizierungsstelle	rootCA.key rootCA.crt	Private Key unserer Zertifizierungsstelle. Zertifikat unserer Zertifizierungsstelle.
2. Parametrisierung des Zertifikats	cert.conf	Konfiguration des erteilten Zertifikats.
rootCA.key    cert.conf rootCA.crt    server.csr		
3. Signierung des Zertifikats	server.crt	Public Key für server.key durch rootCA.key signiert.

# Aufbau einer sicheren Verbindung mit Java (Keystore)

- Java erwartet Zertifikate und Schlüssel in einem Keystore:

```
## Erstelle PKCS12-Bundle mit Serverzertifikat, Private-Key and, rootCA-Zertifikat)
```

```
openssl pkcs12 -export -in server.crt -inkey server.key -chain -CAfile rootCA.crt -name localhost -out server.p12
```

```
## Erstelle Keystore der alle Element des PKCS12-Bundle enthält
```

```
keytool -importkeystore -deststorepass myServerPass -destkeystore server.jks -srckeystore server.p12 -srcstoretype PKCS12
```

```
## create a java keystore that contains the certificate of our own CA
```

```
keytool -import -v -trustcacerts -alias server-alias -file rootCA.crt -keystore cacerts.jks -keypass myCaCertsPass -storepass myCaCertsPass
```

# Aufbau einer sicheren Verbindung mit Java (Client)

```
private static final String HOST = ...;  
  
public static void main(final String[] args) {
```

```
    SSLSocketFactory factory = (SSLSocketFactory)  
        SSLSocketFactory.getDefault();
```

## Schritt 1: Erstellung einer SSLSocketFactory

```
    try (SSLSocket socket = (SSLSocket) factory.createSocket(HOST, 1234)) {
```

## Schritt 2: Erstellung eines SSLSocket

```
        socket.setEnabledProtocols(new String[] {"TLSv1.3"});  
        socket.setEnabledCipherSuites(new String[] {"TLS_AES_128_GCM_SHA256"});
```

## Schritt 3: Setzen unterstützter Versionen und Algorithmen

```
        // ab hier: Verwenden wie regulären TCP -Socket  
        // ...  
    }  
}
```

**Server:** Analog mittels SSLSocketFactory vorgehen

# Beispiel: Starten von Client und Server mit TLS

**Server:** Keystore (Name ist beliebig, hier server.jks) hier mit Private Key und Zertifikat (immer):

```
java -Djavax.net.ssl.keyStore=server.jks \
      -Djavax.net.ssl.keyStorePassword=myServerPass \
      myServerClassName
```

**Client:** Keystore (Name ist beliebig, hier cacerts.jks) mit Zertifizierungsstelle (nur bei selbsterstellen Zertifikaten):

```
java -Djavax.net.ssl.trustStore=cacerts.jks \
      -Djavax.net.ssl.trustStorePassword=myCaCertsPass \
      myClientClassName
```

**Debug-Modus:** Falls SSL-Modus nicht funktioniert:

```
-Djavax.net.debug=ssl
```

# TLS bei gRPC und ZeroMQ

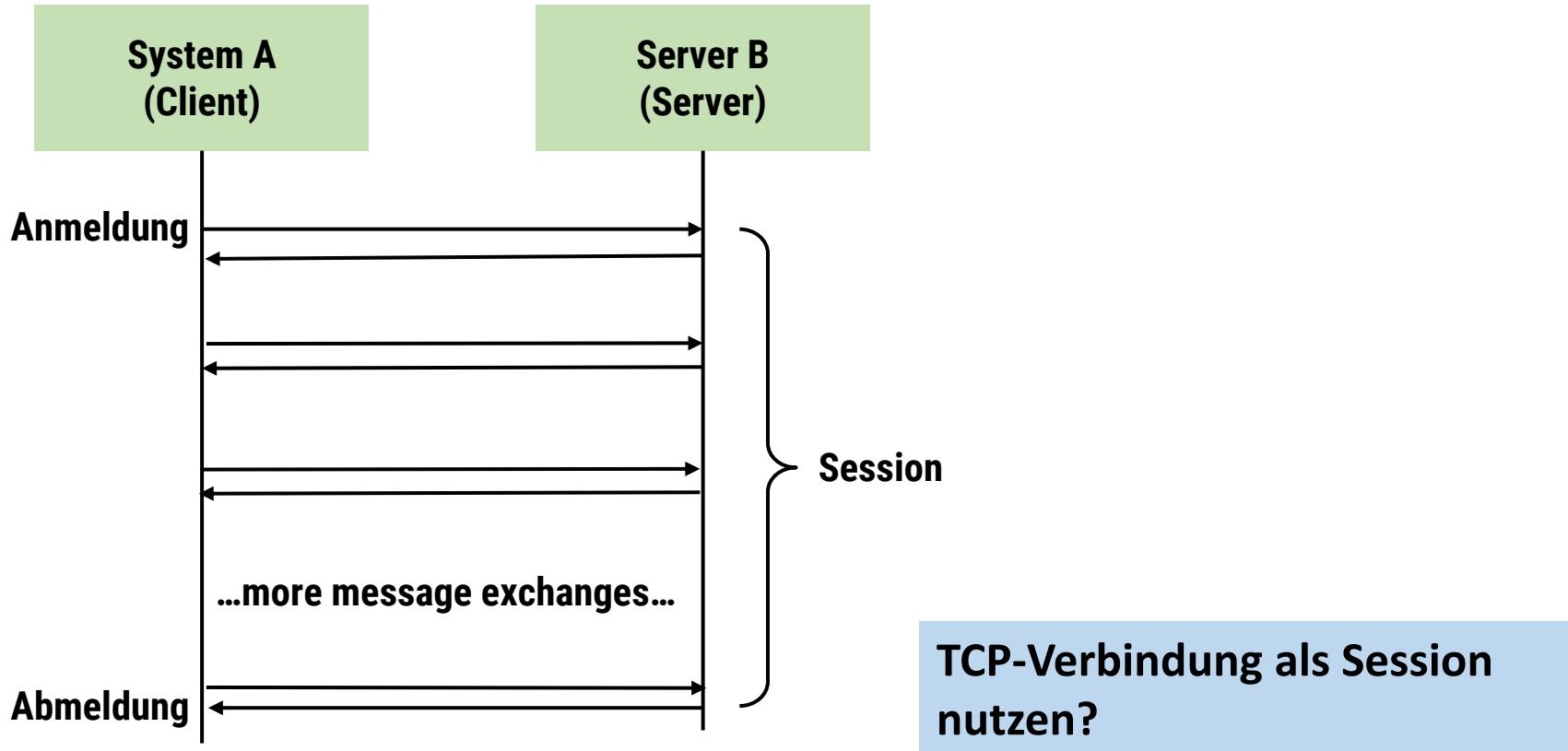
- **gRPC:** Unterstützt TLS per Default  
Anleitung: <https://grpc.io/docs/guides/auth/>
- **ZeroMQ:** Eigenes Transportverschlüsselungsprotokoll basierend auf **CurveCP** (<http://curvecp.org>).
  - Alternativ: Kommunikation separat mittels Werkzeugen via stunnel ([https://www.stunnel.org/config\\_unix.html](https://www.stunnel.org/config_unix.html)) verschlüsseln.

# **Sessions**

# Session (Sitzung)

## Definition:

- Zeitlich beschränkte Zweiwege-Kommunikation (Anmelden bis Abmelden).
- Typischerweise zwischen Client und Server (Request-Response).

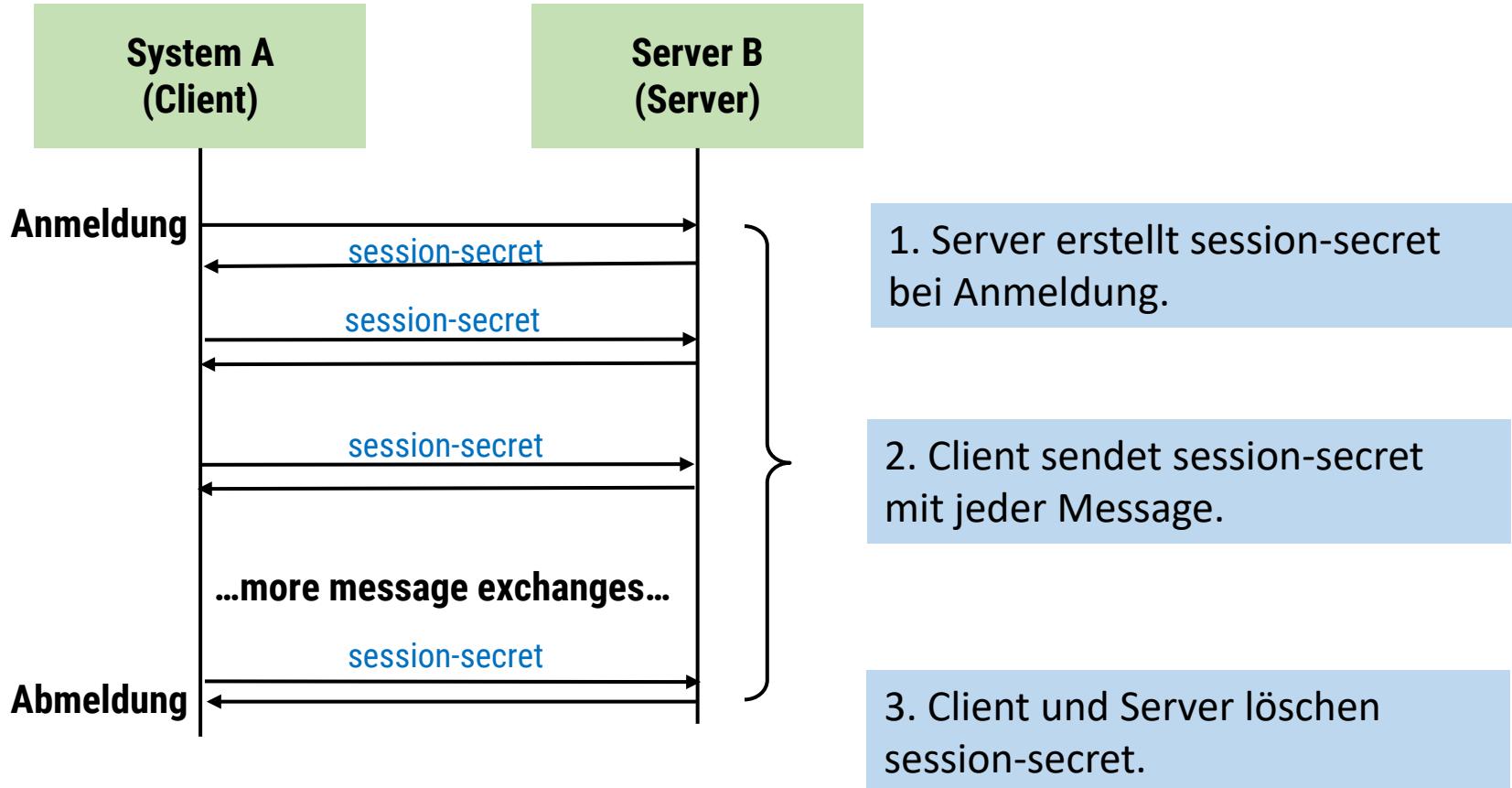


# Beispiel: SMTP-Protokoll (Wiederholung)

SMTP-Client		SMTP-Server
telnet mail.example.com 25	→	
	←	220 service ready
HELO foobar.example.net	→	
	←	250 OK
MAIL FROM:<sender@example.org>	→	
	←	250 OK
RCPT TO:<receiver@example.com>	→	
	←	250 OK
DATA	→	
	←	354 start mail input
From: <sender@example.org>	→	
To: <receiver@example.com>	→	
Subject: Testmail	→	
Date: Thu, 26 Oct 2006 13:10:50 +0200	→	
Lorem ipsum dolor sit amet, consectetur	→	
ut labore et dolore magna aliqua.	→	
.	→	
	←	250 OK
QUIT	→	
	←	221 closing channel

# Session-Secret: Entkopplung der Session von TCP-Connection

- Session-Secret nur dem Server und dem einen Client bekannt.



# Varianten von Session-Secrets

## Zufällige Zahl ohne Informationsgehalt:

- Zufallsgenerator muss kryptographisch sicher sein.
- Typische Größenordnung: 16 bytes.
- Server speichert Informationen, welche zur Session gehören (z.B. Hauptspeicher / Datei / Datenbank /Key-Value Stores (Hazelcast/Redis/etc.).
- **Beispiel:** Session-Cookie in Webapplikationen.

## Verschlüsselte statische Informationen (AccessToken):

- Public/Private-Key Verfahren:
  - Kryptographisch (Signatur) gegenüber Veränderung gesichert.
- **Beispiel:** JSONWebToken
  - dient als AccessToken (enthält z.B. Verknüpfung mit Benutzerkonto).
  - Besteht auf Header/Payload/Signature

## Beispiel: Berechnung eines Session-Secrets (Zufallszahl)

- Für Java ist SecureRandom eine geeignete Implementation:
- Unter Linux bezieht SecureRandom per Default seine Zahlen von /dev/random (blockiert, falls nicht genügend Entropie).

```
private byte[] generateSessionKey() {  
    SecureRandom secureRandom = new SecureRandom();  
    byte[] sessionKey = new byte[16];  
    secureRandom.nextBytes(sessionKey); // may block  
    return sessionKey;  
}
```

# Beispiel: JSONWebToken

Encoded

PASTE A TOKEN HERE

```
eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJzdWIiOiIxMjM0NTY3ODkwIiwibmFtZSI6IkpvaG4gRG9lIiwiaWF0IjoxNTE2MjM5MDIyfQ.Sf1KxwRJSMeKKF2QT4fwpMeJf36P0k6yJV_adQssw5c
```

Decoded

EDIT THE PAYLOAD AND SECRET

HEADER: ALGORITHM & TOKEN TYPE

```
{  
  "alg": "HS256",  
  "typ": "JWT"  
}
```

PAYOUT: DATA

```
{  
  "sub": "1234567890",  
  "name": "John Doe",  
  "iat": 1516239022  
}
```

VERIFY SIGNATURE

```
HMACSHA256(  
  base64UrlEncode(header) + "." +  
  base64UrlEncode(payload),  
  your-256-bit-secret  
)  secret base64 encoded
```

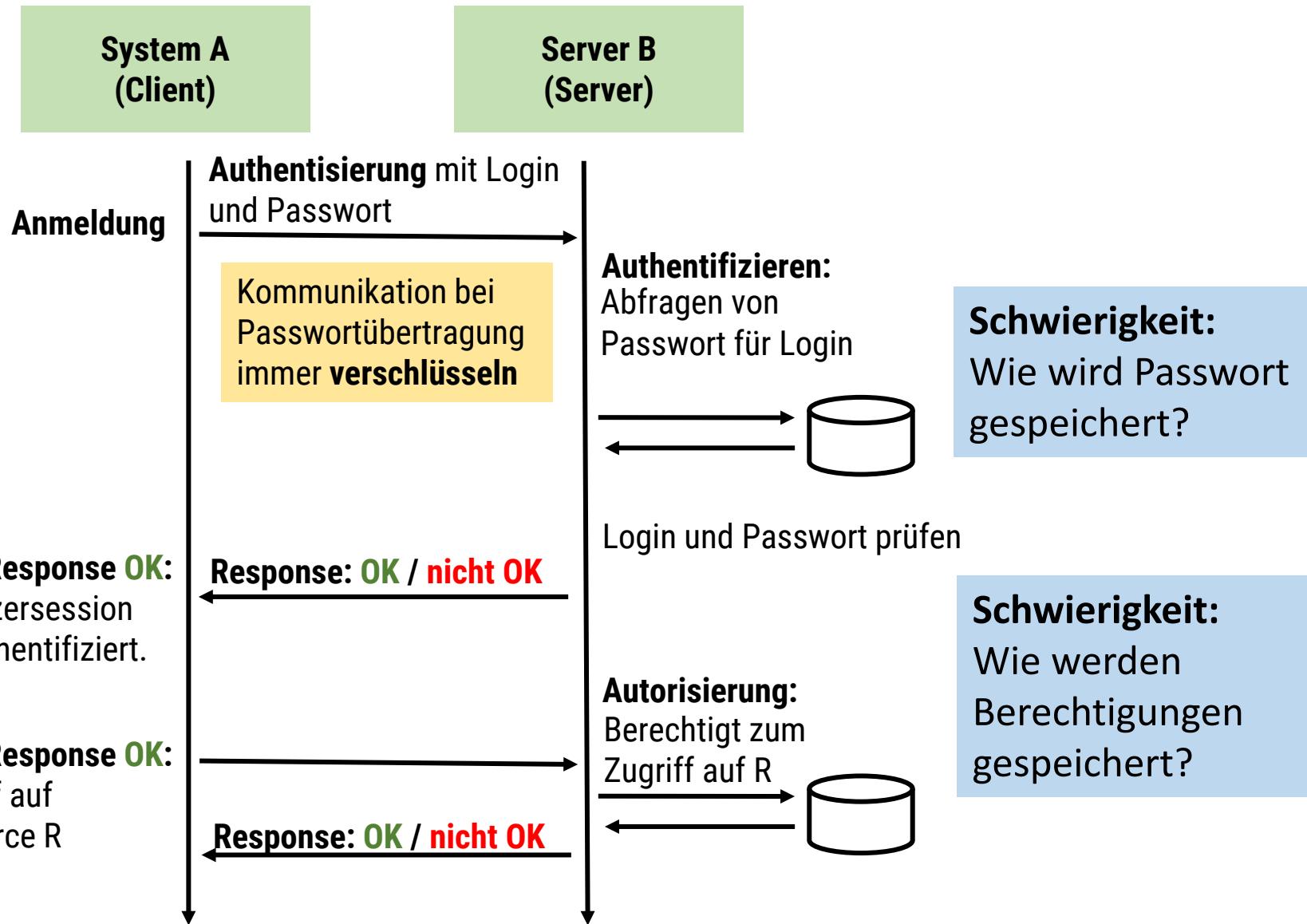
– Quelle: [jwt.io](https://jwt.io)

# **Authentifizierung und Autorisierung**

# Terminologie

- **Authentisieren:** Eine Partei P (Benutzer/in oder System) weisst sich gegenüber einem System S mittels einem Geheimnis aus:
  - I.d.R. mittels Kenntnis einer Information, einem Zugang, oder einem Besitz, welcher nur Partei P hat (Passwort, AccessToken, Smartcard, Empfang einer Email, etc.).
- **Authentifizieren:** System S prüft, ob Partei P diejenige ist, welcher sie vorgibt zu sein, in dem dieses Geheimnis überprüft wird.
- **Autorisierung:** Überprüfen, ob eine authentifizierte Partei berechtigt ist, auf eine Ressource zu zugreifen.

# Authentifizierung einer Session (am Beispiel mit Passwort)



# Authentifizierung einer Session (forts.)

- Authentifiziert wird i.d.R. eine Session, welche für eine Partei P steht.
- Nach erfolgreicher Authentifizierung wird eine Session mit dem Konto der Partei P **verknüpft**.

```
void login(Message message) {  
    String account = message.getAccount();  
    String password = message.getPassword();  
    PasswordRecord record = PasswordManager.getAccount(account);  
  
    if (record.matches(password)) {  
        session.setAccount(account);  
    }  
}
```

Überprüfung des Passworts

Verknüpfung der Benutzersession

- Wesentlicher Teil der Login-Vorgangs ist die **Passwortprüfung**.

# Passwortprüfung

## Passwörter als Hashwert gespeichert:

- In jedes System wird früher oder später eingebrochen!
- Klartext Passwörter:  
Gefahr für [andere Systeme](#).
- Speichen als vom Passwort abgeleiteter Wert, [ein sogenannter Hash](#).
- Kommt ein Angreifer in Besitz eines Hashwerts kennt er das Passwort nicht.

login	hash
anna	\$2y\$10\$bsD5ralzGlwENSF1JY3Wre/JFp9qlIxKldMuuuD3cnk...
joe	\$2y\$10\$ASC9x1HQpq9ruZx/1w7lMebkg4SuQFTubDKcc.ROVvX...
jack	\$2y\$10\$ByfdpPBHYrkhtMPrgQFKOZNJKyt8nzPpDkiTp8HB1m...
alice	\$2y\$10\$km5WnWbOf9zGqSD9.vqDlezRr9R0spSC1v3u/8VJp.G...

## Rad nicht neu erfinden:

- Sichere Technologie stammt aus den 1970ern.
- Vorgefertigte und geprüfte Funktionen und Libraries verwenden:  
Bei Sicherheitslücken wird man informiert und steht nicht alleine da.

# Exkurs: Hashfunktionen

- **Einweg-Funktion:** Aus dem Funktionsresultat lässt sich der Input (das Passwort) nur mit viel Rechenaufwand rekonstruieren:

=>  $f(M1s!ecurePW) = Wre/JFp9qIlxKldMuuuD3cnktDfWR$  **(billig)**

=>  $f_{Rev}(Wre/JFp9qIlxKldMuuuD3cnktDfWR) = M1s!ecurePW$  **(sehr teuer)**

# Geeignete Hashfunktionen

- Keine generische Hash-Funktionen verwenden.
- Diese sind auf Geschwindigkeit optimiert mit beschränkter Lebensdauer:

Function	1990	1991	1992	1993	1994	1995	1996	1997	1998	1999	2000	2001	2002	2003	2004	2005	2006	2007	2008	2009	2010	2011	2012	2013	2014	2015	2016	2017	
Snefru																													
MD2 (128-bit)[1]	Yellow	Yellow	Yellow	Yellow	Orange																								
MD4	Green	Orange	Orange	Orange	Red																								
MD5		Grey	Green	Yellow	Yellow	Orange	[2]	Red																					
RIPEMD																			[2]	Red									
HAVAL-128[1]											Yellow	Yellow	Orange	Orange	Orange	Orange	Orange	Orange	[2]	Red									
SHA-0																			Red										
SHA-1																			Orange	[8]									
RIPEMD-160																				Green	Yellow	Yellow	Yellow						
SHA-2 family																			[4]	Yellow									
SHA-3 (Keccak)																				Grey	Grey	Grey	Grey	Green	Green	Green	Green	Green	Green

Quelle: <http://valerieaurora.org/hash.html> (2017 Valerie Aurora, licensed CC BY-SA)

**Besser:** Passwort-Hashfunktionen verwenden: PBKDF2, Bcrypt, Scrypt.

- Diese sind langsamer in der Berechnung, dies ist aber ein Vorteil.
- PBKDF2 in Kombination mit HMAC-SHA256 empfohlen von NIST [1].

[1] <https://pages.nist.gov/800-63-3/sp800-63b.html#memsecretver>

# Hashfunktionen: Weitere Massnahmen

Beispiel eines Password-Records (hier von PHP):



- **Cost:** Anpassen der Kosten der Hashwertberechnung an aktuelle Hardware.
  - Kosten für ein Login sollte in etwa 100ms sein (immer noch schnell genug für ein Benutzer, aber teuer für Angriffe).
  - Muss jeweils an aktuelle Hardware angepasst werden.
- **Salt:** Verhindert Brute-Force- ("alle Kombinationen durchprobieren") oder Wörterbuch-Attacken.
  - Pro Passwort zufällige Zahl, welche zusammen mit Password «gehasht» wird.
- **Pepper:** (Nicht abgebildet) Zusätzlich alle Passwörter mit weiterer (für alle Passwörter identische) Zahl Z hashen. Diese Zahl Z separat von der Passwort-Db speichern.

# Beispiel: Generierung von Salt und Hash in Java

```
private byte[] generateSalt() {  
    SecureRandom random = new SecureRandom();  
    byte[] salt = new byte[16];  
    random.nextBytes(salt);  
    return salt;  
}
```

Generierung des Salts  
mittels SecureRandom

```
private static final int ITERATION_COUNT = 65536;
```

Einstellung der Kosten

```
private static final int KEY_LENGTH = 512;  
private static final String CRYPTO = "PBKDF2WithHmacSHA512";
```

Hashfunktion  
und Schlüssel-  
länge

```
public byte[] generateHash(String password, byte[] salt) {  
    KeySpec spec = new PBEKeySpec(  
        password.toCharArray(), salt, ITERATION_COUNT, KEY_LENGTH);  
    SecretKeyFactory factory = SecretKeyFactory.getInstance(CRYPTO);  
    return factory.generateSecret(spec).getEncoded();  
}
```

Generierung  
des Hashs

# Beispiel: Erstellung eines Passwortrecords / Passwordprüfung

```
public static class PasswordRecord {  
    private final byte[] hash;  
    private final byte[] salt;  
  
    public PasswordRecord(byte[] hash, byte[] salt) {  
        this.hash = hash;  
        this.salt = salt;    Record enthält mindestens Hash und Salt. Ideal wäre  
    }                                noch Parameter der Hashfunktion zu speichern.
```

```
public PasswordRecord create(String password) {  
    byte[] salt = generateSalt();  
    byte[] hash = generateHash(password, salt);  
    return new PasswordRecord(hash, salt);  
}
```

1. Salt erstellen.
2. Hash mit Password und Salt erstellen.
3. Record mit Hash und Salt zurückgeben.

```
public boolean compare(String password, PasswordRecord record) {  
    byte[] newPasswordHash = generateHash(password, record.salt);  
    return Arrays.equals(newPasswordHash, record.hash);  
}
```

Erstellung eines Passwordhash mit eingegebenem Passwort und Salt aus dem Record. Anschliessend Vergleich.

# Autorisierung

- Nach Verknüpfung mit Konto ist eine Session autorisiert auf bestimmte **Ressourcen** oder **Funktionalitäten** zuzugreifen:

```
if (session.hasRole("logger")) {  
    while (true) {  
        LogMessage msg = receiveLogMsg();  
        ...  
    }  
}
```

Typische Verfahren zur Zugriffskontrolle:

- **Role Based Access Control:** Definition von Rollen (bspw. Verkäufer, Buchhalter, Manager, Mechaniker, ...) und prüfen, ob Benutzersession für die benötigte Rolle die Rechte hat.
- **Row Level Security:** Funktionalität von Datenbanken: Definiert pro Ressource (oft in Table-Row gespeichert), wer darauf zugreifen darf.

# Zusammenfassung

- Sichere Kommunikation in verteilten Systemen: Zugriffsschutz, Manipulationssicherheit, Abhörsicherheit Nachvollziehbarkeit.
- TLS-Protokoll: Transportverschlüsselung implementiert als Anwendungsprotokoll.
- Zertifikate authentifizieren das Zielsystem und werden von vertrauenswürdigen Zertifizierungsstellen erstellt.
- Eine Benutzersession ist eine zeitlich beschränkte Zweiwege-Kommunikation.
- Authentifizierung eines Client findet in der Regel auf Basis einer Benutzersession statt.
- Bei einer Authentifizierung mittels Passwort muss dieses sicher gespeichert werden.
- Mittels Autorisierung wird sichergestellt, dass nur berechtige Benutzer auf Ressourcen und Funktionalität zugreifen.

# Literatur

- Distributed Systems (3rd Edition), Maarten van Steen, Andrew S. Tanenbaum, Verleger: Maarten van Steen (ehemals Pearson Education Inc.), 2017.

# **Fragen?**