Verteilte Systeme und Komponenten

`I.BA_VSK_MM.F25` — Zusammenfassung

Author(s) Dominic, Elias, Hannah, Ivo, Joël, Laura, Lukas, Niklas
Date 16. Mar. 2026
Pages 146

Inhaltsverzeichnis

1. Clean Code 🧹

1.1. Kommentare

TL;DR: Der beste Kommentar ist derjenige, welchen man gar nicht schreibt

⇒ Energie lieber in guten, selbsterklärenden Code stecken.

- Statt einen Kommentar zu schreiben, lieber die unklare Code-Stelle umschreiben
- Die eigentliche Wahrheit liegt im Code, nicht im Kommentar
- Akzeptierbare Kommentare sind
  - juristische Kommentare (z. B. Copyright)
  - `TODO` -Kommentare (jedoch nur temporär)
  - Kommentare zum Hervorheben von Inhalten, welche ansonsten untergehen würden
  - Warnungen vor Konsequenzen
  - Erklärung einer übergeordneten Absicht
- Schlechte Kommentare

  Redundant Einfache Wiederholung einer Aussage, welche bereits im Code steht
  Irreführend falsch oder unpräzise
  vorgeschrieben / erzwungen Kommentieren zur Vollständigkeit ohne tiefere Aussage
  Changelog-Kommentare Diese Aufgabe übernimmt unser VCS für uns
  Optische Kommentare Kommentare, welche lediglich zur visuellen Unterteilung von grossen Quellcodedateien verwendet werden
  Nebenbemerkungen Anmerkungen des Autors, mehrwertfreie Zusatzbemerkungen
  Auskommentierter Code 🚨 Auch das löst unser VCS für uns
  HTML-Kommentare Wir sollten Kommentare ohne weitere Hilfsmittel lesen können
  zu viele Informationen Zusatzinfos, welche keinen Mehrwert bieten

1.2. Namensgebung

TL;DR: Genügend Zeit in die Namensfindung investieren, **sprechende, korrekte, eindeutige und suchbare** Namen verwenden.

- Gute Namensgebung ist schwierig
- Klassen- und vor allem Interfacenamen in Zukunft nicht trivial umzubenennen
- Anforderungen an einen guten Namen
  - Zweckbeschreiben
  - keine Fehlinformationen
  - Differenzierbarkeit von anderen
  - gut aussprechbar und suchbar
  - keine Codierung enthalten
  - korrekte Rechtschreibung

1.2.1. Namensgebungsheuristiken

1. Beschreibende Namen wählen
2. Namen passend zur Abstraktionsebene wählen
3. Standardnomenklatur verwenden
4. Eindeutige Namen wählen
5. Namenslänge abhängig vom Geltungsbereich
6. Codierungen vermeiden
7. Namen sollten auch Nebeneffekte beschreiben

1.3. „Clean Code Developer“

TL;DR: Wiederkehrendes prüfen und optimieren auf Basis der gängigen Clean Code-Prinzipien führt zu stetig steigender Code-Qualität.

- Auswahl von 42 Punkten aus Clean Code und co.
- Gruppierung in 7 Gruppen
- Anspielung an die 7 Gürtel in Judo
- Jeder grad fokussiert auf eine überschaubare Menge von Prinzipien
- Aufteilung immer in **Prinzipien** und **Praktiken**

1.3.1. Schwarz ⚫

Sie wissen, was Clean Code Developer ist, und haben damit den ersten, schwarzen Grad erreicht!

1.3.2. 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

1.3.3. 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

1.3.4. 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

1.3.5. 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

1.3.6. 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

1.3.7. Weiss 🤍
- Vereinigt alle Prinzipien und Praktiken der Grade
- Zyklisches durchlaufen der Farbigen Grade

1.4. Praxistipps

TL;DR: Just do it.

- Regelmässige Reviews durchführen
- Offene und vertraute Atmosphäre
- Erfahrungen und Wissen im Bereich Clean Code an andere Weitergeben
- Clean Code fördernde Werkzeuge einsetzen
  - Checkstyle, PMD, Spotbugs (Namenslänge, Designprinzipien, Abhängigkeiten, ...)
  - SonarQube (Statistiken zur Verbesserung der Code-Qualität)
  - Messungen der Codeabdeckung (JaCoCo, Clover, ...)

2. Sockets und RPC 🧦

2.1. RPC

- Funktion auf entferntem System analog zu einer auf dem lokalen System aufrufen können
- Stubs (Stummel) als Endpunkt zur Kommunikation

  Client Stub Serialisierung der Daten zur Übertragung über das Netzwerk; Umwandlung von *interner Datenstruktur* zu *Bytetrom*

  Server Stub Deserialisierung der empfangenen Daten; Umwandlung von *Byte Strom* zu *interner Datenstruktur*
- Netzwerkübertragung und somit Format des Bytestroms über gemeinsames Kommunikationsprotokoll definiert
- Bei der (De)Serialisierung ist Endian wichtig!

2.2. Kommunikationsprotokolle

- Vereinbarung über Format der Datenübertragung
- in der Regel auf Anwendungsschicht
- Bestandteile sind
  1. Übertragungskanal definieren, welche Transportschicht (TCP/UDP), wie diese konfiguriert wird und weitere Anforderungen an den Kanal wie Bandbreite oder Latenz
  2. Definition der Nachrichtenstruktur, also welches Format, Länge, Stoppzeichen, Fehlercodes, ...
  3. Definition der möglichen Nachrichten, Datenstrukturen, Datenformate und Zustände der Kommunikation; die allgemeine Nachrichtenstruktur ist dabei gleich, der Inhalt unterscheidet sich jedoch.

2.2.1. Nachrichtenstruktur

Im Allgemeinen zwei Arten von Strukturen, entweder mit Längenangabe oder mit Stoppzeichen:

```
Längenangabe:
+-----------+----+--------+
| Länge (n) | ID | Inhalt |
+-----------+----+--------+
    a bytes   |         |
               + b bytes |
                         + (n-a-b) bytes

Stoppzeichen:
+----+----------------------------+--------------+
| ID | Inhalt (ohne Stoppzeichen) | Stoppzeichen |
+----+----------------------------+--------------+
  a bytes    |                          |
             + b bytes                  |
                                        + c bytes
```

2.2.2. Beispiel HTTP-Response

Bei der HTTP-Response handelt es sich um eine Kombination aus Längenangabe und Stoppzeichen.

Ein Stoppzeichen (2 x Newline) trennt hier den Header vom Body, wobei der Body dann durch die `Content-length` beschränkt ist.

```
HTTP/1.1 200 OK
Server: ExperimentalWebServer 1.0
Content-type: text/html
Content-length: 143

<html>
```

```
<head><title>Anzeige von Bildern</title></head>
<body background="pictures/bg.gif">
<img src="pictures/logo.jpg">
</body>
</html>
```

2.2.3. Nachrichtendefinition

- Machen Grossteil eines Komm. Protokolle aus
- Folgen einem allgemein für dieses Protokoll definierten Nachrichtenformats
- Nachrichtenstruktur hat in der Regel folgenden Aufbau
  - ID
    - Identifiziert die Nachricht (z.B. GET, POST, … bei HTTP)
    - Bei zustandsbasierten Protokollen teilweise nicht benötigt
  - Argumente
    - Einfache Datentypen (Integer, Strings, Floating-Point).
    - Strukturen oder Arrays.
    - Referenzen.

2.3. Sockets

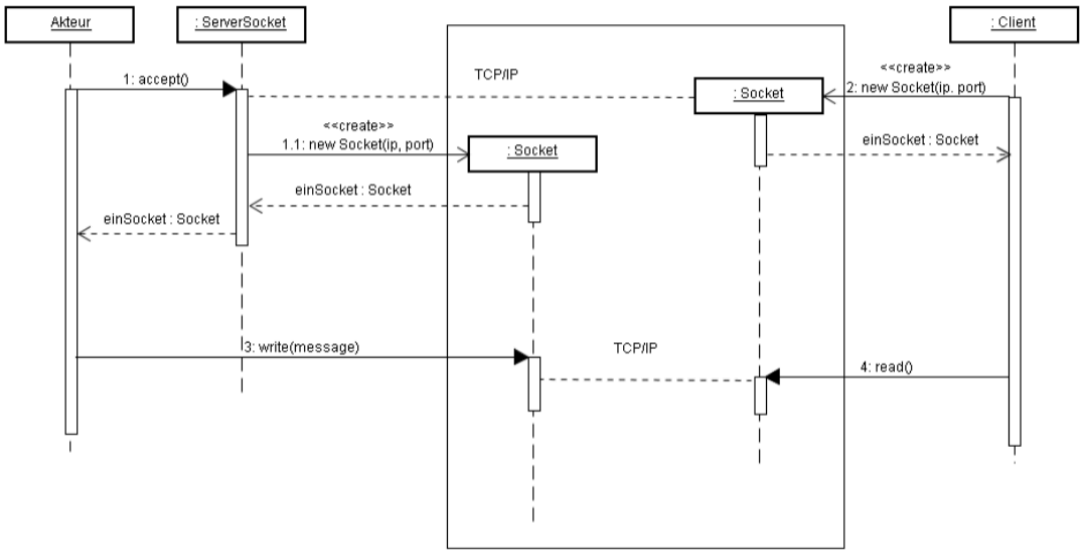


Abbildung 1: Sequenzdiagramm: Verbindungsaufbau mit TCP-Sockets

Socket Kommunikationsendpunkt im TCP/IP-Netzwerk
- Besteht aus IP + Port

Server Socket Socket auf der Serverseite, welcher auf eingehende Verbindungen hört

Für Ports ist dabei wichtig, dass

- eine Postnummer nicht bereits in Verwendung ist
- nach Benutzung für z. B. 4 Minuten nicht wieder verwendet werden kann, da evtl. noch Nachrichten des vorherigen Clients auf diese Portnummer kommen könnten (Sliding Window)
- Auf unix-Systemen können Portnummern < 1024 nur von privilegierten Accounts verwendet werden

2.3.1. Client-Socket erstellen

1. Socket erstellen
2. Auf lokalen, serverseitigen Port binden (in der Regel zufälliger Port > 1024)
3. Verbindung mit Zieladresse aufbauen
4. Daten über Socket übertragen
5. Socket schliessen

2.3.2. Server-Socket erstellen

1. Server-Socket erzeugen
2. Mit `accept()`-Methode (Java ☕) auf Verbindungen warten
3. Input- / Output-Stream mit Socket verknüpfen
4. Daten entsprechend dem Kommunikationsprotokoll lesen / schreiben
5. Stream schliessen
6. zu 2. springen oder Socket schliessen

2.3.3. Beispiel in Java

- `accept()` Methode der `ServerSocket`-Klasse nimmt genau eine wartende Verbindung an und blockiert den Programmablauf
- Kommunikation läuft über den von der `accept()`-Methode retournierten Socket
- Um wieder verfügbar für neue Verbindungen zu werden, muss das Programm wieder zur `accept()`-Methode gelangen
- Nebenläufige Ausführung zur besseren Skalierbarkeit naheliegend

Client

```java
static void getTime(String host, int port) throws IOException {
  Socket socket = new Socket(host, port); // Socketverbindung starten

  DataInputStream is;
  is = new DataInputStream(socket.getInputStream()); // Input-Stream öffnen

  byte[] bytes = is.readAllBytes(); // Bytes einlesen
  socket.close(); // Socket schliessen

  String time = new String(bytes);
  System.out.println(time);
}
```

Server

```java
public class SimpleDayTimeServer {
  //...
  public static void main(final String[] args) {
    try {
      final ServerSocket listen = new ServerSocket(1300); // Bind to Port 3000
      while (true) {
        try (final Socket client = listen.accept()) { // Wait for client
          final DataOutputStream dout =
            new DataOutputStream(client.getOutputStream());
          final Date date = new Date();
          dout.write((date.toString()).getBytes()); // Transmit time
        } // Close Socket via try-with-ressources
      }
    } catch (IOException ex) {
      LOG.debug(ex.getMessage());
    }
  }
}
```

2.4. Parallele Verarbeitung

2.4.1. Blocking I/O

Wartet ein Thread auf Daten, so ist seine Ausführung blockiert:

```java
DataInputStream is;
is = new DataInputStream(socket.getInputStream());
byte[] bytes = is.readAllBytes(); // Blockiert die weitere Ausführung
```

2.4.2. Varianten der parallelen Verarbeitung

Es gibt folgende unterschiedliche Möglichkeiten zur parallelen Verarbeitung.

Blocking I/O + Single-Threaded Eine langlebige Verbindung oder wenige kurzlebige Verbindungen.
Blocking I/O + mehrere Prozesse (Fork) Wenige langlebige Verbindungen.
Blocking I/O + mehrere Threads Wenige langlebige Verbindungen.
Blocking I/O + Thread-Pools Viele kurzlebige Verbindungen.
Non-Blocking I/O + Single-Threaded Viele Verbindungen (kurz- oder langlebig) mit wenig Bearbeitungszeit.
Non-Blocking I/O + mehrere Threads Viele Verbindungen (kurz- oder langlebig) mit wenig bis viel Bearbeitungszeit.
Blocking I/O + mehrere virtuelle Threads Viele Verbindungen (kurz- oder langlebig) mit wenig bis viel Bearbeitungszeit.

2.4.3. Java Beispiel

Nachfolgend ist das Beispiel eines parallelen EchoServers aufgeführt:

2.4.3.1. EchoServer

Der `EchoServer` startet einen `EchoHandler` für jede Ankommende Request.

```java
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.newVirtualThreadPerTaskExecutor();

    while (true) {
      try {
        LOG.info("Waiting for connection...");
        final Socket client = listen.accept();
        final EchoHandler handler = new EchoHandler(client);
        executor.execute(handler); // Dispatching an EchoHandler
      } catch (Exception ex) {
        LOG.debug(ex.getMessage());
      }
    }
  }
}
```

2.4.3.2. Echo-Handler

- Verarbeitung eines ankommenden Requests
- Handler läuft unabhängig von anderen laufende Threads

```java
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;
  }
```

```
  @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());
    }
  }
}
```

2.4.3.3. Echo Client

```
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 (...) {
      String messageOut = userIn.readLine();
      sendMessage(dataOut, messageOut);
      dataOut.flush();
      String messageIn = getMessage(dataIn);
      System.out.println(messageIn);
    }
  } catch (Exception ex) {
    LOG.debug(ex.getMessage());
  }
}
```

2.4.3.4. Kommunikationsprotokoll

Es wird ein einfaches Kommunikationsprotokoll nach folgendem Schema verwendet.

```
+-----------+---------------------+
| Länge (n) | Message (Raw UTF-8) |
+-----------+---------------------+
   a bytes        (n-a) bytes
```

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

private static String getMessage(DataInputStream is) throws IOException {
  int length = is.readInt();
  byte[] bytesIn = new byte[length];
  is.readFully(bytesIn);
  return new String(bytesIn, StandardCharsets.UTF_8);
}
```

2.5. gRPC

- Plattform und sprachübergreifendes RPC-Framework
- Verwendet **Google Protocol Buffers**
- Basiert auf HTTP/2

2.5.1. Protocol Buffers

- Serialisierung und Deserialisierung von strukturierten Daten
- Eigene Sprache mit Dateiendung `.proto`

```proto
syntax = "proto3";      // Version, default Version 2

service SearchService { // Service mit einem oder mehreren rpcs
  rpc Search(SearchRequest) returns (SearchResponse);
}

message SearchRequest { // Nachrichtenstruktur
  string query = 1;     // Typisierte Felder inkl. ID
  int32 page_number = 2;
  int32 result_per_page = 3;
}

message SearchResponse { ... }
```

2.5.2. gRPC Typen

.proto	C++	Java/Kotlin	Python	Go	C#	PHP	Dart
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

.proto	C++	Java/Kotlin	Python	Go	C#	PHP	Dart
bool	bool	boolean	bool	bool	bool	boolean	bool
string	string	String	str/unicode	string	string	string	String

2.5.3. Workflow

1. Datenstruktur in `.proto`-File definieren
2. Code mittels `protoc`-Compiler generieren
3. Generierter Code Projekt verwenden

`protoc --java_out=<some-dir> path/to/file.proto`

Für Java werden

- eine `.java`-Datei mit einer Klasse pro Nachrichtentyp
- eine spezielle Builder-Klasse zur Erzeugung der Nachrichten

generiert.

2.5.4. Echo-Service mit gRPC

2.5.4.1. Protokolldefinition

```proto
syntax = "proto3";

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

message EchoRequest {
  string message = 1;
}

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

2.5.4.2. gRPC Echo Server

```java
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> observe
    ) {
      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();
  }
}
```

2.5.4.3. gRPC Echo-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();
  }
}
```

3. Komponentenbegriff

3.1. Konzept der Software-Komponenten

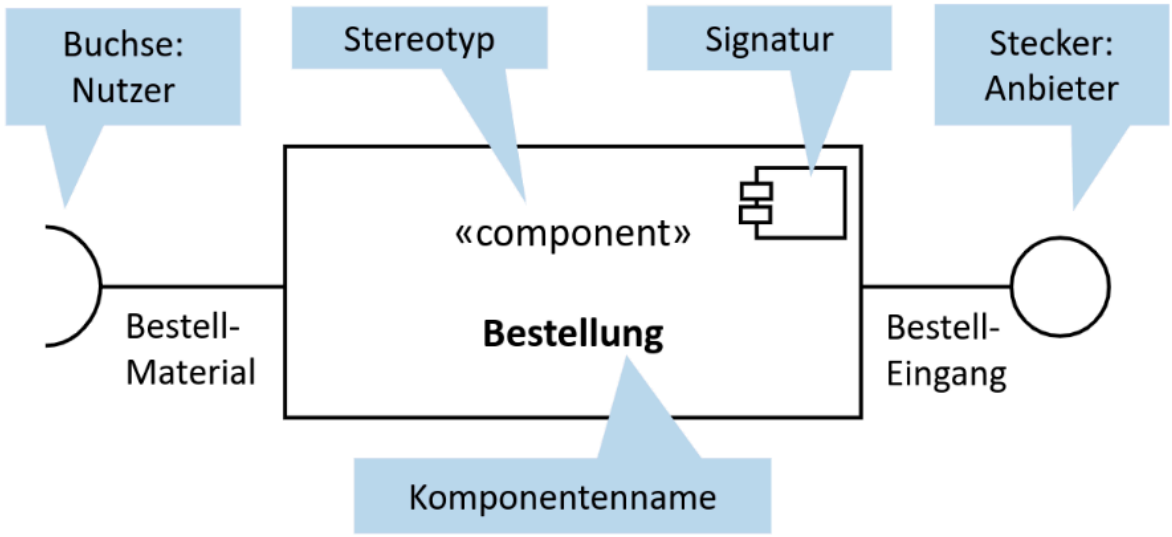
Eine **Software-Komponente** ist eine eigenständige, wiederverwendbare Softwareeinheit mit klar definierten Schnittstellen und expliziten Kontextabhängigkeiten. Sie kann unabhängig entwickelt, getestet und in verschiedene Systeme integriert werden.

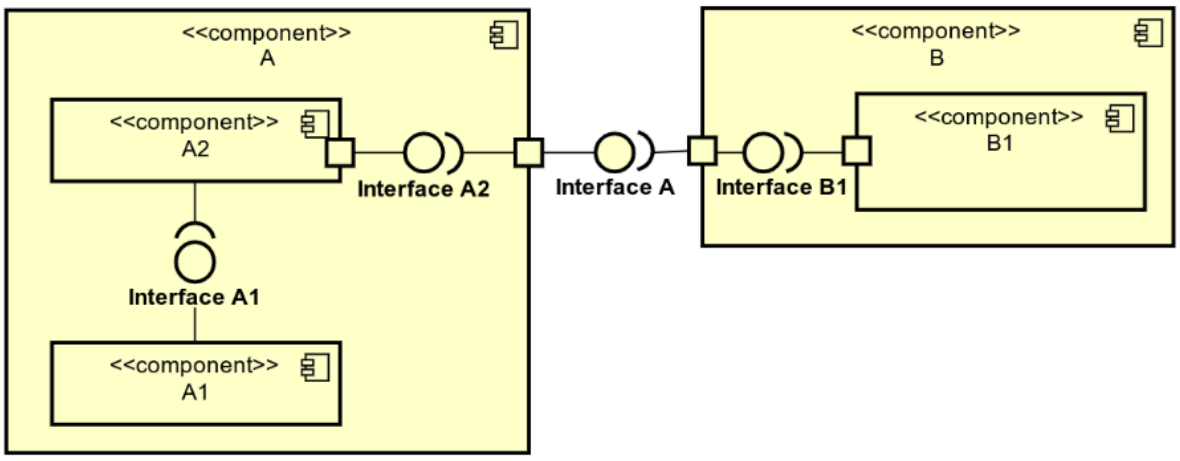
Eigenschaften:

- **Eigenständigkeit**: Komponenten sind ausführbare Einheiten, die unabhängig funktionieren können.
- **Schnittstellenbasiert**: Kommunikation erfolgt über klar definierte Schnittstellen.
- **Wiederverwendbarkeit**: Komponenten können in verschiedenen Anwendungen eingesetzt werden.
- **Austauschbarkeit**: Komponenten können durch andere mit gleicher Schnittstelle ersetzt werden.

3.1.1. Modelieren im UML

Komponenten können wie folgt in UML modelliert werden:





- 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).

3.1.2. 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.

Beispiel:

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

3.1.3. Interaction Standard

- Beschreibt den Schnittstellenstandard, also wie Komponenten innerhalb eines Komponentenmodells miteinander kommunizieren
  - 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
  - Interface Definition Language (CORBA)
  - WSDL (SOAP)

3.1.4. 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.).

3.2. 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 Auswirkung von Veränderung

Geringere Änderungsauswirkung

- Änderungen wirken sich nur auf die unmittelbare Komponenten aus und nicht auf das gesammte System
- Bei Monolythischer Applikation viel weitreichendere Folgen

3.3. Entwurf mittels Komponenten

3.3.1. Spezifikationen von Komponenten

Export unterstützte Interfaces, die andere Komponenten nutzen können
Import benötigte / benutzte Interfaces von anderen Komponenten
Verhalten Verhalten der Komponente
Kontext Rahmenbedingungen für Betrieb der Komponente

3.3.2. Verhaltenssicht

- Systemverhalten auf höherer Flughöhe gut duch Kompoonenten darstellbar

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.

4. Schnittstellen

4.1. Begriff und Konzept

- An Stelle wo Komponenten zueinander geführt werden, müssen diese aufeinander passen
- Es werden Verbinungsstellen konstruiert, welche diese Kombinierbarkeit sicherstellen
- Eine Schnittstelle an sich tut und kan nnichts
- Sie verbinden
  - Komponenten untereinander
  - Komponenten mit dem Benutzer (Benutzerschnittstelle)

4.2. Schnittstellenkonzept

Ein Schnittstellenkonzept versucht folgende Punkte zu erfüllen:

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.

4.3. Schnittstellenbreite

Die *Schnittstellenbreite* 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!

4.4. Kriterien für gute Schnittstellen

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

Beispiel:

```java
interface Modem {
  void dial(string number)
  void hangup()
  void send(char data)
  char receive()
}
```

Durch Aufteilung in kleinere Schnittstellen kann die Komplexität reduziert werden:

```java
interface Connection {
  void dial(string number)
  void hangup()
}

interface Transmit {
  void send(char data)
  char receive()
}
```

4.5. Design by Contract

Das Zusammenspiel von zwei Komponenten wird durch einen Vertrag (die Schnittstelle) erreicht, welcher aus folgenden Teilen Besteht:

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)

4.5.1. Verantwortlichkeiten

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

4.5.1.1. Beispiel mit Pre- und Postconditions

```java
public class LinkedList {
  private static class Node {
    int value;
    Node prev, next;
    LinkedList list;
  }

  // Nachfolgend Klasseninvarianz (INV)
  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(...);
    ...
}
```

4.6. Spezifikation von Schnittstellen

Zur Schnittstelle gehört

- was für die Benutzung der Komponente wichtig ist
- was der Programmierer verstehen und beachten muss

Jede Schnittstelle definiert Menge von Operationen, welche folgende Eigenschaften aufweisen:

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

Bei der Art *wie* das ganze Dokumentiert wird, ist auf folgendes zu achten:

Syntax Notation analog zur verwendeten Programmiersprache.

Semantik Präzise und kompakt textuell beschreiben, ggf. unter zu Hilfenahme formaler math. Konstrukte. Keine allgemein akzeptierte Art der Dokumentation.

Eingabeparameter Welche Informationen werden an die Komponente weitergeleitet

Ausgabeparameter Welche Informationen die Komponenten zurückliefert

Zustand Zustandsänderung der Komponenten

Spezifikation inwiefern sich Eingabeparameter auf die Zustandsänderung und die Ausgabeparameter auswirken

4.6.1. Schnittstellen in Java

- Deklaration mit *Java Interfaces*
- Werden zu `.class`-Dateien kompiliert
- Können in *JAR*-Dateien verpackt und verteilt werden
- Dokumentation mittels Java-Doc essentiell
- Schnittstellen an der Systemgrenze sowie zwischen Subsystemen werden in der **Architekturbeschreibung dokumentiert**
- Öffentliche Schnittstellen bezeichnet man oft als APIs, welche auch aus mehreren Interfaces bestehen können

4.6.2. APIs

- Spezifiziert Operationen sowie Ein- und Ausgabe einer Softwarekomponente
- Hauptzweck sit menge von FUnktionen unabhängig ihrer Implementation zu definieren
- Implementation kann ohne Beeinflussung von Benutzer

API ist dabei ein Oberbegriff mit unterschiedlichen Unterkategorien:

- **API**
  - **Objektorienterte API**
    - Sprachabhängig
    - z. B. die Java-Schnittstelle einer Bibliothek
  - **REST-API**
    - sprach- und plattformunabhängig
    - Daten stehen im Zentrum
    - z. B. RESTful HTTP
  - **Messaging-API**
    - sprach- und plattformunabhängig
    - Entkopplung
    - z. B. Push-notifications für mobile Applikationen
  - **Dateibasierte API**
    - asynchroner Informationsaustausch
    - z. B. Konfigurationsdateien

5. Versionskontrolle

5.1. Grundlagen eines Versionskontrollsystems (VCS)

- Hauptziel: Nachvollziehbarkeit aller Änderungen an Dateien (Artefakten).
- Anwendungsbereich: primär Softwareentwicklung (Source Code Management - SCM).
- Abgrenzung zu Filesharing-Diensten (Dropbox, OneDrive): VCS hat bewusstes Versionieren und Nachvollziehbarkeit im Fokus, nicht automatische Synchronisation.

5.2. Arbeitsweise mit VCS

5.2.1. Zentrale Befehle

`checkout` lokale Arbeitskopie erstellen.
`update` Änderungen Dritter aktualisieren.
`commit` lokale Änderungen in Repository schreiben, stets mit aussagekräftigem Kommentar.
`log` Änderungsgeschichte ansehen.
`diff` Revisionen vergleichen.

5.2.2. Verteilte Befehle (z.B. git)

`clone` Repository lokal klonen.
`pull` Änderungen vom entfernten Repository übernehmen.
`push` lokale Änderungen auf entferntes Repository übertragen.

5.2.3. Erweiterte Konzepte

- **Tagging**: Markierung von spezifischen Revisionsständen (z.B. Releases).
- **Branching**: Erstellung separater Entwicklungszweige für parallele Entwicklung (z.B. Feature-Entwicklung, Bugfixes).
- **Wichtig**: Niemals generierte Dateien (z.B. `*.class`) einchecken; Nutzung von .gitignore.

5.3. Unterschiedliche Konzepte und Produkte

- Zentralisierte VCS (z.B. CVS, SVN): Zentraler Server, Transaktionen (bei SVN), branch- und tag-Struktur über Kopien.
- Verteilte VCS (git): verteiltes Repository-Modell, schnelle lokale Operationen, flexibles Branching.

5.4. Git als modernes, verbreitetes VCS

- Verteiltes System, entwickelt von Linus Torvalds, geeignet für umfangreiches Branching und Teamarbeit.
- Erfordert solides Verständnis bei verteiltem Einsatz.

5.5. Benutzerschnittstellen

- **Kommandozeile**: schnell und effizient, aber voraussetzungsreich.
- **GUI-Clients (z.B. SmartGit, SourceTree)**: benutzerfreundlich, bieten einfache Übersicht.
- **Integrierte IDE-Clients**: komfortabel, aber ggf. komplexer.

5.6. GitLab (Switch)

- Vollständige Codehosting-Plattform inkl. Issue-Tracking, CI/CD und Wiki.
- Zugriff via https oder ssh möglich (eduID für Login empfohlen).
- Beliebte Clients: SmartGit (Empfehlung), SourceTree, Eclipse (EGit), NetBeans.

5.7. Empfohlene Praxis bei der Nutzung von git

- Vor Commits/Pull-Requests stets pull durchführen.
- Commits klar kommentieren (optional Issue-Referenz).
- Nutzung von Conventional Commits zur besseren Automatisierung empfohlen.
- Diese Zusammenfassung enthält die wesentlichen Aspekte der Versionskontrolle, um optimal auf die Prüfung vorbereitet zu sein.

6. Buildautomation

6.1. Grundlagen der Buildautomatisierung

- **Hauptziel**: Automatisierung repetitiver Aufgaben bei der Softwareerstellung zur Steigerung von Effizienz und Reproduzierbarkeit.
- Buildprozess umfasst: Kompilieren, Testen, Erstellen von Distributionen und Deployment.

6.2. Automatisation des Buildprozesses

- Man möchte ein Script schreiben, dass notwendige Tools mit allen Parametern ausführt
- **Vorteile**
  - Effizienz und Zeitersparnis
  - Vermeidung manueller Fehler
  - Reproduzierbarkeit der Builds
  - Unabhängigkeit von Entwicklungsumgebungen
- **Nachteile**
  - Eher stur und unflexibel
  - Abhängigkeit von Shell und Platform
  - Aufwändige Wartung und Erweiterung

6.3. Buildwerkzeuge

Make Ursprüngliches Build-Werkzeug, häufig für C/C++ verwendet, flexibel aber plattformabhängig.
Ant Java-basiertes Tool mit XML-Skript, älter, aber bewährt.
Apache Maven Sehr populär, XML-basiert, deklarativer Ansatz.
Gradle Modern, benutzt Groovy und eine DSL, flexibel und leistungsfähig.
Bazel Von Google entwickelt, verwendet Python-ähnliche Skriptsprache.

6.4. Apache Maven

Deklaratives Projektmanagement Verwendung des Project Object Model (`pom.xml`).
Lifecycle und Goals Klar definierte Build-Phasen (validate, compile, test, package, verify, install, deploy).
Plugin-Architektur Modular und erweiterbar, Plugins führen konkrete Aufgaben aus.
Dependency Management Zentrale Verwaltung und automatische Auflösung von Abhängigkeiten über Repositories.
- Zentral: Maven Central Repository
- Lokal: `$HOME/.m2/repository`

Multimodul-Projekte Unterstützt Strukturierung komplexer Projekte durch hierarchische Module.
Lifecycle und Goals Allgemeiner Ablauf des Buildprozesses, in welche sich Plugins einhängen können.
`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.

6.5. Anwendung und Praxis mit Maven

- Maven Integration in Entwicklungsumgebungen (Eclipse, IntelliJ, NetBeans).
- Empfohlene Nutzung: eigenständige Installation außerhalb der IDE für maximale Flexibilität und Unabhängigkeit.

6.6. Weiterführende Hinweise

- Verwendung von CI/CD-Pipelines mit Maven (z.B. GitLab CI).
- Praktische Erfahrung sammeln und Maven-Dokumentation nutzen zur Vertiefung.

7. Messageorientierte Kommunikation

Kommunikation mittels expliziter Messages gesendet von einem Sender zu einem oder mehreren Empfängern.

Verwendung:
- Nebenläufige und parallele Programmierung (z.B. Actor-Model
- Interprozesskommunikation (z.B. Unix-Sockets).
- Kommunikation zwischen verteilten Systemen.

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

7.1. Eigenschaften

- **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)

Ausführungszeitpunkt: Synchron vs asynchron

Sicherheit: Offen vs Zugangsgeschützt, Verschlüsselung

7.2. synchrone Kommunikation

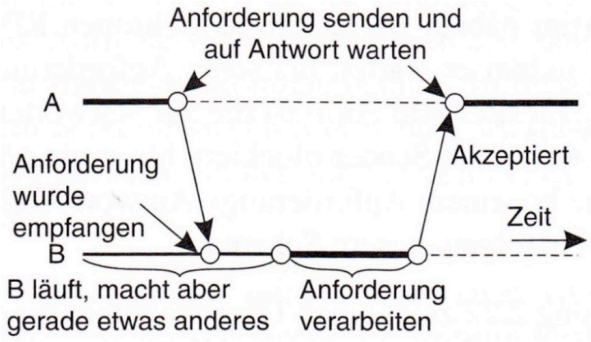


Abbildung 2: synchrone Kommunikation

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

7.3. asynchrone Kommunikation

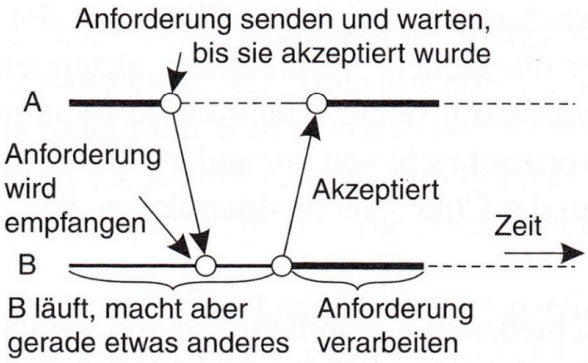


Abbildung 3: synchrone Kommunikation

Beispiele

- Asynchroner Remote-Procedure-Call
- Anfrage einer Ressource mittels HTTP-Protokoll mit Quittierung (ACK)

Bei beiden: Wie kommt das Resultat zurück?

7.4. Persistente vs. transiente Kommunikation

Persistente Kommunikation

- Nachricht wird persistent gespeichert bis Empfänger bereit ist.
- Bsp. E-Mail

transiente Kommunikation

- Nachricht nur flüchtig gespeichert.
- Nachricht nur verfügbar, solange sendende und empfangende Applikation ausgeführt werden
- Bsp. Router, Socket

7.5. Kommunikationsmuster

- Request Reply: Client-Server (1:1)
- Publish Subscribe: Datenverteilung (1:n)
- Pipeline: Verarbeitung in mehreren Schritten (1:n)

7.5.1. Request Reply

Ziel: Verbindung einer Menge von Clients mit einer Menge von Services

- Half-Duplex: Nur Client kann Nachrichten initiieren (Normalfall)
- Full-Duplex: Sowohl Client als auch Service können Nachrichten initiieren

Hinweis: Half-Duplex: In der Kommunikationstechnologie bedeutet das, dass nur in eine Richtung gleichzeitig kommuniziert werden kann, aber die Initiierung ist egal

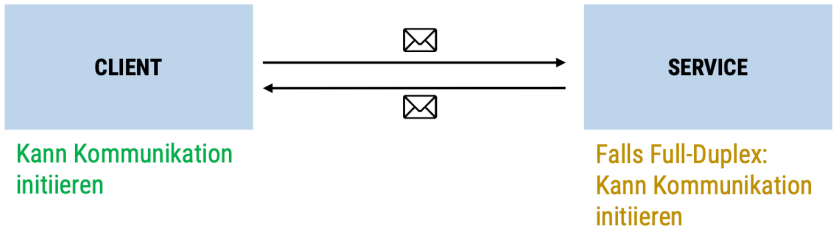


Abbildung 4: Request Reply Kommunikationsmuster

7.5.2. Publish Subscribe

Ziel: Datenverteilung

- Verbindet eine Menge von Publishers mit einer Menge von Subscribers
- Verbreitet im Enterprise- (ActiveMQ/Websphere) und IoT-Umfeld (-> MQTT)

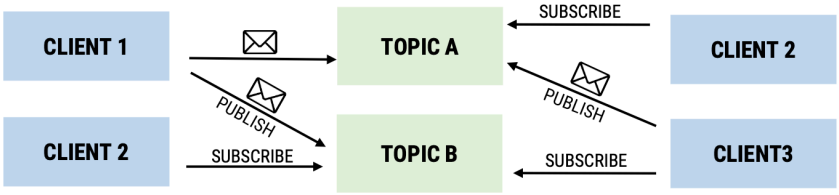


Abbildung 5: Publish Subscribe Kommunikationsmuster

Beispiele:

- eine Adressänderung fand statt. Nun muss die Adresse in allen System eines Unternehmens angepasst werden
- Sensor der die Temperatur misst. Anzeigesystem, Speichersystem usw.

Nachrichten können einem Topic zugeordnet werden. Clients können sich einem Topic subscriben (wenn subscribed erhält er Nachricht, sonst nicht).

Wichtig: es gibt keinen Rückkanal (nur von Client zu Topic)

7.5.3. Pipeline / producer-consumer

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

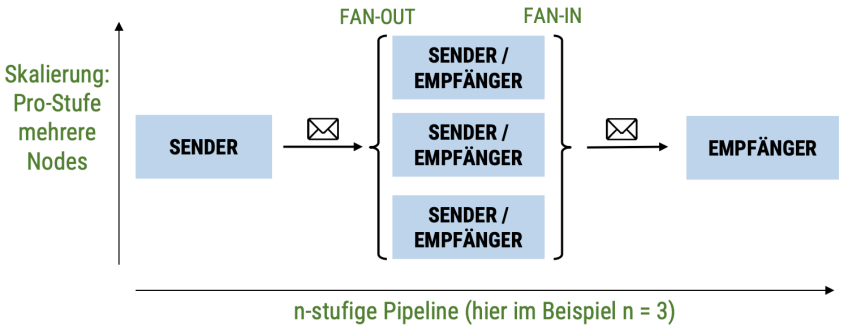


Abbildung 6: Pipeline Kommunikationsmuster

7.5.3.1. Fan-in / Fan-out

Aufgaben werden parallel aufgeteilt (**Fan-out**) und die Ergebnisse später zusammengeführt (**Fan-in**).

7.6. Technologie: Zero MQ

- Unter anderm eine Java-Library für messageorientierte Kommunikation
- Messages werden in der Regel **asynchron** gesendet
- **Transient**: Messages werden nur im flüchtigen Speicher vorgehalten
- Message-Inhalt beliebig (Binär oder Text)
- Framing basierend auf Länge

7.6.1. Sockets

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).

7.6.2. Implementationen

7.6.2.1. Half-Duplex / Request Reply

7.6.2.1.1. Client

```java
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);

        // REQ: Socket für Requests erstellen
        ZMQ.Socket socket = context.createSocket(SocketType.REQ);

        // Clients benutzen connect, um sich mit dem Server zu verbinden
        socket.connect(ADDRESS);

        BufferedReader in = new BufferedReader(new InputStreamReader(System.in));
        while (...) { // Schleife für kontinuierliche Eingabe
            String input = in.readLine();

            // REQ/REP: Auf jedes send muss ein recv folgen
            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());
    }
}
```

7.6.2.1.2. Server

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

public static void main(String[] args) {
    try (ZContext context = new ZContext()) {
        // REP: Socket für Replies erstellen
        ZMQ.Socket socket = context.createSocket(SocketType.REP);

        // Server benutzen bind, um auf Anfragen zu warten
        socket.bind(ADDRESS);

        while (...) { // Endlosschleife zur Verarbeitung von Anfragen
            // REQ/REP: Auf jedes recv muss ein send folgen
            byte[] reply = socket.recv();
            LOG.info("Received message " + new String(reply, ZMQ.CHARSET));
            socket.send(reply);
        }
    }
}
```

7.6.2.2. PubSub / Data Distribution

7.6.2.2.1. Publisher

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

public static void main(String[] args) {
    try (ZContext context = new ZContext()) {
        // PUB: Socket für Publisher erstellen
        ZMQ.Socket socket = context.createSocket(SocketType.PUB);

        // Publisher benutzen bind, um Nachrichten zu senden
        socket.bind(ADDRESS);

        while (...) { // Endlosschleife zur kontinuierlichen Veröffentlichung
            long station = (long) Math.floor(Math.random() * 3.0);
            long temp = 15 + (long) Math.floor(Math.random() * 10.0);

            // Topic implizit definiert durch Message-Prefix
            String message = "Temperature/" + station + "/" + temp;

            socket.send(message.getBytes(ZMQ.CHARSET));
            LOG.info("Published '" + message + "'");

            Thread.sleep(1000);
        }
    } catch (InterruptedException e) {
        LOG.info("interrupted"); // cannot occur
    }
}
```

7.6.2.2.2. Subscriber

```java
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); // SUB: Socket für
Subscriber
        socket.connect(ADDRESS); // Subscriber verbindet sich mit dem Publisher
        socket.subscribe(topic); // Bestimmtes Topic abonnieren (Message-Prefix)

        while (...) {
            byte[] message = socket.recv(); // Nur Messages der abonnierten Topics werden
empfangen
            LOG.info("Received: " + new String(message, ZMQ.CHARSET));
        }
    }
}
```

7.6.2.3. Pipeline / Taskverarbeitung

7.6.2.3.1. Producer

```java
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);

        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 (...) {
            String workPackage = "Work Package #" + ++i;
            socket.send(workPackage.getBytes(ZMQ.CHARSET)); // Blockiert bis Consumer
verbunden sind
            LOG.info("Send task '" + workPackage + "'");
            Thread.sleep(1000);
        }

    } catch (InterruptedException e) {
        LOG.info("interrupted");
    }
}
```

7.6.2.3.2. Consumer

```java
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);

        ZMQ.Socket socket = context.createSocket(SocketType.PULL); // PULL: Socket für
Consumer
        socket.connect(ADDRESS); // Connect für Fan-out, Bind für Fan-in

        while (...) {
            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");
    }
}
```

7.7. Protokoll: Web Sockets

Problem war das Pollen: Immer wenn man einen Auftrag an einen Webserver gesendet hat, musst man pollen (ist der Auftrag erledigt,...). -> mit dem WebSocket Protokoll braucht es das nicht mehr

- Synchrone Message-basierte Kommunikation.
- Erlaubt einfache Implementation des Full-Duplex Kommunikationsmusters.
- Protokoll auf Anwendungslevel.
- Basiert auf HTTP -> Protokollupgrade.

7.7.1. 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
```

7.7.2. Server

```
HTTP/1.1 101 Switching Protocols
Upgrade: websocket
Connection: Upgrade
Sec-WebSocket-Accept: HSmrc0sMlYUkAGmm5OPpG2HaGWk=
Sec-WebSocket-Protocol: chat
```

7.8. Persistente Kommunikation

7.8.1. 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

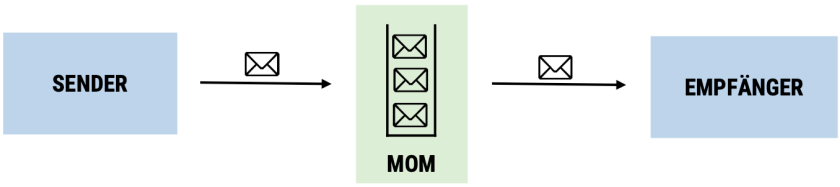


Abbildung 7: Persistente Kommunikation

Beispiele

- Apache ActiveMQ/ ActiveMQ Artemis
- RabbitMQ
- Websphere MQ

7.8.2. Message-Queuing-Model

- Kommunikation durch Einfügen von Message in Warteschlange von Empfänger.
- Keine Garantien, **wann** eine Message gelesen wird.
- Message von MOMs weitergeleitet, bis zum Ziel.
  - in der Regel sind die MOMs **direkt miteinander verbunden**
- in der Regel eine Queue pro Empfänger
  - Queues können aber geteilt werden

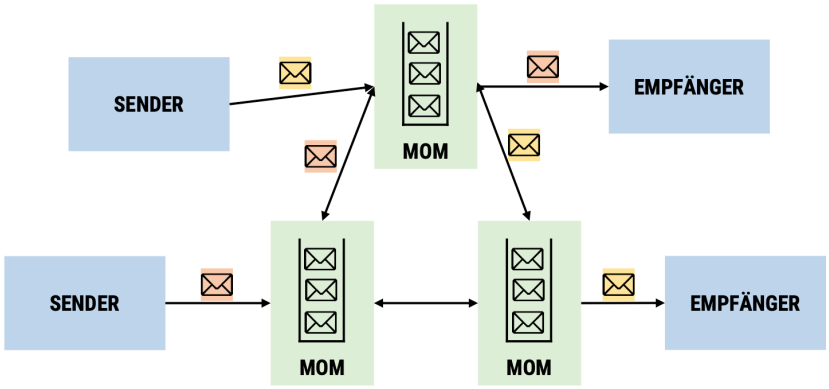


Abbildung 8: Message-Queuing-Model

7.8.2.1. Zeitliche Entkopplung mittels Message Queues

Vier Kombinationen:

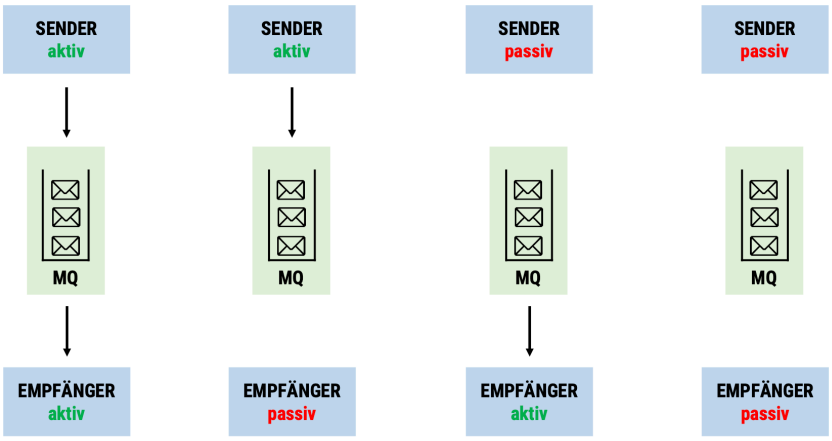


Abbildung 9: Zeitliche Entkopplung mittels Message Queues

7.8.2.2. Message Queue API

- `put()` : Nachricht zur Queue hinzufügen.
- `get()` : Warte bis eine Queue nicht mehr leer ist und retourniere die erste Message in der Queue. (**blockierend**).
- `poll()` : Prüfe auf Nachricht, falls vorhanden, erste Nachricht zurückgeben (**nicht blockierend**).
- `notify(callback)` : Callback registrieren, wird bei neuer Nachricht ausgeführt.

7.8.2.3. 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.

7.8.3. Message Broker

Integration von heterogenen Applikationen.

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

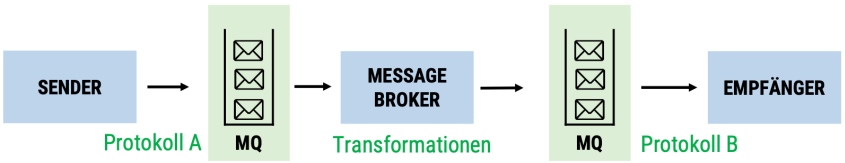


Abbildung 10: Message Broker

8. Dependency Management

- Organisation / Techniken für den Umgang mit **Abhängigkeiten zu anderen Modulen**.
- Abhängigkeiten typisch in **binär Form**
  - Binär-Repositories und Packagemanager (Für Java sind das **Maven Repository**)

Abhängigkeiten

- **intern:** Modul im selben Projekt
- **extern:** Dritt-Modul, aus anderem Projekt oder Organisation

8.1. Dependency Management für Java

Binäre Module (kompilierte Projekte) werden bei Java typisch als JAR-Dateien (oder EAR, WAR, RAR, JMOD, etc.) ausgetauscht.

Früher Dependency von Hand in `/lib` Verzeichnis kopieren.

- Fehleranfällig
- hohe Redundanz
- grosser Platzbedarf

Heute: Dependency Management von Apache Maven

Man unterscheidet zwischen:

- **Format** für die zentrale Ablage binärer Artefakte mit Metainformationen im Repository.
- **Werkzeug** zur Suche, Bezug, Deployment und Verwaltung von Artefakten im Repository.

8.2. Apache Maven

Ein Projekt wird mit folgenden drei Attribute identifiziert („maven coordinates“):

8.2.1. GroupId

- Meistens zusammengesetzt aus dem «reverse domain name»
- und einem Zusatz für eine OE, eine Projektgruppe etc. (Java Package Name)

Beispiel: `ch.hslu.vsk`

8.2.2. ArtifactId

- Name des Projektes

Beispiel: `stringpersistor-api`

8.2.3. Version

Empfohlen wir eine dreistellige Versionsnummer

Beispiel: `8.0.1`

8.3. Maven Coordinates

Mittels dieser Koordinaten wird / soll eine Dependency weltweit absolut eindeutig identifiziert werden können.

```xml
<groupId>ch.hslu.vsk</groupId>
<artifactId>stringpersistor-api</artifactId>
<version>8.0.3</version>
```

8.4. Deklaration im POM

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

```xml
<dependency>
  <groupId>ch.hslu.vsk</groupId>
  <artifactId>stringpersistor-api</artifactId>
  <version>8.0.3</version>
  <scope>compile</scope>
</dependency>
```

Diese werden beim Build automatisch vom Repository herunter geladen und im lokalen Repository `($HOME/.m2/repository)` gespeichert.

Repositorys manuell im Pom eintragen:

```xml
<repositories>
  <repository>
    <id>gitlab-maven-repo-vsk-stringpersistor-api</id>
    <url>https://gitlab.com/api/…/…/packages/maven</url>
  </repository>
</repositories>
```

8.5. Dependency Scopes

Damit wird der Zweck und Geltungsbereich (Scope) der Dependency qualifiziert. (Wird empfohlen)

8.5.1. compile

Kompilation und zur Laufzeit des Programmes benötigt (Default).

- Interface im Compile

8.5.2. test

Kompilation und Ausführung von Testfälle (Beispiele: JUnit, AssertJ etc.).

8.5.3. runtime

nur zur Laufzeit

- für dynamisch geladene Implementationen.

8.6. Transitive Dependencies

Abbildung 11: Transitive Dependencies

Auflösung der Dependencies:
- Modul **A** ist von Modul **B**, und dieses von Modul **C** abhängig.
- Somit ist Modul **A transitiv** von Modul **C** abhängig.
- Bei Kompilation von **A** wird Modul **C** auch miteinbezogen.

Durch **direkte / transitive Abhängigkeiten** können auch (Versions-)**Konflikte** oder **Zyklen** auftreten!
- Maven erkennt Konflikte
- Einfachere Versionskonflikte werden automatisch aufgelöst.

8.7. Semantic Versioning

8.7.1. Major (X.x.x)

Veränderung 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.

8.7.2. Minor (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.

8.7.3. Bugfix / Maintenance (x.x.X)

Reine Korrekturen oder Änderungen in der Implementation, voll rückwärtskompatibel, **keinerlei neue Funktionen**, **keine veränderten Funktionen**. - Direkter, sofortiger Einsatz möglich bzw. notwendig (Bugfix)

8.8. Snapshot

Mit Appendix –SNAPSHOT: Gilt diese als «erneuerbar» und (noch) nicht stabil, (in Entwicklung)
- Bei jedem Build immer wieder vom Repository aufgelöst und aktualisiert.
- Im Repository mit einem Timestamp versehen

8.9. Managed Dependencies in Multimodul-Projekten

Mehrere Submodule können die gleiche Dependency nutzen.

- Jedes Modul verwendet Log4J oder JUnit.

Im Parent-POM definiert `dependencyManagement` eine «Baseline» für Dependencies (Version, Scope).

- Submodule geben nur Group und ArtifactId an, der Rest wird vererbt.

Definition im (Parent-)POM:

```xml
<dependencyManagement>
  <dependencies>
    <dependency>
      <groupId>org.slf4j</groupId>
      <artifactId>slf4j-api</artifactId>
      <version>2.1.0</version>
      <scope>compile</scope>
    </dependency>
  </dependencies>
  ...
</dependencyManagement>
```

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

```xml
<dependencies>
  <dependency>
  <groupId>org.slf4j</groupId>
  <artifactId>slf4j-api</artifactId>
  // Version und Scope entfällt, weil von Parent geerbt!
  </dependency>
</dependencies>
```

8.9.1. BOM

Eine Alternative ist die BOM («Bill of Material»).

- Referenziert verschiedene Versionen als «Baseline».
- Lieferant legt kompatible Versionen fest.
- Wird selbst als versionierte Abhängigkeit definiert.

8.10. Deployment in Maven Repositories

Das Deployment in öffentliche Repositories (z. B. Maven Central) ist restriktiv:

- Keine nachträglichen Änderungen oder Löschungen zur Sicherung der Build-Stabilität.
- Entwickler deployen nicht direkt, sondern über einen automatisierten, nachvollziehbaren und verifizierbaren Release-Prozess vom Buildserver.

9. Buildserver

Serversoftware, die automatisierte Builds ausführt und Ergebnisse dem Team bereitstellt.

Auslösung eines Builds aufgrund verschiedener Events:

- Automatisch durch Änderungen im Versionskontrollsystem.
- Automatisch durch Zeitsteuerung
- Manuell durch Anwender in.

Positive Effekte:

- Entlastung von Entwickler repetitiven Aufgaben
- Häufigere Verifikation (Buildprozess, Tests, Deploying etc.).
- Statistische Information über Entwicklungsprozess.
- Offensive (automatische) Information über den Zustand der Projekte.

9.1. Konfiguration

Variante 1 (typisch für «on-site» Produkte/Projekte)

- Konfiguration ist vom Projekt vollständig getrennt.
- Wird interaktiv direkt auf dem Buildserver vorgenommen.
- Infrastruktur (Buildagents etc.) wird «geschützt».
- -> Es resultiert eine Art «Gewaltentrennung».

Variante 2 (eher für Cloud und Hosting-Plattform und OSS)

- Konfiguration ist direkt im Projekt abgelegt (z.B. .yml-Datei).
- Wird direkt durch Entwicklerinnen konfiguriert.
- Weniger restriktiv, sehr häufig mit Docker (ad-hoc Agents).
- -> Mehr Freiheit und Eigenverantwortung der Entwicklerinnen.

9.2. Einsatz

Einsatz eine Buildservers setzt andere Technologien voraus:

- **Automatisierte Builds**, z.B. mit Maven oder Gradle etc.
- Einsatz eines **Versionskontrollsystems**, z.B. Git, Subversion etc.

Auf Aufgabentrennung 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.

9.3. Buildarten

9.3.1. Continous

Automatisch bei einer Änderung (Push/Merge-Request/Pull-Request) im Versionskontrollsystem.

- Schnelle, möglichst kurze Builds.
- Ziel: Schnelles Feedback für Entwickler

9.3.2. Nightly

Automatisch nach Zeitsteuerung, typisch Nachts.

- Eher umfangreiche, lange Builds.
- Auch für zeitintensive Tests und Metriken geeignet.
- Ziel: Am Morgen stehen umfassende Resultate zur Verfügung.

9.3.3. Release

Manuell ausgelöst.

- Build einer auslieferbaren Version, im VCS getagged.
- Ziel: Reproduzierbarkeit gewährleisten.

- Alternative: Build Pipeline.

9.4. Integration

Integration von
- verschiedenen Buildtools.
- verschiedenen Versionskontrollsystemen.
- verschiedenen Kommunikationstechnologien zur Notifikation.
- verschiedene Visualisierungen / Plugins für IDE‘s.
- etc.

Verknüpfung mit
- Issue-Tracking Systemene
- Code-Review Werkzeugen
- etc.

10. Architekturbeschreibung

☝ Was man im Code sieht, soll NICHT dokumentiert werden!

10.1. Ziel der Architekturbeschreibung

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

Anforderungen (ggf. iterativ) abgeleitet aus:
- übergeordneten Anforderungsdokumenten (z.B. Lastenheft).
- übergeordneten Systemen (z.B. Schnittstellen der umgebenden Systeme).

Alternative Namen:
- Systemspezifikation
- Systembeschreibung

10.2. Nutzen der Architekturbeschreibung

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

10.3. Kompatibilität mit agilen Vorgehensmodellen

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

10.4. Vorlagen

Auswahl von verschiedenen Vorlagen:
- arc42
- SA4D (Software Architecture for Developers)
- Firmeneigenen Vorlagen (spezifischer auf ein Projekt, besser geeignet bei grossen Projekten)

Warum Vorlagen verwenden?
- Dokumentationen sind gleich aufgebaut
- einfacher etwas zu finden
- ist auch quasi eine Checkliste

10.4.1. arc42

- Gängiger Standard im deutschsprachigen Raum
- Beantwortet zentrale Fragen:
  - **WAS** sollen wir bei unsere Architektur kommunizieren / dokumentieren?
  - **WIE** sollen wir kommunizieren / dokumentieren?

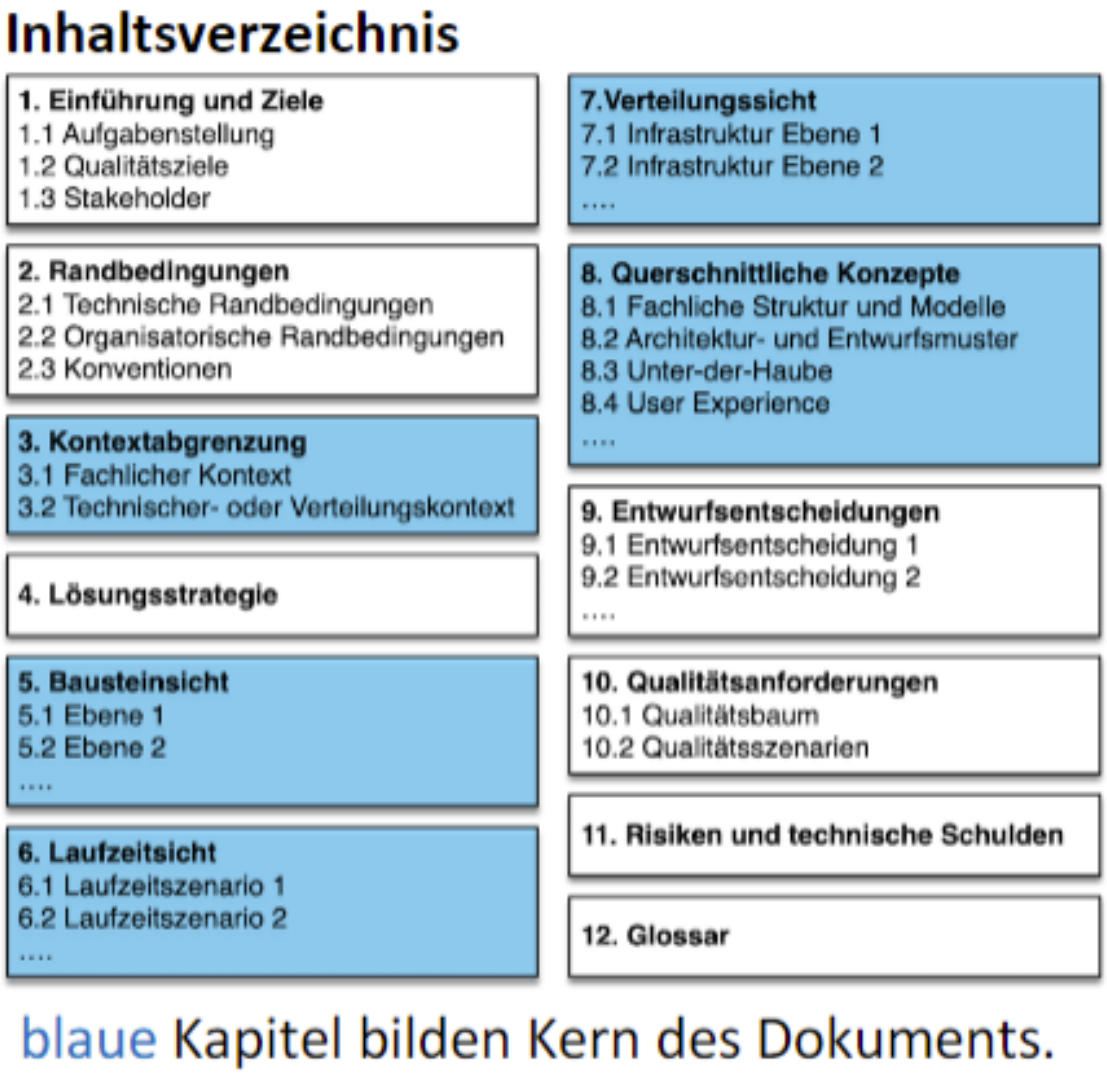


Abbildung 12: arc42

10.4.2. SA4D

ZIEL : Beschreibe was nicht offensichtlich im Code erkennbar ist

Hinweis: Wird kombiniert mit C4-Architektur-Visualisierungsmodell.

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

Abbildung 13: SA4D

10.4.3. Firmeneigene Dokumente

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

10.5. Inhalt der Architekturbeschreibung

Notwendige Inhalte der Architekturbeschreibung:

- Einführung mit Systemübersicht und Kontextabgrenzung
- Schnittstellen nach Aussen
- Architektonische Sichten (Softwarearchitektur, Bausteinsicht, Laufzeitsicht)
- Verteilungssicht
- Querschnittsthemen

10.5.1. Einführung und Systemübersicht

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

10.5.2. Kontextabgrenzung

ZIEL : Das zu entwickelnde System in seinem Kontext zeigen (fachlich / technisch)

WICHTIG: jedes Diagramm muss mit Text beschrieben werden!

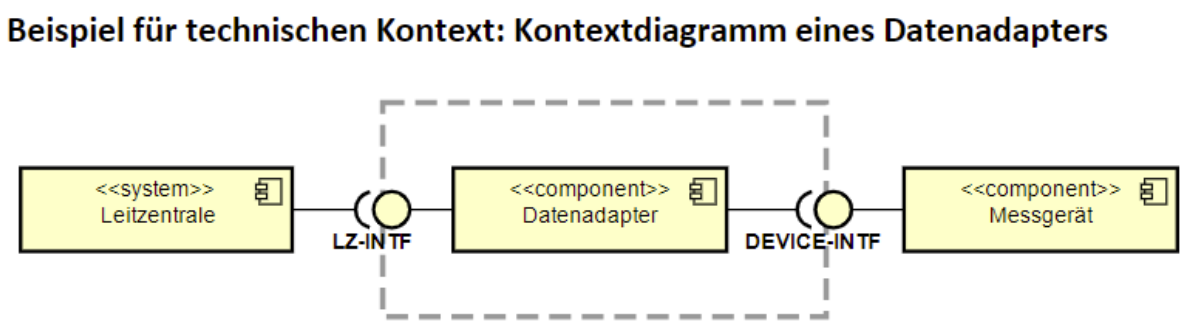


Abbildung 14: Kontextabgrenzung

10.5.2.1. Schnittstellen

WAS:

- Externe Schnittstellen
  - exportierte und auch importierte Schnittstellen
- Benutzerschnittstellen

WIE:

- Schnittstellen in separatem Dokument beschreiben
- in diesem Kapital exakt referenzieren

10.5.3. Softwarearchitektur

- Darstellung von unterschiedliche Sichten 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).

10.5.4. Bausteinsicht und Laufzeitsicht (Logische Sichten)

- Funktionalität des Systems
  - auf unterschiedlichen Flughöhen
- C4-Modell am besten geeignet

Pro System mehrere Ebenen (falls zutreffend):

- Teilsystem / Services: z.B. Block- oder Komponentendiagramme.
- Komponenten: z.B. Komponentendiagramme.
- Code: i.d.R. Klassendiagram oder Sequenzdiagramme.

Richtlinien

- Kein Reverse-Engineering von Diagrammen aus dem Code!
- Nur interessante / hilfreiche Stellen dokumentieren.
- Auf das Notwendigste reduzieren: nur relevante Klassen und Attribute.

10.5.4.1. C4 Modell

Level 1 bis 3 sind Kontextdiagramme, Level 4 sind Klassendiagramme

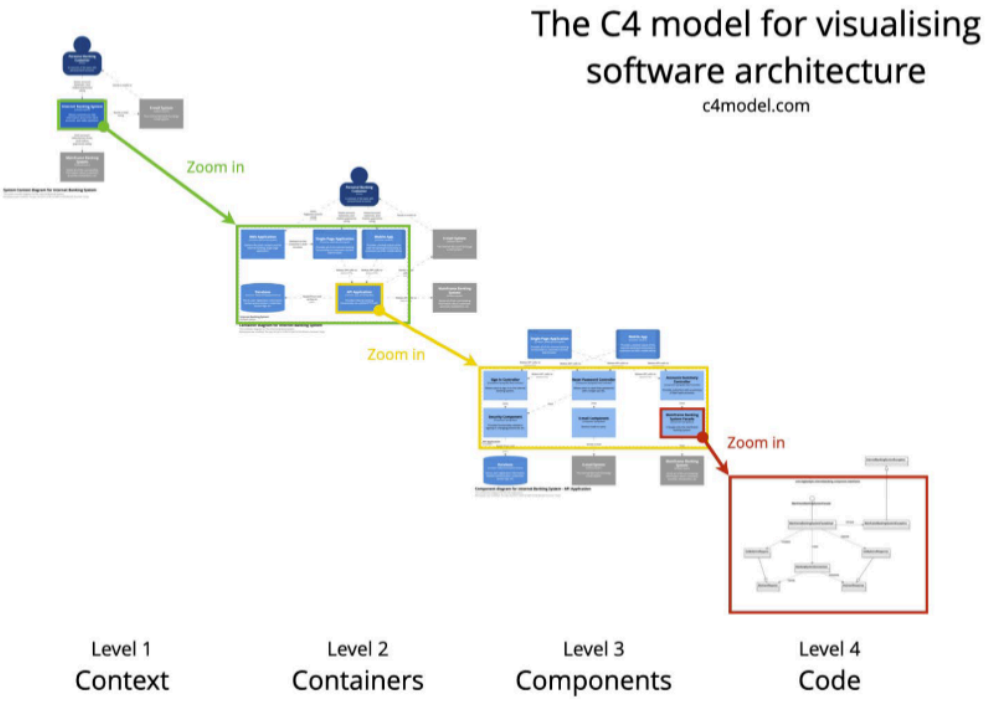
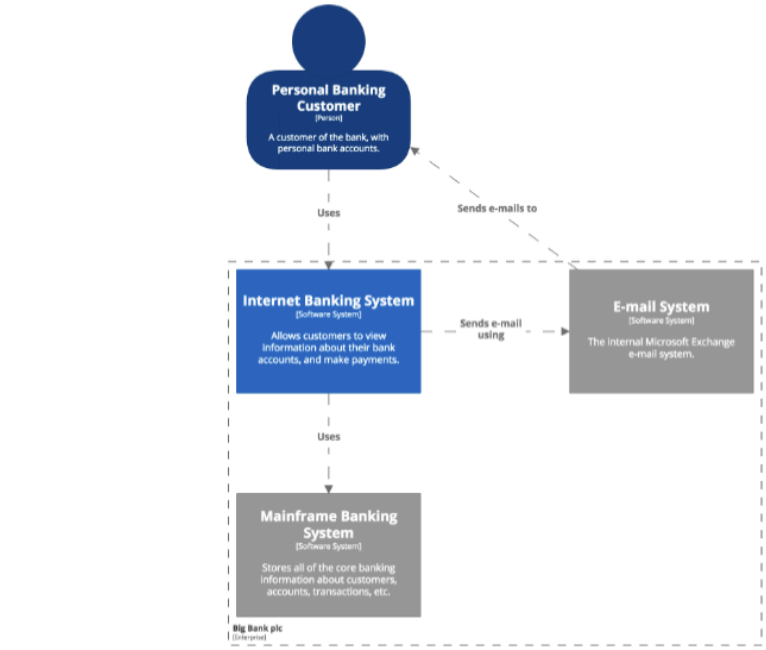


Abbildung 15: C4 model

Bausteinsicht am Beispiel des C4-Modells (Ebene 1)

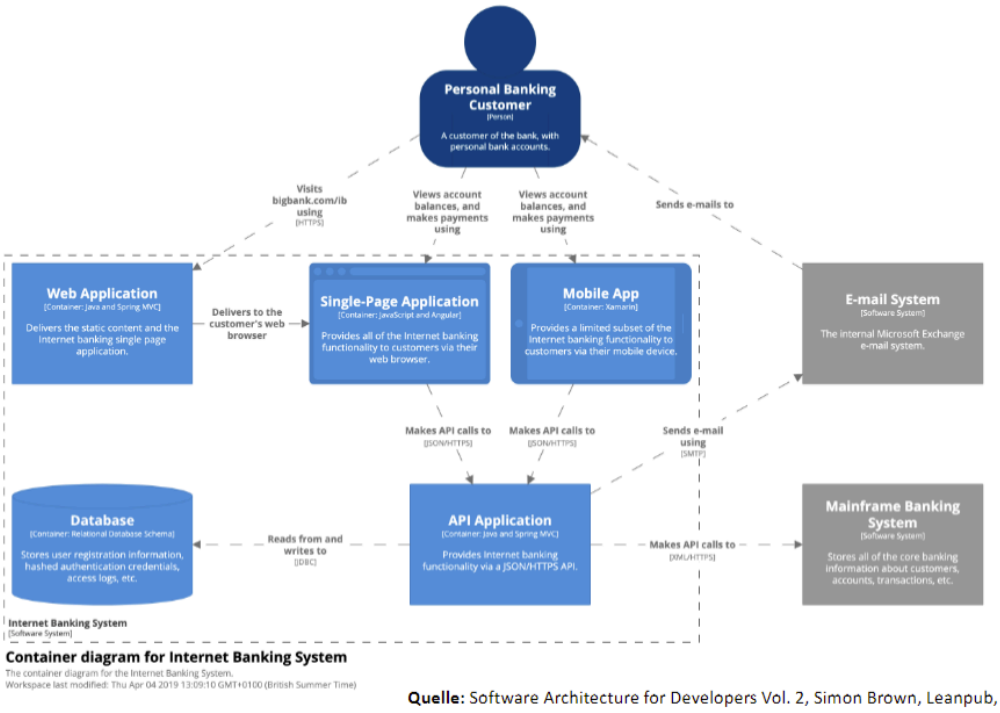


System Context diagram for Internet Banking System
The system context diagram for the Internet Banking System.
Last modified: Wednesday 02 May 2018 13:41 BST

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

Abbildung 16: C4 model Ebene 1

Bausteinsicht am Beispiel des C4-Modells (Ebene 2)

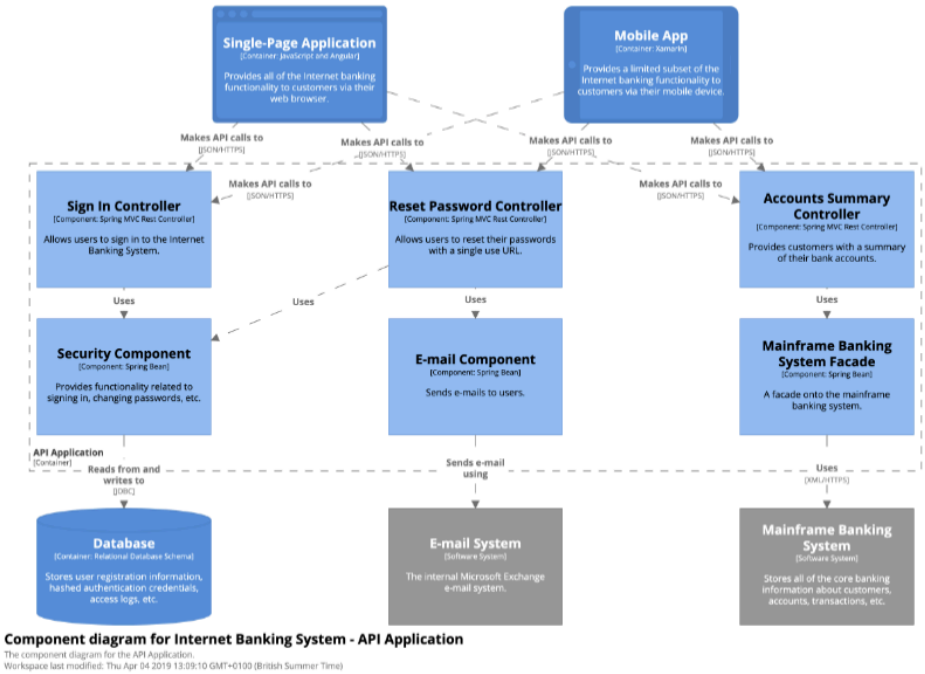


Quelle: Software Architecture for Developers Vol. 2, Simon Brown, Leanpub, 2021.

Abbildung 17: C4 model Ebene 2

-> auf Level 3 gibt es in diesem Beispiel 4 Diagramm (man zoomt in jedes von Ebene 2 hinein)

Bausteinsicht am Beispiel des C4-Modells (Ebene 3)



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

Abbildung 18: C4 model Ebene 3

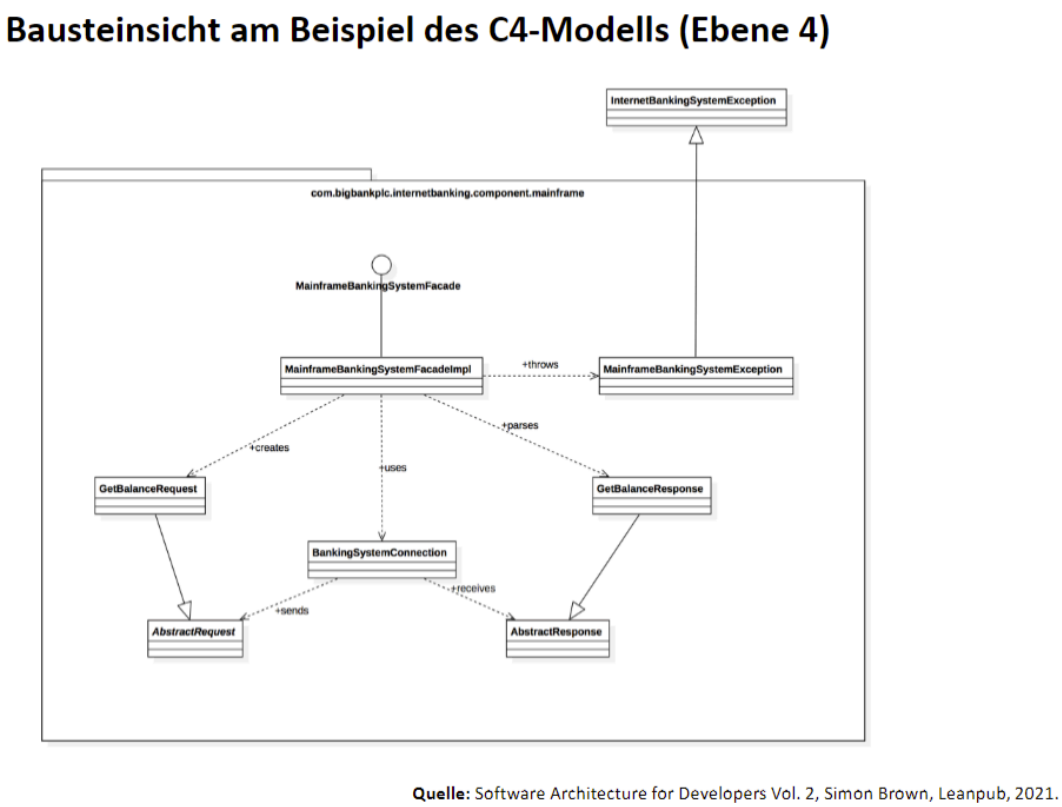


Abbildung 19: C4 model Ebene 4

10.5.5. Verteilungsschicht

-> Wie kann man das System auf Hardware (oder virtuelle Umgebungen) verteilen

- 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.

10.5.6. Querschnittsthemen

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.

Teststrategie

- Wie wird das System und seine Komponenten getestet?

Wichtige Schnittstellen:

- Interne Schnittstellen zwischen Komponenten

10.5.7. 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.

10.5.8. Qualitätsanforderungen

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

11. Komponente - Modularisierung

11.1. Modularisierung: Konzept und Vorgehen

11.1.1. Modul: Begriff

❗ **In sich abgeschlossener Teil** des gesamten Programm-Codes, bestehend aus einer Folge von Verarbeitungsschritten und Datenstrukturen.

Beispiele von Modulen:
- Klasse
- Komponente
- Teilsystem / Services
- Package (Sammlung von Klassen)
  - package in Java ist ein namespace und in diesem Sinne kein Modul
- Funktion (in Java nicht, aber in anderen Sprachen kann es durchaus ein Modul sein)

Big Ball of Mud *-> kommt heraus, wenn man sich nicht um Modularisierung kümmert (grosser Wollknäuel)*

11.1.2. Koppelung und Kohäsion

11.1.2.1. Konzept

Koppelung (Coupling):
- Ausmass der Kommunikation zwischen Modulen
- Unabhängigkeit der einzelnen Modulen

❗ Koppelung minimieren

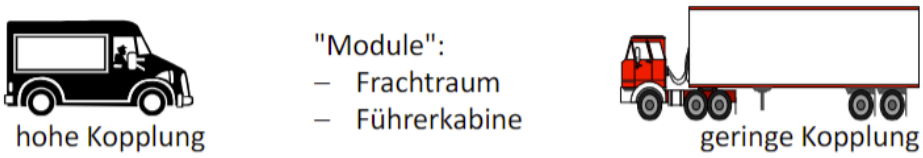


Abbildung 20: Koppelung

Kohäsion (Cohesion):
- Ausmass der Kommunikation innerhalb eines Moduls.
- interner Zusammenhalt innerhalb eines Moduls

❗ Kohäsion maximieren

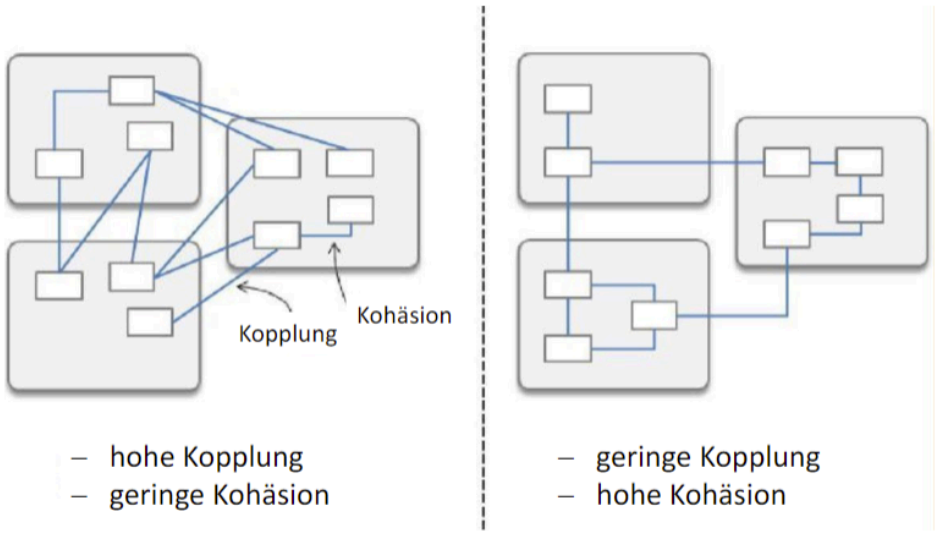


Abbildung 21: Koppelung und Kohäsion

11.1.2.2. Umgang mit Koppelung und Kohäsion

Kohäsion maximieren und gleichzeitig Kopplung minimieren ist nicht immer möglich. -> Optimierungsproblem / -aufgabe

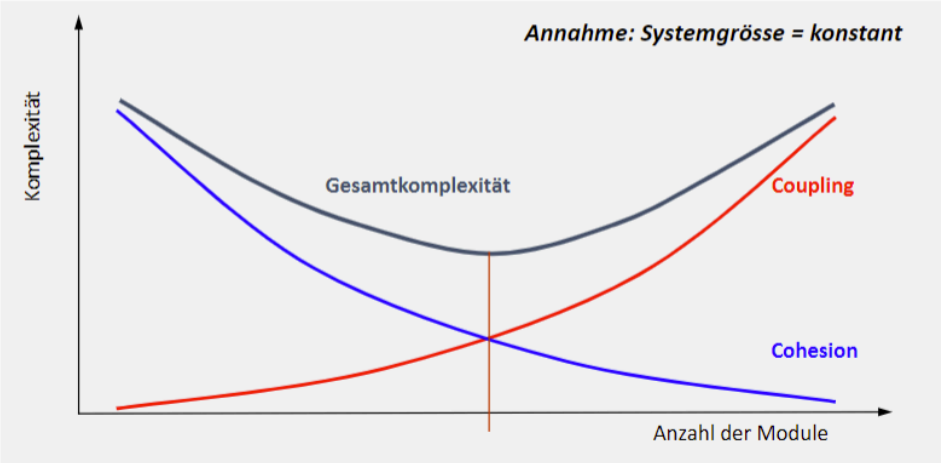


Abbildung 22: Umgang mit Koppelung und Kohäsion

Ziel: Gesamtkomplexität soll so klein wie möglich sein.

11.1.2.3. Arten der Kohäsion

Funktionale Kohäsion: Alle Teile haben eine ineinandergreifende Beziehung zueinander. (Methoden rufen sich auf, nutzen die gleiche Properties usw.) -> kann man automatisiert auswerten

Logische Kohäsion: Logische, aber nicht funktionale Beziehung (z.B. mathematische Library) -> schwieriger auszuwerten

Zufällige Kohäsion: Einheit sind zufällig im selben Modul

Schlechtheitsskala (das Schlechteste zuerst):
- Zufällige Kohäsion
- Logische Kohäsion
- Funktionale Kohäsion

11.1.2.4. Messung der Kohäsion (funktionale Kohäsion)

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

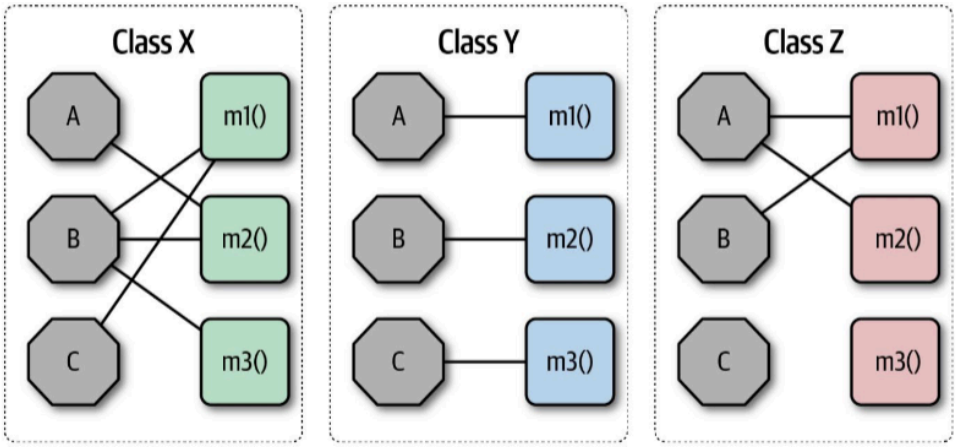


Abbildung 23: Messung der Kohäsion

A, B, C sind Zustände

Fazit Class X: sehr gut Fazit Class Y: könnte man in verschiedene Klassen auslagern Fazit Class Z: mix aus X und Y, m3() in separate Klasse

11.1.2.5. Arten der Koppelung

-> Entkoppelung kostet immer Zeit

❗ Damit Module zusammen kommunizieren können, müssen sie etwas gleich haben. Am besten möglichst tiefe Koppelung (z.B. Daten und Formate).

Laufzeitumgebung, Ausführungsort: Module müssen in derselben Laufzeitumgebung oder auf demselben System ausgeführt werden.

Technologie: Gekoppelte Module müssen (teilweise) dieselben Technologien verwenden.

Zeit: Module müssen zur selben Zeit aktiv sein.

Daten und Formate: Module müssen dieselben Datenformate parsen und verstehen (z.B. Datum oder Headers).

Schlechtigkeitsskala (das schlechteste zuoberst):
- Laufzeitumgebungen, Ausführungsort
- Technologie
- Zeit
- Daten und Formate

11.1.3. 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.

Kombinierbarkeit und Zerlegbarkeit sind voneinander unabhängige Eigenschaften.

11.1.3.1. Zerlegbarkeit (Top-Down)

Ziel: Softwareproblem in weniger komplexe Teilprobleme zerlegen und diese so verknüpfen, 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.

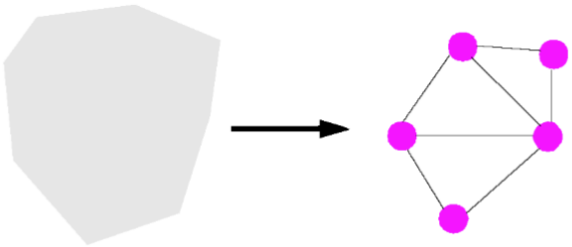


Abbildung 24: Zerlegbarkeit

11.1.3.2. Kombinierbarkeit (Bottom-Up)

Ziel: frei kombinierbare Software-Elemente, die sich auch in einem anderen Umfeld wieder einsetzen lassen

-> prüfen ob ein Element zu spezifisch für das Problem ist, und ob man das verbessern kann

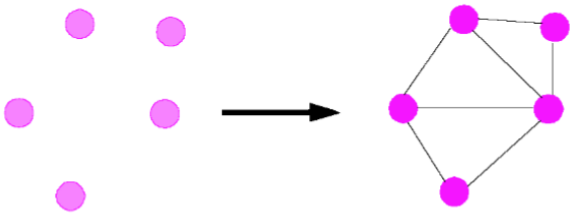


Abbildung 25: Kombinierbarkeit

11.1.3.3. Verständlichkeit

Ziel: Der Quellcode eines Moduls soll auch verständlich 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.

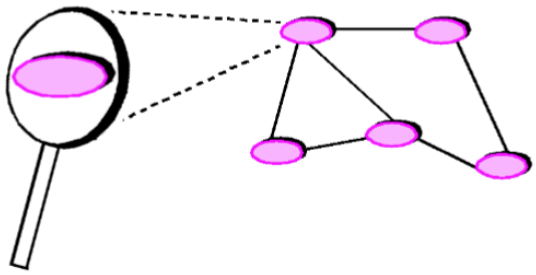


Abbildung 26: Verständlichkeit

11.1.3.4. Stetigkeit

Ziel: 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.

Messung, ob Stetigkeit erfüllt ist: Elementen mit zu vielen Pfeilen = nicht erfüllt

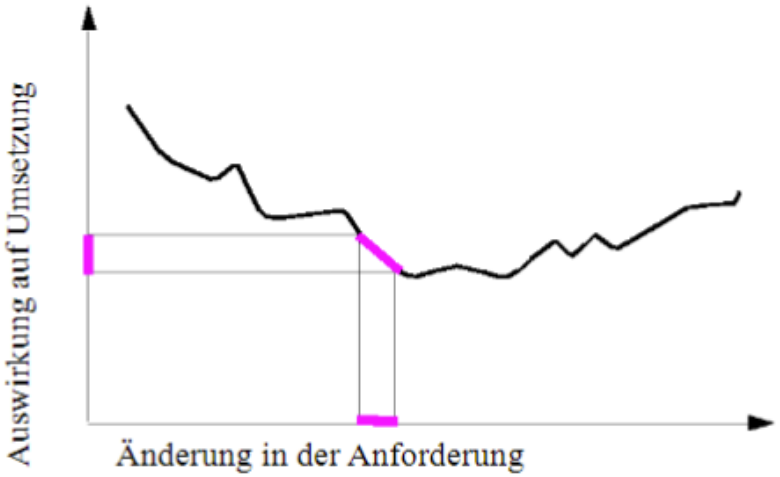


Abbildung 27: Stetigkeit

11.1.4. Prinzipien des modularen Entwurfs

Lose Kopplung: Schmale Schnittstellen, um nur das wirklich Benötigte auszutauschen.

Starke Kohäsion: Hoher Zusammenhalt innerhalb eines Moduls.

Information Hiding (public / private): 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.

11.1.5. Modularisierung: Iteratives Vorgehen

1. Zerlegung (Top-Down oder Buttom-Up) unter Anwendung der Prinzipien
   - wenig Koppelung und viel Kohäsion
   - Information-Hiding
   - wenige und explizite Schnittstellen
2. Beurteilung von:
   - Zerlegbarkeit: Module aufgeteilt und unabhängig bearbeitbar?
   - Kombinierbarkeit: Können Module wiederverwendet werden?
   - Verständlichkeit: Viel Kohärenz und wenig Kopplung?
   - Stetigkeit: Auswirkung von Änderungen?
3. Falls Kriterien nicht zufriedenstellend: Zurück zu 1

11.2. Schichtenarchitektur

11.2.1. Was ist Softwarearchitektur?

Die Softwarearchitektur befasst sich mit den folgenden vier Bereichen:

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

Struktur Verwendete Architekturstile

Architektonische Entscheidungen Regeln, welche EntwicklerInnen beim Entwurf der Komponenten befolgen müssen.

Entwurfsprinzipien Richtlinien, welche EntwicklerInnen bei Entwurf der Komponenten des System anwenden sollen.

11.2.2. Was sind Schichten?

- Modulhierarchie

- öffentliche Methoden der Schicht B (der unterliegenden) dürfen von der Schicht A (oberhalb) genutzt werden
  - umgekehrt NICHT
- use-Beziehung, wenn das korrekte Funktionieren von A von einer korrekten Implementation von B abhängt

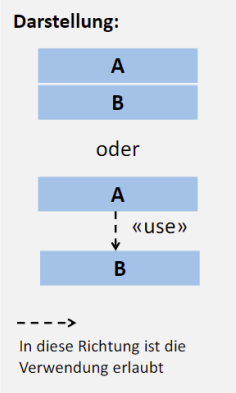


Abbildung 28: SA4D

11.2.3. Schichtenarchitektur

- Technische Modularisierung oft entlang organisatorischen Einheiten

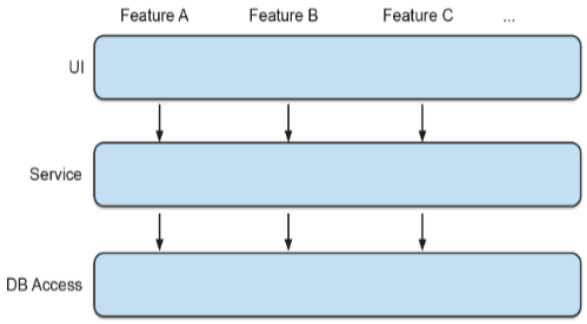


Abbildung 29: SA4D

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

11.2.3.1. 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).

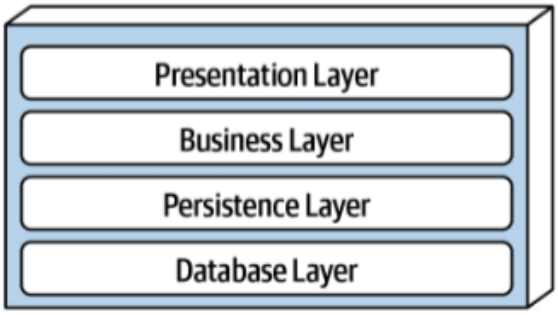
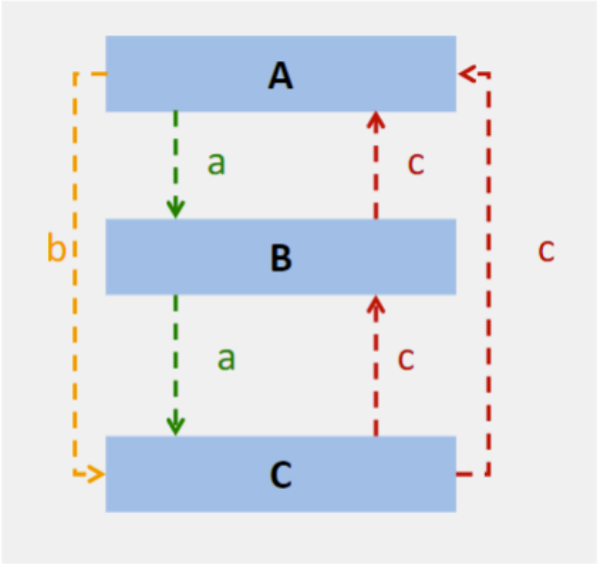


Abbildung 30: Typische Schichten

11.2.4. Schichtenbeziehungen: Zulässigkeit



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

Abbildung 31: Zulässigkeit

Grundarchitektur: man will nur, dass Schichten unterhalb aufgerufen werden können und nicht umgekehrt

11.3. Weitere Schichtenbeziehung

- Eine Klasse P1 kann P2 aufrufen, ohne eine use-Beziehung mit P2 zu haben

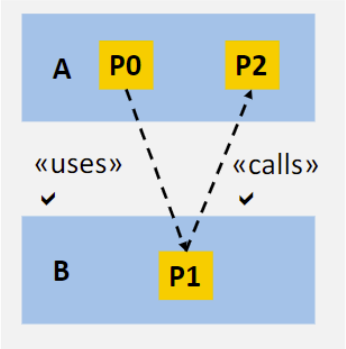


Abbildung 32: Weitere Schichtenbeziehungen

Beispiel:

- P2 sei ein Errorhandler, dessen Referenz von P0 an P1 übergeben wurde. Die Referenz des Errorhandlers ist nicht in P1 festkodiert
- observer pattern

11.3.1. 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

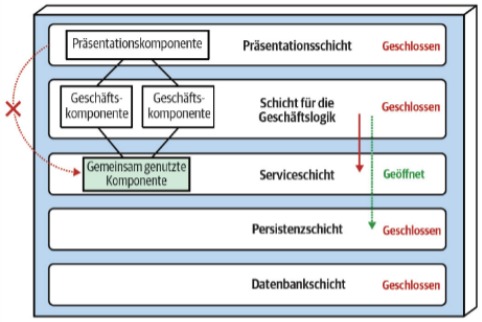


Abbildung 33: offene Schichtarchitektur

11.3.2. Schichten vs. Tier

Schichten: logische Separierung

Tier: Zusätzlich ein physische Separierung oft kombiniert mit Verteilung (heutzutage oft Service)

Beispiel: vier Schichten auf drei Tiers

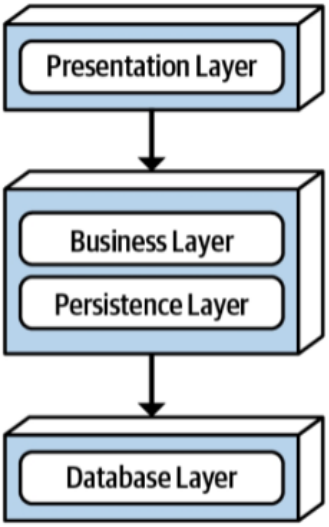


Abbildung 34: Schichten vs. Tier

11.3.3. Bewertung der schichtbasierten Architektur

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

12. Container 🐳

12.1. Was ist Docker?

- Leichtgewichtige virtuelle Maschinen
- Isolation von Prozessen und Applikationen ohne Notwendigkeit eines eigenen virtualisierten Kernels
- Bündelung von Applikationen und Abhängigkeiten in einem Docker-Container
- Nutzdaten von der Laufzeitumgebung trennen
- Entstand in der Linux-Welt, weshalb Unix-Kenntnisse von Vorteil sind
- `docker` ist eine CLI-Anwendung zum Betreiben von Containern
- Alles wird durch eine eindeutige, auf Hash-basierende ID gekennzeichnet
  - Oft auch in gekürzter Form angezeigt
  - Es reichen so viele Stellen, dass die ID eindeutig von den anderen unterscheidbar ist (in der Regel 2 - 3 Zeichen)

12.1.1. Funktionsweise

- Isolation durch *cgroups* und *namespaces* zur Trennung von Ressourcen
- Prozesse werden isoliert auf demselben Kernel betrieben
- Zugriff auf virtuelles Dateisystem und isoliertes Memory
- Basis-Funktionen in der Schnittstelle `libcontainer` zusammengefasst
- Betrieb in VM oder WSL möglich

12.1.2. Begriffe

Container

- Laufende Instanz einer Anwendung
- Basiert auf einem *Image*

Images

- Enthält ausführbare Anwendung in Form eines *virtuellen*, *geschichteten* Filesystems
- Ist *Read-Only*
- Viele Projekte bieten fertig Docker Images an
  - Auf Quelle achten („Docker Official Image“)

Tags

- Versionslabels eines Images
- Relabeling möglich -> Label wird an anderes Image umgehängt
- *nicht* analog zu einem *git tag*

Volumes

- Virtuelles Filesystem
- Persistierungsmöglichkeit für Container-Daten
- Wird in der Regel in den Container „gemountet“

Registry

- Service, welcher mehrere *Repositorys* zur Verfügung stellt
- Teil in DevOps-Plattformen wie z. B. in GitLab integriert
- Auch als alleinstehende Services möglich
- Vergleichbar mit einem Git-Server bei der Softwareversionierung
- SaaS Beispiele: Docker Hub, GitHub Container Registry, …

Repository

- Gruppe von Images unterschiedlicher Tags
- In einer Registry angesiedelt
- Vergleichbar mit einem Git *Repository*

Netzwerk

- Es gibt unterschiedliche Netzwerkmodelle, um zu bestimmen, mit wem Container Daten austauschen können

12.1.2.1. Beispiel

```sh
docker run -d \  # Container erstellen und im Hintergrund starten
  --name my_container \  # Container-Name setzen
  -v my_volume:/data \  # Volume für persistente Daten einbinden
  --network my_network \  # Container mit einem bestimmten Netzwerk verbinden
  my_registry/my_repository:latest  # Image aus einer Registry mit dem Tag latest
```

12.1.3. Einsatzszenarien

- Ausprobieren von neuen Anwendungen (Einfaches Aufsetzen und wieder abräumen von komplexen systemen)
- Buildumgebung für unterschiedliche Technologien
- JDK & Build-Tools: Verschiedene Versionen ohne lokale Installation nutzbar
- Deployment und Betrieb von produktiven Applikationen
- Schnelle und flexible Testumgebungen (Testcontainer)

12.1.4. Port Exposure

- Notwendig, um Container-Applikationen über Port nach aussen zugänglich zu machen
- Mapping von Container-Ports auf Host-System-Ports
- Über CLI-Flag `-p` / `--publish` möglich
- Zuerst Angabe des Ports auf dem Hostsystem, dann Port innerhalb des Containers, also z. B. `-p 80:8080`

12.2. Docker Images

- Werden in *Layers* erstellt
- Aufeinanderschichtung von unterschiedlichen Änderungen
- Schritte, ein Image aufzubauen, werden in einem Dockerfile beschrieben
- Images werden dadurch reproduzierbar
- Dockerfiles werden oft unter Versionskontrolle genommen

12.2.1. Dockerfile

- Definition von Aktionen, welche ausgeführt werden, um ein Image zu erstellen

Beispiel eines einfachen `Dockerfile`s:

```dockerfile
FROM amazoncorretto:21.0.6-alpine
COPY target/oop_maven_template.jar /opt/demo/oop_maven_template.jar
CMD ["java","-jar","/opt/demo/oop_maven_template.jar"]
```

`FROM` Basislayer auf welchen weiter aufgeschichtet wird / auf welchem das Image beruht
`COPY` Gebaute `.jar`-Datei wird zur Ausführung in den Container kopiert (Shade-JAR inkl. Manifest)
`CMD` Hinterlegen des Startbefehls, welcher beim Starten eines Containers mit diesem Image ausgeführt wird

12.2.2. Veröffentlichen eines Images

1. Lokales Bauen eines Images und Ablegen im lokalen Registry (`docker build -t "beispiel/demo:latest"`)
2. Veröffentlichen eines Images in einer Registry (`docker push "beispiel/demo:latest"`)
3. Images sind dann öffentlich oder privat in Docker Hub verfügbar

12.2.3. Herausforderungen

- Konfiguration nach Laufzeitumgebung -> Evtl. andere Architektur
- Aus bestehenden Dockerfiles kann vieles gelernt werden, jedoch auch unschöne Sachen kopiert werden
- Sinnvolles bilden von Layers nicht immer einfach
  - So wenig wie möglich, aber so viel wie nötig
  - Möglichst ähnliche Änderungshäufigkeit pro Layer
  - Möglichst kompakte Images

- Layers, die häufiger geändert werden, weiter unten im Dockerfile ansiedeln
- Statische Layers weiter oben platzieren, damit sie nicht immer neu erstellt werden müssen“

12.3. Docker + Java

- Java-Applikationen brauchen eine Laufzeitumgebung (JRE), welche Platz braucht
- Unterschiedliche Deploymentmöglichkeiten
  - Single JARs, alles in einem File
  - App-JAR und Dependency-JAR, classpath nötig, typischerweise durch ein Shell-Script gestartet
  - Alles im selben oder in unterschiedlichen Layers?
  - Applikation modularisieren? (Java Platform Modul System)
- Es gibt Maven-Plugins, welche das Erstellen von Containern vereinfachen
  - Fabric8 Maven Docker Plugin
    - Basiert auf Dockerfile
    - Konfiguration teils über `pom.xml`

```sh
mvn docker:build   # Docker-Image bauen
mvn docker:start   # Container starten
mvn docker:stop    # Container stoppen
```

  - Google Jib Plugin
    - Hohe Automatisierung
    - Konfiguration *nur* über `pom.xml`
    - Build und Deploy auch ohne Docker-Runtime möglich

```sh
mvn jib:build         # Image bauen & in Registry pushen
mvn jib:dockerBuild   # Image lokal bauen
```

12.4. Testcontainer

- Schnelles Hoch- und Runterfahren für (Integrations-)Tests
- Testcontainers als Lösung: https://www.testcontainers.org
- Automatisches Starten und Stoppen von Containern
- Dynamische Port-Vergabe
- Automatisches Aufräumen nach Tests

12.5. Wichtige Docker Befehle

12.5.1. Starten eines Containers

Die meisten der nachfolgenden Flags sind auch kombinierbar

`docker run <image>:<tag>`
- Container wird auf Basis von `<image>:<tag>` gestartet
- Container ist im Vordergrund und alle Outputs werden direkt angezeigt
- Durch `CTRL+C` wird der Prozess unterbrochen und der Container beendet sich

`docker run --name <name> <image>:<tag>`
- Versieht den Container mit einem unter `<name>` angegebenen Namen
- Eine Art Alias, welcher synonym für dessen ID verwendet werden kann

`docker run -d <image>:<tag>`
- Container läuft im `detached` Mode und somit im Hintergrund
- Stoppen durch `docker stop [<id>|<name>]` möglich

`docker run --rm <image>:<tag>`
- Container wird nach dem Beenden automatisch entfernen

- Wird dieses Flag nicht mitgegeben, so bleibt der Container im `Exited`-Status liegen und kann mit `docker start [<id>|>name>]` wieder gestartet werden

`docker run -p <host-port>:<container-port> <image>:<tag>`

- Verbindet einen Port auf dem Hostsystem mit einem Port innerhalb des Containers
- Mit `-p 80:8080` wird der Port 80 auf dem Hostsystem mit Port `8080` innerhalb des Containers verbunden

`docker run -v [<host-path>|<volume-name>]:<container-path>:<flags> <image>:<tag>`

- Ermöglicht das Anhängen eines Volumens zur Persistenz von Daten
- Wir können entweder einen Pfad auf dem Hostsystem mounten oder ein *named volume* verwenden
  - Host-Pfad `/tmp/some` innerhalb des Containers als `/tmp` mounten: `-v /tmp/some:/tmp`
  - Volumen namens `test` innerhalb des Containers als `/tmp` mounten: `-v test:/tmp`

—

`docker exec -it [<name>|<id>] <command>`

- Startet den unter `<command>` angegebenen Befehl interaktiv im Container
- Durch `docker exec -it test /bin/bash` wird z. B. eine `bash`-Shell in einem Container namend `test` gestartet

`docker ps` Anzeigen der aktuell laufenden Container

`docker ps -a` Anzeigen aller vorhanden Container

`docker cp <from> <to>`

- Kopieren von Inhalten in oder aus einem Container
- Ein Pfad innerhalb eines Containers wird durch `[<name>|<id>]:<path>` angegeben
- Wir kopieren die Datei `/tmp/test.txt` in den Container `test` nach `/tmp/test.txt`: `docker cp /tmp/test.txt test:/tmp/test.txt`

`docker logs <name>`

- Anzeigen der Logs eines Containers

`docker logs -f <name>`

- Anzeigen der Logs inkl. following
- Wenn neue Logs generiert werden, erscheinen diese ebenfalls
- Viewer bleibt attached

`docker inspect [<name>|<id>]`

- Anzeigen von Container-Informationen

`docker images` Anzeigen der in der lokalen Registry vorhandenen Images

`docker pull <name>:<tag>` Herunterladen eines Images aus einer Registry

`docker start [<name>|<id>]` Starten eines Containers

`docker stop [<name>|<id>]` Stoppen eines Containers

`docker restart  <name>>` Neustart eines Containers

`docker rm [<name>|<id>]` Entfernen eines Containers

`docker rmi [<name>|<id>]` Entfernen eines Images

`docker volume ls` Listet Docker-Volumes auf

`docker volume rm <volume>` Löscht ein Docker-Volume

`docker network ls` Listet Docker-Netzwerke auf

`docker network create <name>` Erstellt ein neues Netzwerk

`docker system prune -af --volumes` Entfernen *aller* Container, Netzwerke, Volumen, Images und cache-daten welche nicht gerade mit einem laufenden Container in Verbindung stehen

`docker container prune` Löscht gestoppte Container

`docker image prune` Löscht ungenutzte Images

`docker volume prune` Löscht ungenutzte Volumes

13. Testing - Integration- und Systemtests

Funktionalitäten und Anwendungsfälle sollen systematisch getestet werden. Eigenschaften von guten Tests sind: automatisiert, wiederholbar, spezifisch.

13.1. Teststrategie

Die Teststrategie sollte fachgerecht, risikoorientert und wirtschaftlich definiert werden. Passende Testmethoden und Testarten sollten als erstes auf kritische Komponenten angewandt werden, da wo das Risiko am grössten ist.

Bei der Teststrategie sind zudem folgende **Dimensionen** zu berücksichtigen:

- Qualitätskriterien (Bsp. nach ISO 9126)
  - Änderbarkeit
  - Effizienz
  - Übertragbarkeit
  - Zuverlässigkeit
  - Funktionalität
  - Benutzbarkeit
- Hierarchiebenen
  - Klassisch agile Testpyramide (Unit > Integration > System)
  - Alternative Testpyramide (System > Integration > Unit)
- Automatisierung
  - Nutzen/Ertrag abwägen
  - Automatisiert != Unittest!

13.1.1. Vorgehen

1. Wie soll getestet werden? (Unit oder Integration)
2. Welche Qualitätskriterien sollen getestet werden?
3. Soll automatisiert werden?

13.1.2. Integrationstest

Ein Integrationstest prüft die Schnittstellen und das Zusammenspiel von mehreren Komponenten. Idealerweise sollten die einzelnen Komponenten bereits (so weit wie möglich) erfolgreich getestet sein.

Integrationstests testen folgendes:

- Schnittstellen (Objektkompatibilität, Aufrufsequenzen, Inputvalidierung)
- Datenabhängigkeiten (indirekte Abhängigkeiten über gemeinsame Daten, z.B. Datenbank)
- Abdeckung des Call-Graph (Komponenten welche von mehreren Aufrufern gleichzeitig genutzt werden)

13.1.2.1. Hilfsmittel

Mit Äquivalenzklassenanalyse werden äquivalente Inputs in Tests zusammengefasst, da i.d.R. nicht alle Kombinationen getestet werden können.

Beispiele: Alle positiven Werte, alle negativen Werte, alle maximalen Werte, usw.

13.1.3. Systemtest

Mit Systemtests wird die **gesamte Wirkungskette** in einem Softwareprodukt getestet.

Systemtestfälle lassen sich in der Regel aus folgenden Punkten herleiten:

- Akzeptanzkriterien (Scrum)
- Anforderunngen (funktional/nicht-funktional)
- Use-Case-Beschreibungen

Systemtests werden als **Regressionstests** in weiteren Entwicklungsschritten immer wieder gebraucht.

13.1.3.1. Dokumentation

Wichtige Bestandteile der Dokumentation von Systemtests:

- Vorbedingungen für Testausführung
- Handlungen und Eingaben für die Durchführung des Tests

- Erwartete Ergebnisse und Nachbedingungen

HINWEIS: Automatisierung dokumentiert Systemtests am besten.
(Teil-)Automatisierung lohnt sich!

13.1.4. Testing in Scrum

Entweder Testing als Teil der *Definition of Done* oder als eigenen Task.

- Testing bei Aufwandschätzungen immer einberechnen
- Tests möglichst bald durchführen, nicht anhäufen

14. Automatisiertes Testing

14.1. Test-Doubles

15. Fehlertoleranz und Resilienz

15.1. Definitionen

Verfügbarkeit
- Sofort einsatzbereit
- Bezieht sich auf einen bestimmten Zeitpunkt
- Nicht verfügbar bei Ausfall oder Abschaltung

Hochverfügbarkeit
- Mit sehr hoher Wahrscheinlichkeit für Einsatz bereit.

Zuverlässigkeit
- Zeitintervall innerhalb dessen ein System verfügbar ist

Wartbarkeit
- Wie schnell kann ein ausgefallenes System wieder hochgefahren werden

Betriebssicherheit
- Risiko katastrophaler Folgen bei Systemausfall.

Fehler
- Ursache einer Funktionsstörung

Fehlertoleranz
- System kann trotz Vorkommnissen von Fehlern seine Dienste anbieten

Resilienz
- Widerstandfähigkeit gegenüber schwerwiegenden Fehlerereignissen.
- Diese Ereignisse müssen trotzdem vorhersehbar sein.

Resilienz durch Redundanz
- Schutz gegen **Systemausfall**: Systeme oder Daten mehrfach vorhanden.
- Schutz gegen **byzantinische Fehler** (korrupte Daten) -> erfordert Consensus-Protokolle.

Resilienz durch Wiederherstellbarkeit
- System kann nach Ausfall innert kurzer Zeit wiederhergestellt werden
- operiert exakt wie vor dem Ausfall (-> Wartbarkeit).

15.2. Consensus Protokolle

Consensus
- Wenn redundante Systeme **unterschiedliche Ergebnisse** liefern, muss entschieden werden, **welches dieser Ergebnisse verwendet wird**.

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

15.3. Fehlerarten verteilte Systeme

Zielsystem nicht erreichbar

- Daten / Messages können nicht verschickt werden

Auslassungsfehler: Falsche Reihenfolge der Daten / Messages

- Datenkorruption / Programmierfehler

Abrupter Verbindungsabbruch durch Netzwerkausfall

- Folge: Verlorene Daten (z.B. Messages)

Abrupter Verbindungsabbruch durch Systemausfall

- Folge: Verlorene Daten (z.B. Messages) oder inkonsistente Daten

15.3.1. Beispiel RPC

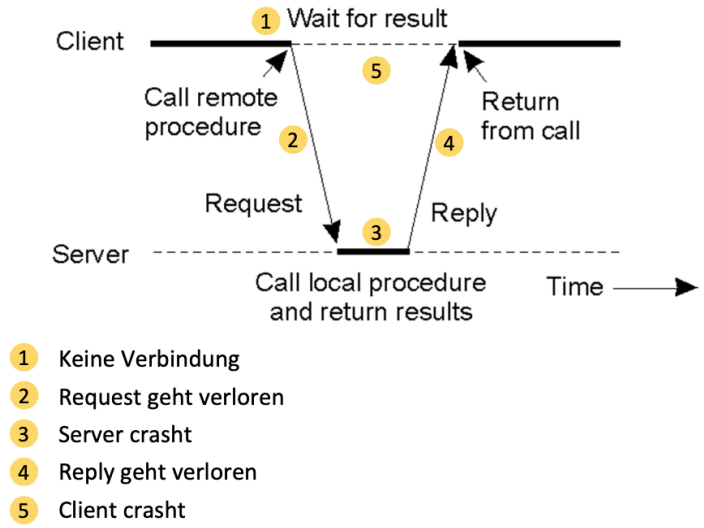


Abbildung 35: Fehler bei einem synchronen Aufruf - RPC

15.4. Server nicht erreichbar vor Verbindungsaufbau

Gegenstelle nicht erreichbar vor Aufbau einer Verbindung / Senden einer Anfrage

Abbildung 36: Fehler: Server nicht erreichbar vor Verbindungsaufbau

Erkennung

- Aufbau der Verbindung schlägt fehl

Massnahmen

- Kein Betrieb möglich => User informieren
- Reduzierter Betrieb, falls Server nur Zusatzfunktionalität anbietet
- Verwendung eines lokalen Caches

15.4.1. Lösung: Lokaler Chache

Beim Lesen:

- Beziehe Information aus vorangehender Kommunikation.

Beim Schreiben:

- Führe Schreib-Operationen lokal durch, sende an Server, sobald verfügbar.

Implemenation in Java z.B. mittels `BlockingQueue`, bzw. `java.util.concurrent.LinkedBlockingQueue`

15.5. Verlorene Message (Request oder Reply)

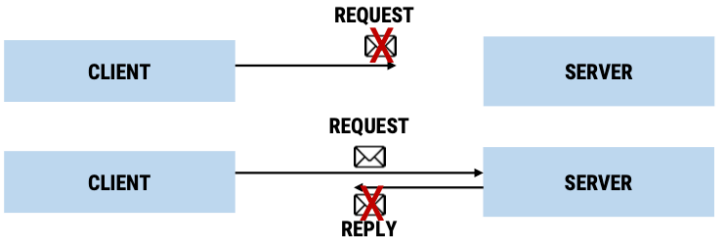


Abbildung 37: Fehler: Verlorene Message (Request oder Reply)

Erkennung
- Client startet Timer bei Absenden eines Requests.
- Ist nach Ablauf des Timers noch keine Antwort eingetroffen, gilt die Message als verloren.

Massnahmen
- Benutzer Informieren / Fragen z.B. bei interaktiven Verbindungen.
- Falls sinnvoll, Request nochmals senden => Duplikatscheck i.d.R. notwendig. (Auslieferungsgarantien)

15.6. Auslieferungsgarantien

At-least-once
- so oft gesendet, bis Antwort eintrifft
- Aktion wird möglicherweise doppelt ausgeführt

At-most-once
- höchstens einmal gesendet
- Aktion wird möglicherweise nicht ausgeführt

Exactly-once
- exakt nur einmal gesendet
- möglicherweise zu teuer oder unnötig

15.6.1. At-least-once & Idempotenz

Idempotente Funktion (math.): Funktion, welche auf sich selbst angewandt das identische 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: $f(x) = -x$

15.6.2. Idempotente Anfragen

- Anfragen sind idempotent, wenn die Funktion auf der Gegenseite idempotent ist.
- Keine Nebeneffekte erlaubt
- Wiederholbar ohne Probleme

15.6.2.1. Beispiel

Idempotent:

- Leseanfragen (z.B. Fahrplanabfrage)
- Adresse ändern A ⟶ B (ohne weitere Effekte)

Nicht-Idempotent

- Bestellung / Reservation ⟶ erstellt neues Element

15.7. At-least-once & Duplikatserkennung

Simulation von **exactly-once** mittels **at-least-once** und **Duplikatserkennung**.

Duplikatserkennung entweder durch:

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

15.7.1. Heuristik

Mit gewisser Wahrscheinlichkeit durch Heuristiken:

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

15.7.2. Sequenznummern

- Jeder Request mit eindeutiger Sequenznummer
- Server speichert Sequenznummer + Ausführung
- Grosse Nummern ⟶ grössere Messages
- Variante: Server gibt nächste Sequenznummer in Response an

UUID: eindeutiger Bezeichner pro Request

Idempotence Key: Client liefert eindeutigen Schlüssel zur Erkennung von Wiederholungen

15.8. Resilienz - Wiederherstellbarkeit

15.8.1. Server-Crash während Requestverarbeitung

Abbildung 38: Server-Crash während Requestverarbeitung

Erkennung

Client:

- Erkennt nicht, ob Message verloren ging oder verarbeitet wurde

Server:

- Führt Log über Aktionen ⟶ kann bei Restart erneut abarbeiten
- Achtung: Zusätzlicher Aufwand (Diskzugriffe, meist DB nötig)

Massnahmen

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

15.8.2. Client Crash während Warten auf Antwort

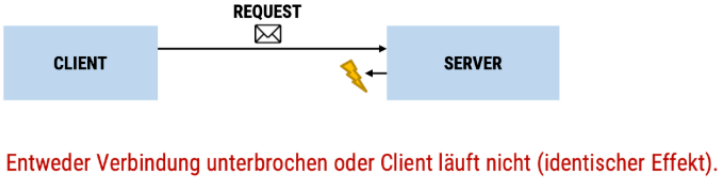


Abbildung 39: Client Crash während Warten auf Antwort

Erkennung

Server:

- Keine Verbindung zu Client möglich.

Massnahmen

- Antwort speichern, falls Client identifizierbar (=> z.B. mittels Token).
  - Client könnte erneuten Request stellen => Duplikatserkennung.
- Antwort verwerfen, falls Client nicht identifizierbar
- Problematisch, z.B. falls ein Client Ressourcen blockieren kann.

15.8.3. Fehlende Konsistenz bei Duplikatserkennung

Wie sicherstellen, dass Sequenznummer N als verarbeitet markiert wird, wenn der Request ausgeführt wurde?

Reihenfolge hilft nicht:

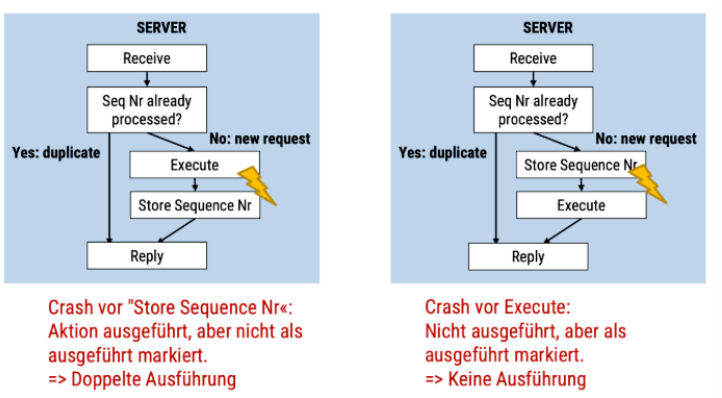


Abbildung 40: Fehlende Konsistenz bei Duplikatserkennung

Lösung: Transaktionen

- Ausführung der beiden Schritte in einer Transaktion
  - Entweder beide Aktionen ausgeführt oder keine

15.9. Transaktion

Atomare Einheit der Ausführung

- **Entweder alles ausgeführt oder nichts.**

I.d.R. von Datenbanken und Message-Oriented-Middleware umgesetzt

Ablauf

1. Transaktion starten
2. Mehrere zusammengehörige Operationen innerhalb der Transaktion durchführen
   - CRUD-Operationen
3. Transaktion entweder
   - erfolgreich abschliessen (commit)
   - abbrechen (rollback)

15.10. Verteilte Transaktionen

Ziel:

- Zusammenfügen (merge) lokaler Transaktionen von zwei oder mehr Systemen.
- Alle Transaktionen werden entweder ausgeführt oder nicht

Umsetzung

- Transaktionsmanager (Bittronix, JBossTS, Atomikos, ...)

Voraussetzung

- Teilnehmende Systeme (Datenbank, Message-oriented-Middleware, ...) verfügen über einen kompatiblen Transaktionsmechanismus.

Grapshisch:

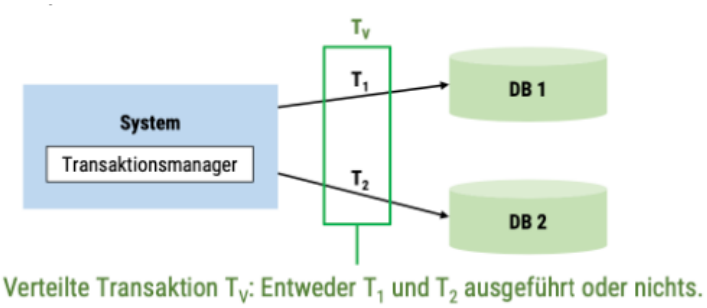


Abbildung 41: Verteilte Transaktionen

15.10.1. Exactly-once Kommunikation mit verteilter Transaktionen - Message-oriented Middleware

Message-oriented Middleware (MOM)

- MOM M unterstützt Transaktionen via Transaktionsmanager
- Alle Systeme kommunizieren ausschliesslich über MOM
- Integration von M in alle beteiligten Systeme
- Aktionen + Datenzugriffe erfolgen innerhalb verteilter Transaktion

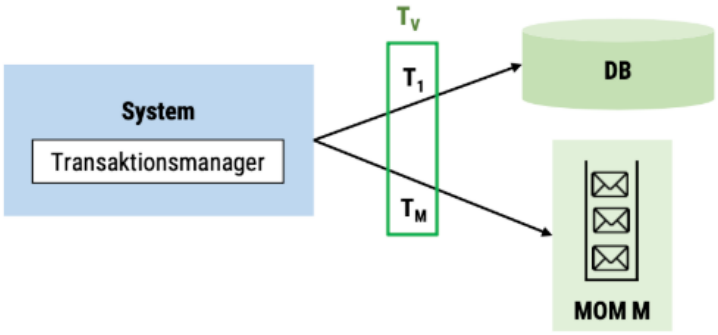


Abbildung 42: Message-oriented Middleware (MOM)

15.10.2. Funktionsweise verteilter Transaktionen - Two-Phase-Commit

Two-Phase-Commit (2PC)

- Standard für verteilte Transaktionen
- Koordinator (z.B. erste Transaktion mit Commit) steuert Ablauf

Phase 1:

- Koordinator fragt alle Systeme: Can you commit? ⟶ YES / NO

Phase 2:

- Alle committen bei nur YES
- Abbruch bei mindestens einem NO

15.10.2.1. Detaillierter Ablauf

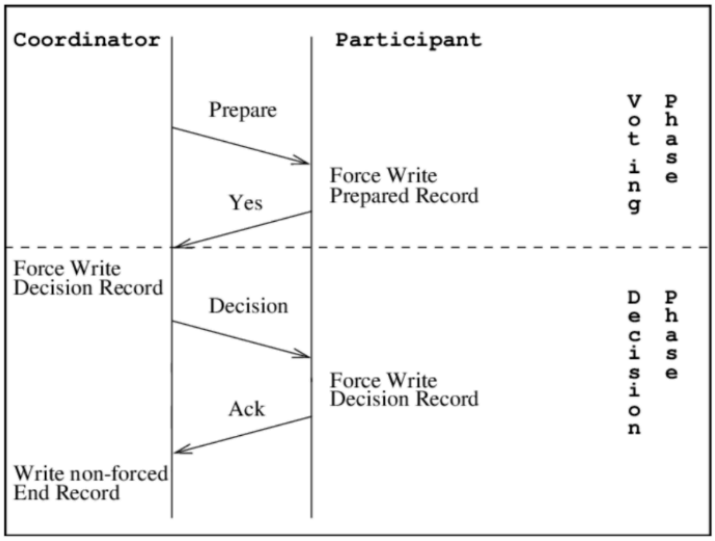


Abbildung 43: Two-Phase-Commit

Voting Phase

- Koordinator sendet **PREPARE** an alle Teilnehmer
- Teilnehmer antworten mit **YES** (bereit) oder **NO** (nicht möglich)

Decision Phase

- Koordinator sammelt Antworten
  - **Alle YES** → sende **COMMIT**
  - **Mind. ein NO** → sende **ABORT**
- Teilnehmer führen Anweisung aus (commit oder rollback)
- Bei Koordinator-Crash: andere Teilnehmer können Entscheidung anfragen

16. Entwurfsmuster (design pattern)

Objektorientierte Entwurfsmuster nach GoF (Gang of Four) -> Erich Gamma, Richard Helm, Ralph Johnson und John Vlissides -> Buch „Design Pattern“ (1995)

DEFINITION: Elemente wiederverwendbare, objektorientierter Software oder Bewährtes objektorientierte Entwürfe (Schablonen) für ein wiederkehrendes Entwurfsproblem.

16.1. Wiederverwendung in der Softwareentwicklung

16.1.1. Ziel

Wiederverwendung von bewährten Entwurfsmustern

16.1.2. Arten der Wiederverwendung

- Wiederverwendung von Objekten zur Laufzeit
- Wiederverwendung von Quellcode / Klassen
- Wiederverwendung von einzelnen Komponenten
- Einsatz von Klassen-Bibliotheken / Frameworks
- Wiederverwendung von Konzepten (Entwurfs-, Architektur- oder Kommunikationsmuster)

16.1.2.1. Objekte

DEFINITION: Nutzung von Objekten mehrfach zur Laufzeit

Beispiele:
- Threads in einem `Executor`-Pool
- Datenbankverbindungen im Connection-Pool

Effekte:
- Bessere Performance
- Weniger Ressourcenverbrauch

Herausforderungen:
- Effiziente Objektverwaltung
- Einsatz von Entwurfsmustern zur Optimierung

16.1.2.2. Quellcode / Klassen

DEFINITION: Wiederverwendung von Quellcode / Klassen

Wiederverwendung durch:
- Copy & Paste ⟹ schlecht (vgl. DRY-Prinzip)
- Vererbung ⟹ oft problematisch
- Aggregation & Komposition ⟹ gut (vgl. „Favor Composition over Inheritance“)

Effekte:
- Weniger Entwicklungsaufwand
- Geringere Fehlerrate durch getestete Klassen

Herausforderungen:
- Schnittstellen oft fremdbestimmt
- Auswahl geeigneter Klassen/Bibliotheken
- Lernaufwand, Wartung, Weiterentwicklung

16.1.2.3. Komponenten

DEFINITION: Wiederverwendung von einzelnen Komponenten

Beispiele:

- Logging-Komponente
- Jakarta EE-Beans
- Corba-Komponenten

Effekte:
- Geringerer Entwicklungsaufwand
- Weniger Fehler
- Blackbox-Prinzip, starke Abstraktion

Herausforderungen:
- Anforderungen an Umgebung/Kontext
- Inkompatible Schnittstellen ⟹ Entwurfsmuster helfen
- Komponentenverwaltung (z. B. Konfigurationsmanagement)
- Abhängigkeit von Lieferanten
- Wartung und Weiterentwicklung

16.1.3. Herausforderung bei Wiederverwendung
- Unterschiedliche Kontexte / Fachverständnisse
- Unterschiedliche Technologien / Lösungsansätze
- Aufwand bei Weiterentwicklung & Wartung
- Komplexes Konfigurationsmanagement
- Inkonsistente Designkonzepte
- Abhängigkeit von Drittanbietern

Fazit:
- Wiederverwendung von Quellcode bleibt eine grosse Herausforderung!

16.1.4. Konzepte
- Stabil, langfristig gültig
- Breit anerkannt, erprobt, sprach- & implementierungsunabhängig
- Wiederverwendung von Entwurfsmustern = elegante, effektive, kostensparende Lösung

Vergleiche:
- Kommunikationsmuster (z. B. Handshaking)
- Architekturmuster (z. B. Client/Server, Schichtenmodell, MVC)

16.2. Design Patterns - Klassifikation

- Rund 20 dokumentierte Entwurfsmuster.
- Buch „Entwurfsmuster“ (1995)
- „Gang of Four“ (**GoF**)

Klassifiziert in 3 Gruppen:

- **Erzeugungsmuster** (Creational Patterns)
- **Strukturmuster** (Structural Patterns)
- **Verhaltensmuster** (Behavioral Patterns)

sekundäre Unterteilung

- Klassenmuster: Festlegung der Beziehung zum Kompilierzeitpunkt
- Objektmuster: Beziehung zu Laufzeit veränderbar

16.2.1. Creational Patterns (Erzeugungsmuster)

Abstrahieren der Objekterzeugung

- Wahl des dynamischen Typs
- Zeitpunkt der Erzeugung (z. B. lazy)
- Konfigurationsweise (z. B. Kontext, Initial-Konfiguration)

Delegation der Objekterzeugung

- Fabrik-Konzept:
  - Objekt wird angefordert
  - Instanziierung & Konfiguration werden versteckt
  - Nutzer braucht sich nicht um Details zu kümmern

Patterns:

- Abstract Factory, Kit (Abstrakte Fabrik)
- Builder (Erbauer)
- Factory Method, Virtual Constructor (Fabrikmethode)
- Prototype (Prototyp)
- Singelton (Einzelstück)

16.2.2. Singelton (Einzelstück)

- Stellt sicher, dass **nur eine einzige Instanz** einer Klasse existiert
- Bietet einen **zentralen Zugriffspunkt** auf diese Instanz

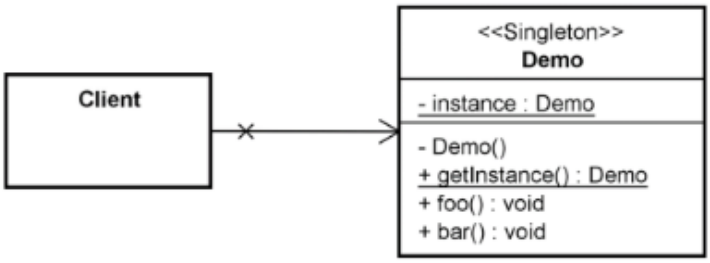


Abbildung 44: Singelton

Wesentliche Eigenschaften:

- Privates, statisches Attribut für Instanz
- öffentliche, statische Zugriffsmethode
- Privater Konstruktor (verhindert externe Instanzierung)

Kritik:

- Schlechter Ruf (sogar Gamma distanziert sich)
- Führt zu starker Kopplung
- Austausch der Instanz ist aufwendig

Empfehlung:
- Sehr sparsam und gezielt einsetzen
- **Niemals** als globalen Zugriffspunkt verwenden (Missbrauch!)

16.2.3. Structural Patterns (Strukturmuster)
- Objekte/Klassen zu grösseren/veränderten Strukturen zusammenfassen
- Unterschiedliche Strukturen anpassen und verbinden

Patterns:

- Adapter (Adapter)
- Bridge (Brücke)
- Decorator (Dekorierer)
- Facade (Fassade
- Flyweight (Fliegengewicht)
- Composite (Kompositum)
- Proxy (Stellvertreter)

16.2.4. Adapter
- Passt die **Schnittstelle einer vorhandenen Klasse** an die **erwartete Ziel-Schnittstelle** an
- Ermöglicht die Nutzung inkompatibler Klassen ohne deren Änderung

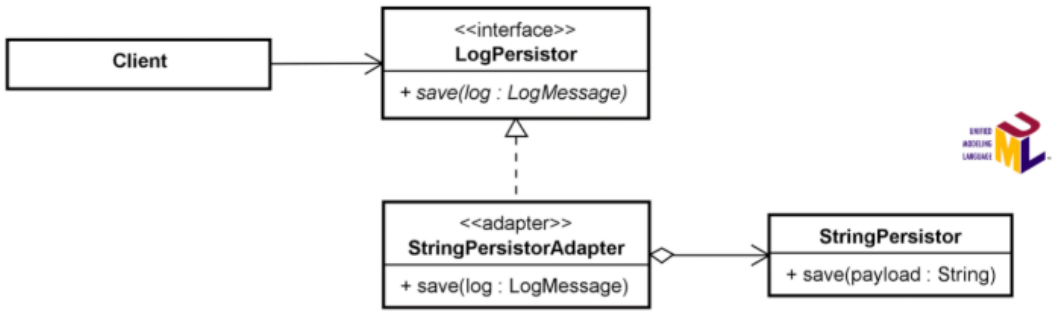


Abbildung 45: Singelton

Motivation
- **Wiederverwendung** von existierenden Klassen/Komponenten trotz unpassender Schnittstelle
- Eine **allgemeine Ziel-Schnittstelle definieren** und durch **Adapter anpassen**

Ziel-Interface (z. B. `LogPersistor`)
- Gewünschte Schnittstelle für den Nutzer
- Kann Interface **oder abstrakte Klasse** sein

Adapter (z. B. `StringPersistorAdapter`)
- Verwendet die vorhandene Klasse (z. B. durch Komposition)
- **Implementiert oder erweitert** die Ziel-Schnittstelle
- Vermittelt zwischen Ziel und adaptiertem Objekt

Adaptiertes Objekt (z. B. `StringPersistor`)
- Bestehende Klasse mit inkompatibler Schnittstelle
- Wird durch den Adapter „verpackt“ bzw. **angepasst (wrapped)**

16.2.5. Facade (Fassade)
- Bietet eine **einheitliche, zusammengefasste Schnittstelle**
- Dient dem Zugriff auf mehrere Schnittstellen von Subsystemen

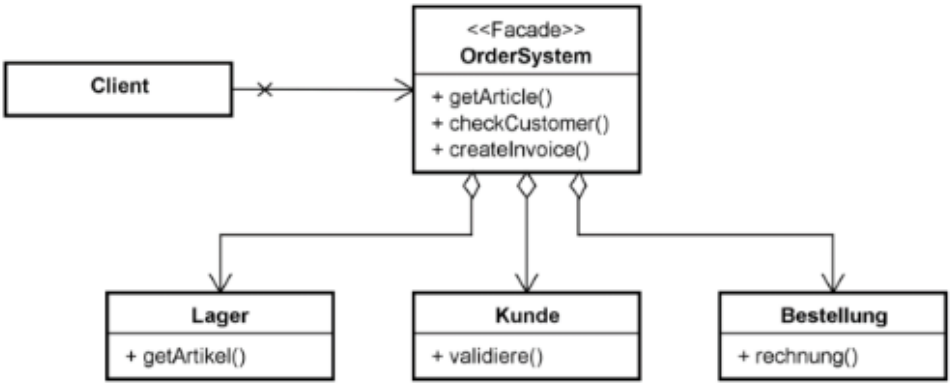


Abbildung 46: Facade

Wesentliche Eigenschaften
- Vereinfacht Nutzung mehrerer Subsysteme
- Reduziert Abhängigkeiten → **Kopplung minimieren**
- Erleichtert den Austausch einzelner Subsysteme

Gefahr:
- Fassade wird zum „Durchlauferhitzer“ ohne Nutzen

Empfehlung:
- Fassade gezielt zur Entkopplung einsetzen
- **Nur delegieren**, keine eigene Logik einbauen

16.3. Behavioral Patterns (Verhaltensmuster)
- Beschreiben die **Interaktionen zwischen Objekten**.
- Legen die **Kontrollflüsse zwischen den Objekten** fest.
- **Zuständigkeit und/oder Kontrolle** delegieren.

Patterns:

- **Command (Befehl)**
- **Observer (Beobachter)**
- **Visitor (Besucher)**
- **Interpreter (Interpreter)**
- **Iterator**
- **Memento**
- **Template Method (Schablonenmethode)**
- **Strategy (Strategie)**
- **Mediator (Vermittler)**
- **State (Zustand)**
- **Chain of Responsibility (Zuständigkeitskette)**

16.3.1. Observer - Event/Listener (Beobachter)
- Definiert **Abhängigkeit zwischen einem Subjekt** (`Observable`)
- und **mehreren Beobachtern** (`Observer`)
- Wenn sich der Zustand des Subjekts ändert, werden die Observe **automatisch informiert**

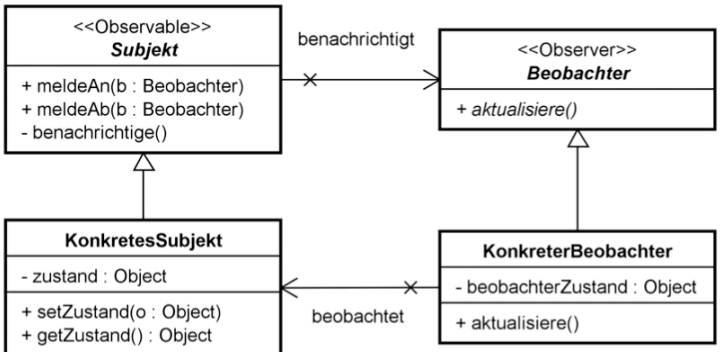


Abbildung 47: Observer

Subjekt – `Observable`
- Verwalten von 0..n Beobachtern
- Methoden zur **An und Abmeldung** von Beobachtern

Beobachter – `Observer`
- Definiert eine **Benachrichtigungsschnittstelle**

Konkrete Implementierungen
- **Konkretes Subjekt**: sendet Updates
- **Konkreter Beobachter**: empfängt und verarbeitet Updates

Motivation
- **Lose Kopplung** zwischen Subjekt und Beobachtern
- Anzahl der Beobachter ist irrelevant
- Kommunikation **entgegen der Abhaengigkeitsrichtung**
- Auflösung zyklischer Referenzen

Typische Anwendung: MVC
- Modell-Änderungen ⟶ automatische Aktualisierung der Views
- **MVC** = Model View Control ⟶ klare Trennung der Verantwortlichkeiten bei GUI-Apps

16.3.1.1. Event Listener in Java
- **Event/Listener-Modell** macht das Ganze eleganter
- Grundlage für **ereignisorientierte Programmierung**

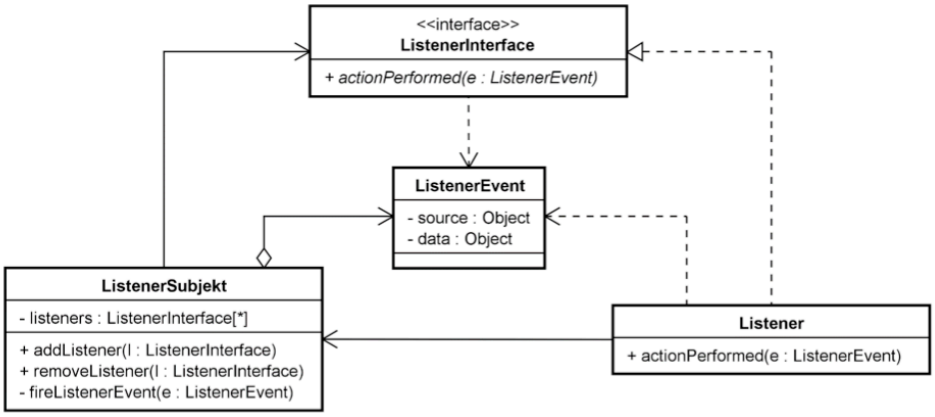


Abbildung 48: Event Listener in Java

- **Event/Listener-Modell** in Java ist moderner & flexibler als das klassische GoF-Pattern
- **Java Interfaces** ermöglichen eleganteres Design (keine Vererbung nötig)

⟶ In Java konsequent das Event/Listener-Modell verwenden!

16.3.2. Strategy (Strategie)
- Definiert eine **Familie von Algorithmen**

- Kapselt jeden Algorithmus einzeln
- Algorithmen sind **austauschbar**
- Client bleibt **unabhängig vom verwendeten Algorithmus**
- Open Closed Principle

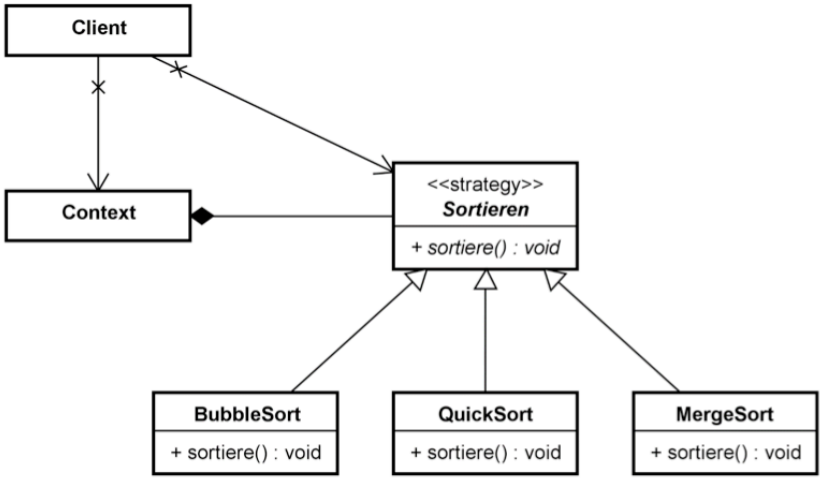


Abbildung 49: Strategy

Strategie (z. B. `Sortable`)
- Vollabstrakte Klasse oder Interface
- Definiert die Schnittstelle für alle Strategien

Kontext (z. B. `Context`)
- Optional – kann vom Client oder Kontext selbst genutzt werden
- Hält Referenz auf konkrete Strategie (und erstellt sie ggf. selbst)
- Stellt ggf. Datenschnittstelle für Strategien bereit

Hinweis zu Programmiersprachen (z. B. Java)
- Bei fehlender Mehrfachvererbung → **Interfaces statt abstrakter Basisklassen** verwenden
- **(Voll-)abstrakte Klassen werden durch Interfaces ersetzt**
- **Mit Interfaces wären die Pfeile der einzelnen Sorts gestrichelt**

Einsatzmotivationen:
- Verschiedene Varianten von Algorithmen (z. B. Zeit vs. Speicher)
- Klassen unterscheiden sich nur im Verhalten → zusammenfassen
- Vermeidung vieler **if/switch-Anweisungen**

Empfehlung:
- Wird oft unterschätzt, lohnt sich aber auch bei kleinen Methoden
- Eliminiert grosse, unübersichtliche **switch-Statements** elegant

16.4. Empfehlung zu Design Patterns

16.4.1. Voraussetzungen
- Entwurfsmuster kennen und verstehen
- Informationsquellen: Literatur, Internet

16.4.2. Sinnvoller und überlegter Einsatz
- Muster sind **nicht die ultimative Lösung** für alles
- Erfahrung sammeln ist notwendig
- **Besser kein Muster als das falsche**

16.4.3. Musterwahl ist schwierig
- Ist es ein **Erzeugungs-, Struktur- oder Verhaltensmuster**?
- Vorselektion nach Aufgabenbereich

- Auswahl nach Vorteilen (z. B. Flexibilität, Vereinfachung)
- Wenn unklar: Muster mit **grösster Flexibilität** wählen

16.4.4. Verifikation vor Einsatz

- Muster mit **echten oder fiktiven Beispielen** testen
- Prüfen:
  - Weitere Strategien vorstellbar?
  - Zugriff auf notwendige Daten?
  - Konfigurierbarkeit gegeben?
  - Wird es **einfacher oder komplizierter**?

16.4.5. Muster nicht blind übernehmen

- Muster sind **Konzepte**, keine Dogmen
- Dürfen **angepasst, vereinfacht, optimiert** werden
- Sprache kann Umsetzung vereinfachen
- Unbedachte Änderungen können das Design zerstören

16.4.6. Erfahrung und Augenmass nötig

- **OO-Design ist nicht einfach**, aber spannend!

16.4.7. Kombination von Mustern

- Kommt oft vor, wird aber selten erwähnt
- **Effizienz durch Kombination möglich**, aber:
  - **Mehr Komplexität**
  - **Muster schwerer erkennbar**

Beispiele:
- **Fabrik** für **Zustände**
- **Fassade** für **Fabriken von dekorierten Strategien**

17. Konsistenz und Replikation

17.1. CAP Theorem

Consistency Alle beteiligen Systeme haben identische und aktuelle Daten
Availability Daten sind für alle Lese-/Schreiboperationen sofort bereit.
Partition Tolerance System kann trotz Aufteilung in zwei oder mehr bzgl. Kommunikation getrennte Partitionen weiter operieren

- ein Verteiltes System kann max. 2 dieser 3 Garantien gleichzeitig erfüllen

17.1.1. Systemausprägungen

Da alle 3 Eigenschaften nicht alle aufs mal realisiert werdne können, müssen in gewissen Punkten abstriche gemacht werdne, woraus dann ein AP, CP oder CA-System entsteht.

	AP-Systeme	**CP-Systeme**	**CA-Systeme**
Beschreibung	operieren trotz ausgefallener Kommunikation weiter	operieren trotz ausgefallener Kommunikation weiter, aber nur so weit die Konsistenz gewährleistet ist.	bieten sowohl Konsistenz als auch Verfügbarkeit dürfen darum nicht partitioniert werden
Reduktion der	Konsistenz	Verfügbarkeit	Partitionstoleranz
Beispiele	Domain-Name-System, Consumer-Filesynchronisation	Email, Bankomat	Webapplikation

17.2. Konsistenz

Sobald ein System A und ein System A' über die gleichen Daten verfügen (z.B. wenn A' ein Replikat von A ist), stellt sich die Frage bzgl. deren Konsistenz.

DEFINITION: Liefern A und A' bei identischen Anfragen $X == X'$ die identischen Antworten $Y == Y'$ sind die Daten der beiden Systeme konsistent.

17.2.1. Konsistenzgarantien

Sequential Consistency Die Sequenz der Operationen eines Gesamtsystems S ist für alle Teilsysteme von S die gleiche und passt in eine Gesamtordnung.
Eventual Consistency Modifikationen werden erst mit einer gewissen Verzögerung sichtbar.

17.2.1.1. Sequential Consistency

«Sequential consistency» gibt folgende Konsistenzgarantie: Das Resultat einer verteilten Ausführung ist das gleiche wie…

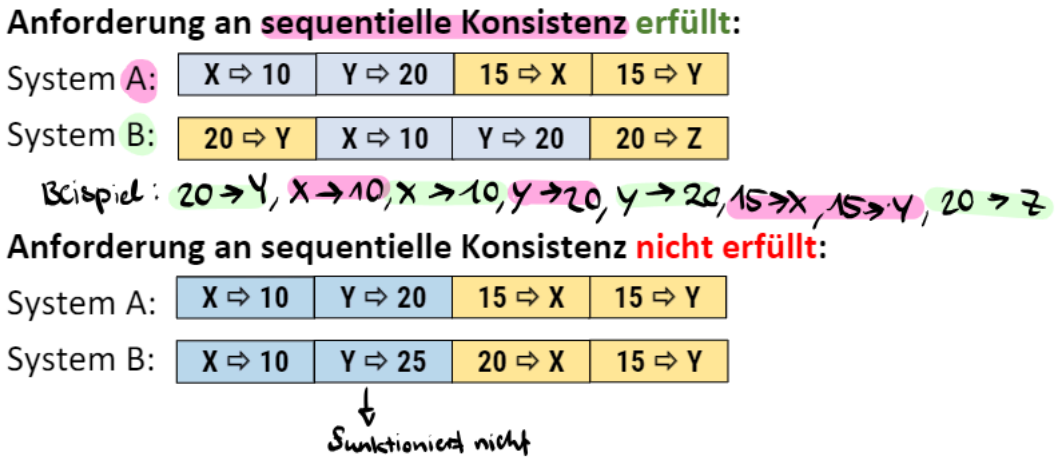
- wenn Lese- und Schreiboperationen aller beteiligen Systeme einer globalen sequentiellen Abfolge zugeordnet werden können,
- und in dieser Abfolge für jedes System die Operationen in der gleichen Reihenfolge, in welcher es die Operationen getätigt hat, vorkommt.
- Die Zeit spielt dabei keine Rolle!

Typische Anforderung einer verteilten Eingabe und Verarbeitungapplikation.
- Vorteil: Unabhängige Teiloperationen blockieren das Gesamtsystem nicht.

Für sequentielle Konsitenz muss Schreiben auf die Disk erzwungen werden, sonst werden Änderungen zuerst nur in einen Cache geschrieben.

`java.nio.channels.FileChannel` ->
`public abstract void force(boolean metaData)) throws IOException`

Beispiel



17.2.1.2. Eventual Consistency

Gesamtsystem konvergiert gegen aktuelle Daten. Ein System kann aber zu einem beliebigen Zeitpunkt noch mit veralteten Daten arbeiten

Wird mir **Monotonic Reads** verbunden: Wert ist immer gleich oder aktueller als der letzte gelesene Wert

Vorteile

- Zeitliche Entkopplung
- Verfügbarkeit
- tolerieren von Netzpartitionierung

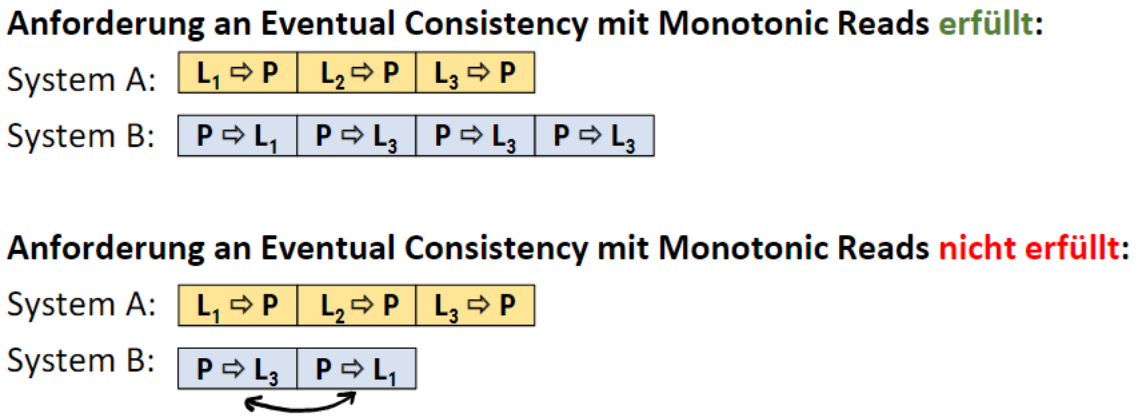
Einsatzszenario

- Wenn Daten nicht zwingend immer aktuell sein müssen und nur eine Instanz die Hoheit über Änderungen hat

ODER

- Wenn nur Schreiboperationen getätigt werden

Beispiel



17.3. Replikation

DEFINITION: Kopie eines Systemes welches zwar nicht immer Bit für Bit identisch sein muss aber die gleichen Informationen enthält

Aus einem Original (Primary) wird eine Kopie (Replikat) erstellt

Anwendung

- Ausfallsicherheit: Beim Ausfall von A übernimmt B
- Performance: Anfragen an beide Systeme verteilen

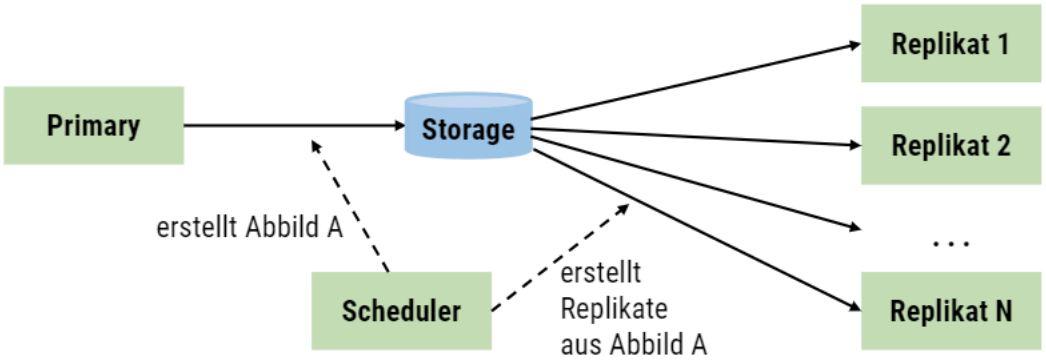
Fragesellungen

- Konsistenz
- Wie **erkennt** man einen Ausfall?
- Wie **toleriert** man einen Ausfall?
- Wo müssen Primary und Replika plaziert sein?
- Wie wir die Last verteilt?

17.3.1. Klassische Replikation

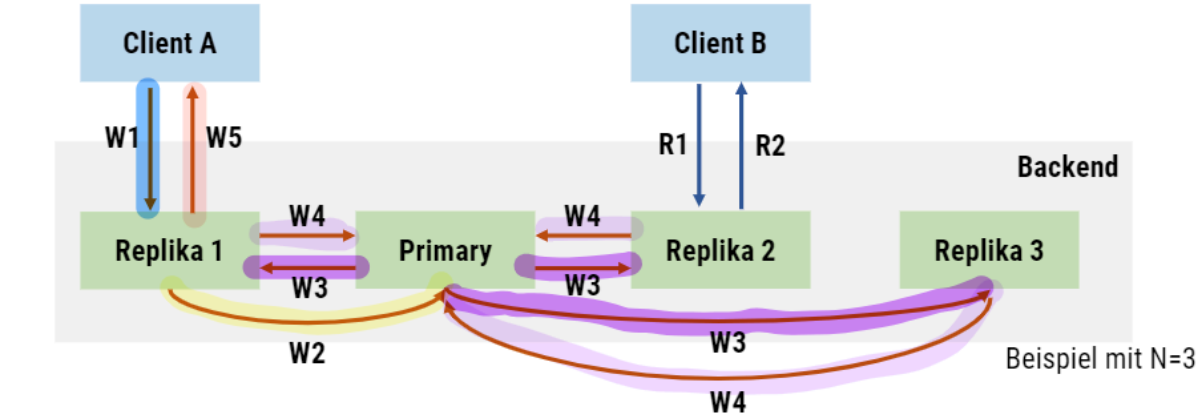
Ein Scheduler macht regelmässig folgendes:

1. Erstellung von einem Abbild A des Primarys
2. Einspielung von Abbild A in alle Replikatsysteme



17.3.2. Primary Backup Protokoll „On the fly“

- Primary behandelt alle Schreibzugriffe
- Replikas
  - Beantwortet Lesezugriff
  - Leitet Schreibzugriff 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

⟶ Standardimplementation: Schreibanfrage blockierend! (Langsame Antwortzeit)

17.3.2.1. Variationen

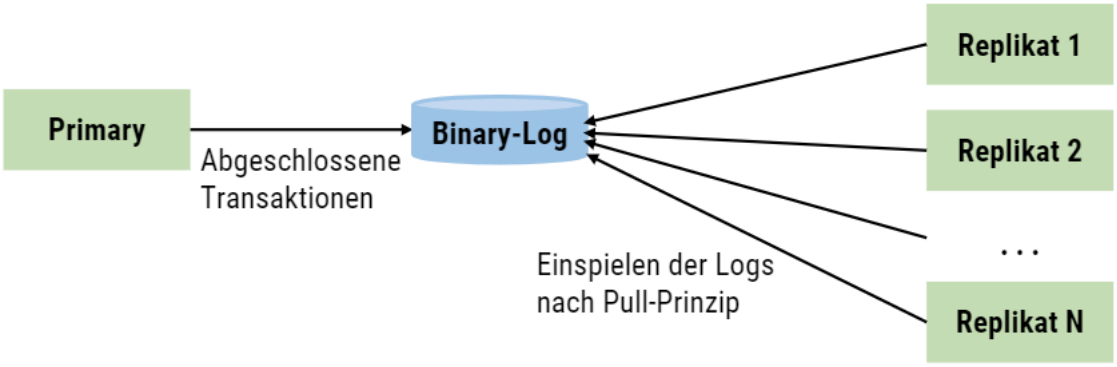
Viele Produkte verwenden eine abgeänderte Version

- Nicht blockierender Schreibzugriff
  - Schnellere Antwortzeit dafür evt. Datenverlust und Downtime bei einem Crash
- Push vs Pull Prinzip
  - Anstatt sequentielle Konststen, Eventual Consisten
- Einzelner vs verteilte Primary
  - System kann zeitgleich Primary und Replikat sein
  - Komplexer aber schneller

17.3.2.2. Standard Replikation bei MariaDb

1. Abgeschlossene Transaktionen werden in Binary Log geschrieben
2. Replikas leses aus dem Binary Log und akzeptieren es lokal (Pull)

⟶ Replikas verfügen nur mit Verzögerung über die gleichen Daten



17.3.2.3. Implementation Push Prinzip

```java
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);
        }
    }

    public void addMessage(Message message) {
        replicationQueue.add(message);
        notifyWaiters();
    }

    public void waitForSynchronization() {
        waitForCondition(true);
    }
```

```
    private void processReplicationQueue(ZMQ.Socket socket) {
        while (true) {
            waitForCondition(false);
            Message message = replicationQueue.peek();
            socket.send(message.toString());
            socket.recvStr();
            replicationQueue.remove();
            notifyWaiters();
        }
    }

    private synchronized void waitForCondition(boolean isEmpty) {
        while (replicationQueue.isEmpty() != isEmpty)
            this.wait();
    }
}
```

17.4. Ausfallsicherheit

Ausfallssicherheit kann erreicht werden in dem automatisch auf eine Replika geswitched wird wenn der Primary ausfällt

Dabei müssen folgende Punkte beachtet werden:

- Wie erkennen Replikate das der Primary ausgefallen ist?
  - Heartbeat Protokolle
- Wie ernennen Replikas einen neuen Primary?
  - Leader-Election-Protokolle
- Wie wissen Clients auf welches System sie verbinden müssen?
  - Namensdienste/IP-Routing

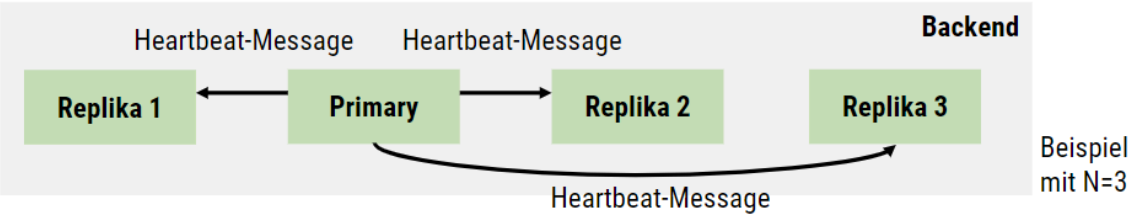
17.4.1. Hearbeat Protokoll

Ziel

Ausfall von Primary erkennen

Vorgehen

- Primary sendet Hearbeat-Message in festem Intervall
- Replikas überprüven ob sie mindestens ein Hearbeat-Signal innerhalb von einem Timeout-Intevall erhalten haben
  - → Ist das nicht der Fall wird der Primary als Down betrachet



Implementation

```
JAVA
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]);
    }
};

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);
}

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();
}
```

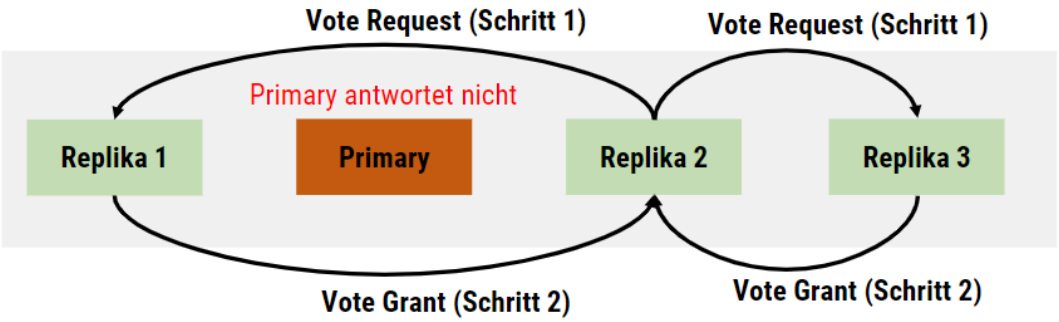
17.4.2. Leader Election mittels Quorum

Ziel

Ernennen eines neuen Leader mittels Mehrheitsentscheid

Vorgehen

- Replikas R erkennen Ausfall des Primary
- Replika R sendet **Vote Request** an alle anderen Replika
- Andere Replika senden **Vote Grant** falls diese mit dem Wechsel einverstanden sind
- Replika R wird neues Primary falls es $\left\lfloor \frac{N}{2} \right\rfloor + 1$ **Vote Grants erhält**



Beispiel mit N=3

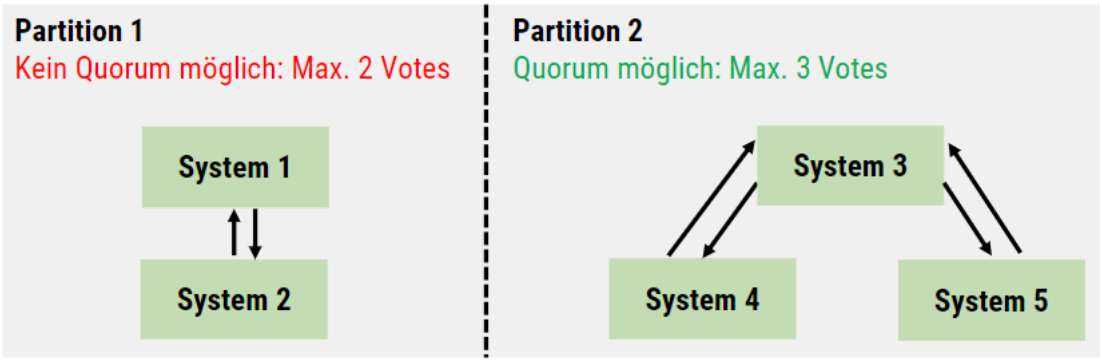
Notwendigkeit eines Quorums

Bei einer Netzwerpartitionierung darf nur eine Partition weiter operieren

Quorum ist sinnvoll ab $N = 3$ Systemen. N sollte ungerade sein.

Beispiel: $N = 5$

$\lfloor \frac{N}{2} \rfloor + 1 = 3$



17.4.2.1. Leader Election mittels externem Store

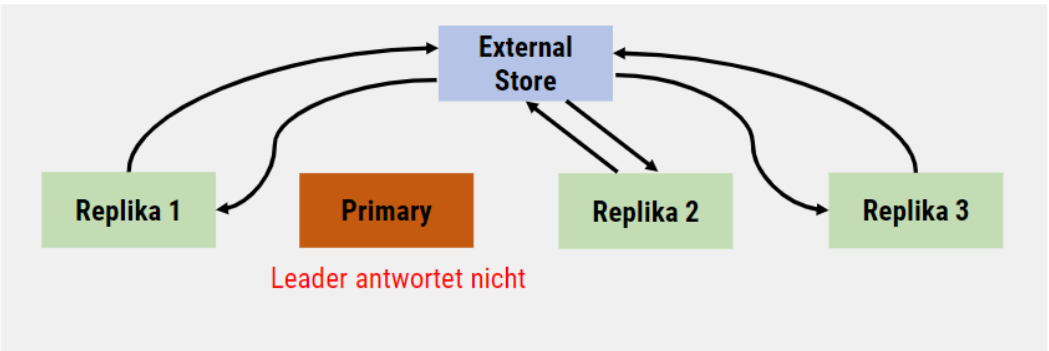
Kleiner ausfallsicherer Store implementiert Quorumalgorithmus.

Beispiel

- ZooKeeper (Java)
- etcd (Kubernetes)

Ablauf

- N Replikas senden Election-Request an zentralen Store
- Eine Replika wird als Primary eingetragen



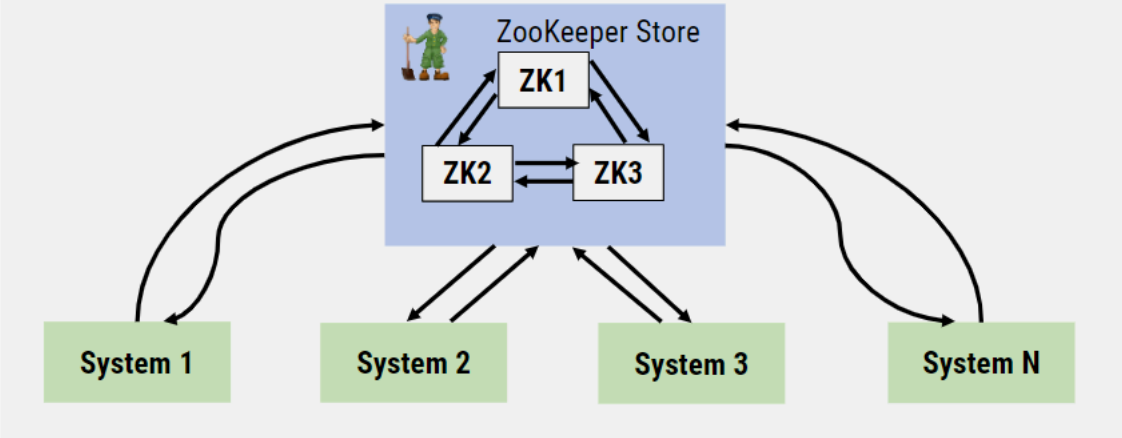
Beispiel mit N=3

17.4.2.1.1. ZooKeeper

ZooKeeper ist eine zentraler Informationsdienst in Java. Er ist Ausfallsicher durch ein Konsensus Protokoll

Angebotene Funktionalitäten

- Namensdienste (Auffinden von Systemen)
- Verteile Synchronisation (systemübergreifende Locks)
- Verwalten von Gruppenzugehörigkeit



Namensraum

ZooKeeper implementiert einen hierarchischen Namensraum mittels einer Baumdatenstruktur

- Nodes können Informationen und/oder weitere Kinder haben
- Nodes sind **PERSISTENT** oder **EPHEMERAL**
  - PERSISTENT: Überdauern Verbindungsabbruch durch Client
  - EPHEMERAL: Überdauern Verbindungsabbruch durch Client nicht

Persistenten Node erstellen

```java
zk.create("/app1/p_3", myData, Ids.OPEN_ACL_UNSAFE, CreateMode.PERSISTENT);
```

Inhalt von Node Abfragen

```java
byte[] data = zk.getData("/app2", event -> setChanged(), stat);
```

Implementation

```java
public final static String PATH = "/myAppLeader";
public final String instance_name = "instanceName"; // 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
    }
}
```

18. Skalierung und Verteilung 📈

18.1. Grundlagen

18.1.1. Topologie verteilter Systeme

18.1.1.1. Embedded

- Für asynchrone Task-Ausführung oder Hochleistungs-Computing
- Jeder Knoten enthält:
  - Anwendung
  - Daten
- Kommunikation über TCP/IP zwischen den Knoten

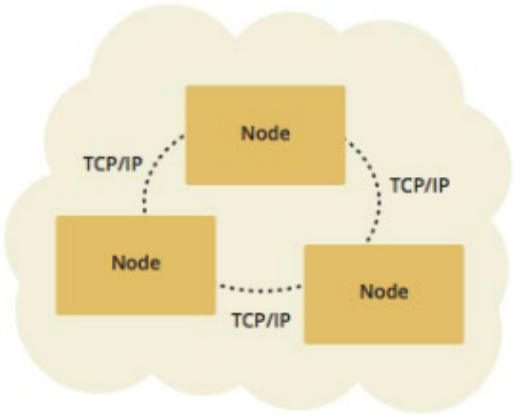


Abbildung 50: Embedded Topologie

18.1.1.2. Client / Server

- Cluster von skalierbaren Server-Knoten
- Kommunikation durch externe Clients:
  - Native Clients (java, .NET, C++)
  - Memcache-Clients
  - REST-Clients
- Clients greifen auf die Daten der Server-Knoten zu
- Kommunikation über spezifische Protokolle (Native, Memcache, REST)

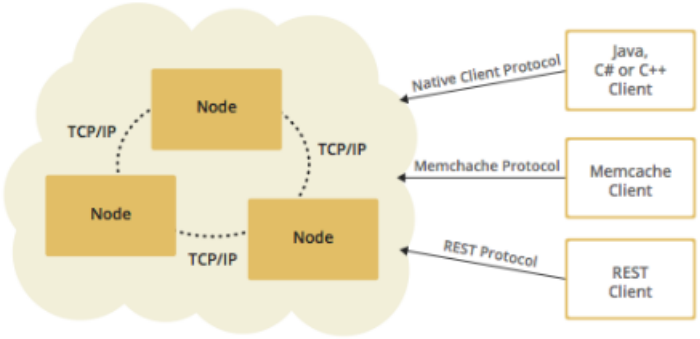


Abbildung 51: Client/Server Topologie

18.1.2. Skalierung verteilter Systeme

18.1.2.1. Zustandslose Systeme

- Anfragen sind unabhängig

- Mehrere Instanzen einer Serveranwendung möglich
- Verteilung der Anfragen je nach Auslastung

→ Einfache Lastverteilung

18.1.2.2. Zustandsbehaftete Systeme mit temporären Daten

- Mehrere Instanzen einer Serveranwendung
- Erste Anfrage einer Session wird nach Auslastung an Instanz N geleitet
- Folgeanfragen gehen erneut an Instanz N

→ Komplexere Lastverteilung durch Inspektion der Kommunikation

18.1.2.3. Zustandsbehaftete Systeme mit persistenten Daten

- Datenreplikation (ggf. Partitionierung)
- Verteilung der Anfragen an Instanzen mit relevanten Daten

→ Komplexeste Lastverteilung (read vs. write, Replikakonsistenz etc.)

18.1.2.4. Beispiel

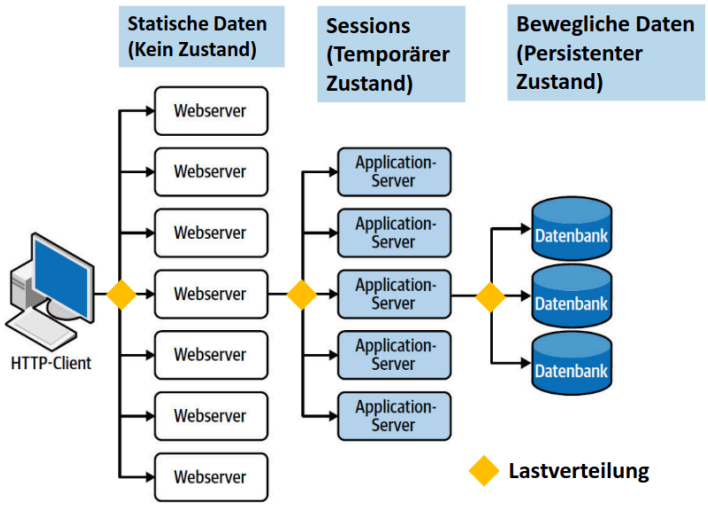


Abbildung 52: Skalierung nach Zustand

18.2. Lastverteilung mittels Reverse Proxys

18.2.1. Reverse-Proxy

- Middleware, die Anfragen an verschiedene Instanzen einer Serveranwendung verteilt

18.2.1.1. Vorteile

- **Load-Balancing**:
  - Verteilung der Last auf mehrere Instanzen
  - Keine Instanz wird überlastet, solange andere noch Kapazität haben
- Zusätzliche Funktionen:
  - Sicherheit (z. B. Verschlüsselung)
  - Monitoring
  - Metriken
  - Kompression

18.2.1.2. Beispiel-Technologien

- HAProxy
- Webserver wie Apache / Nginx
- Traefik

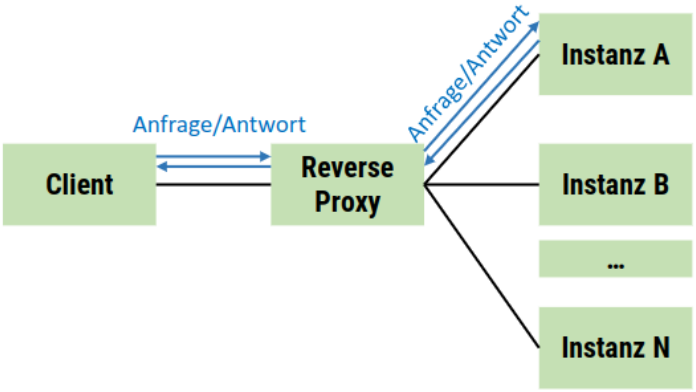


Abbildung 53: Reverse Proxy Prinzip

18.2.2. Lastverteilungsmethoden

18.2.2.1. Round-Robin

- Jede Instanz erhält der Reihe nach eine Anfrage
- Nach letzter Instanz beginnt Proxy von vorne

→ Gut bei homogener Last und homogener Systemausstattung

18.2.2.2. Anzahl bestehender Verbindungen

- Anfrage wird an Instanz mit den wenigsten Verbindungen geschickt

→ Gut für langdauernde TCP-Verbindungen

18.2.2.3. Hash

- Hash-Funktion auf Client-IP zur Auswahl der Zielinstanz
- Alle Anfragen einer IP gehen an dieselbe Instanz

→ Einfach für gleichbleibende Zuweisung (benötigt stabile Client-IPs)

18.2.3. Zusammenspiel mit Serverapplikation

- Mehrere Requests sollen zur gleichen Serverinstanz geleitet werden – unabhängig von der TCP-Verbindung
- Dafür muss der Reverse-Proxy das Protokoll teilweise verstehen (Protokollinspektion notwendig)

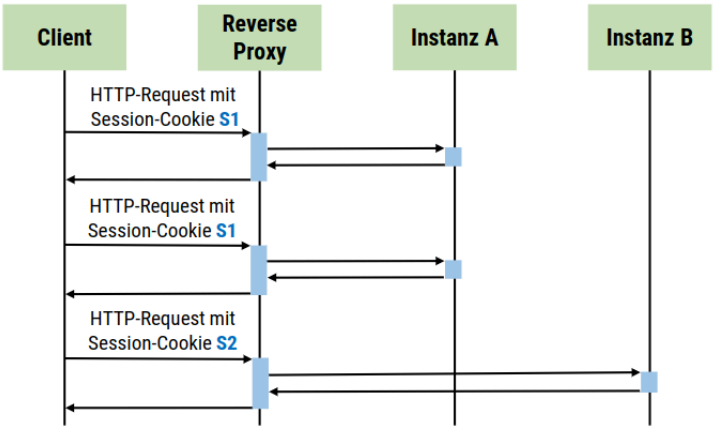


Abbildung 54: Session-Persistenz

18.2.4. Ausfallsicherheit Serverinstanz

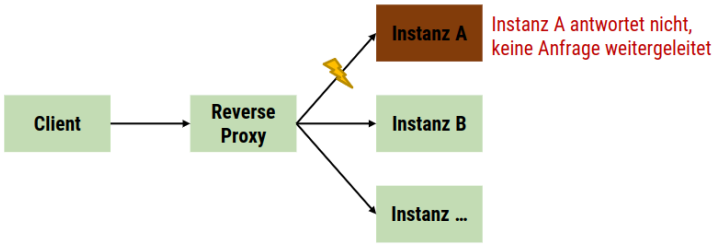


Abbildung 55: Instanz-Ausfall

- Load-Balancer führen Health-Checks durch
  - Anfragen werden nicht an nicht antwortende Instanzen weitergeleitet

18.2.4.1. Typische Health-Checks

- TCP-Verbindung vorhanden (Transport-Schicht)
- Beliebiger HTTP-Request (Applikations-Schicht)
- Agent (Hilfsprogramm) auf Serverinstanz ausführen

18.2.5. Ausfallsicherheit Proxy

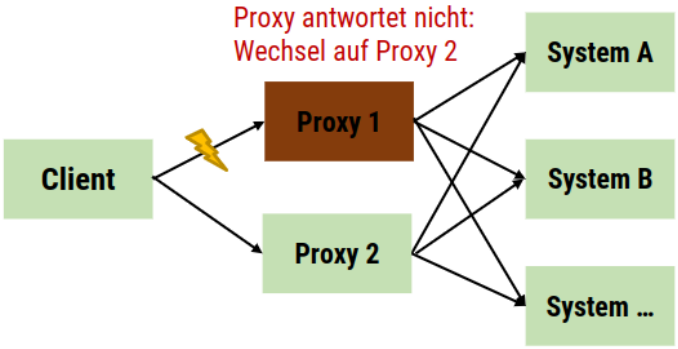


Abbildung 56: Proxy-Failover

- Einzelner Reverse-Proxy = Single-Point of Failure
- Mindestens zwei Proxy-Instanzen notwendig

18.2.5.1. Massnahmen

- Hochverfügbarkeitslösungen (z. B. keepalived)
- Verwendung von Floating-IPs (IP kann auf andere Instanz umgeleitet werden)
- Bei Proxy-Ausfall: IP-Wechsel zur anderen Instanz

18.2.6. Beispiel HAProxy

- Download: haproxy.org
- Open-Source und kommerzielle Version
- Architektur: Asynchrone IOs + mehrere Threads
- Kommerzielle Erweiterungen: z. B. GeoIP-Routing, Traffic-Mirroring

18.2.6.1. Einfacher Aufbau

- Einzelnes Binary: `haproxy`
- Konfiguration z. B. in `haproxy.conf`

18.2.6.2. Start des Proxys

```
haproxy -V -f haproxy.conf
```

18.2.6.3. Konfiguration

```conf
global

defaults
  mode tcp                # Protokoll (z. B. tcp oder http)
  balance roundrobin      # Lastverteilung (z. B. roundrobin, leastconn, source)
  timeout connect 5000ms  # Verbindungsaufbau
  timeout client 50000ms  # Client-Inaktivität
  timeout server 50000ms  # Server-Inaktivität

frontend http-in
  bind *:4000             # Port und Interface
  default_backend servers # Ziel-Backend

backend servers
  server server1 127.0.0.1:8000 check
  server server2 127.0.0.1:8001 check
```

- **Protokoll**: `tcp`, `http`, …
- **Lastverteilung**: `roundrobin`, `leastconn`, `source`, …
- **Timeouts**: Verbindung und Inaktivität
- **Frontend**: Eingangspunkt für Clients
- **Backend**: Liste der Zielinstanzen mit Health-Checks

18.2.6.4. Übung

- Ein Server gestoppt
  - Anfrage geht an anderen Server
- Beide Server gestoppt
  - Direkte Anfrage liefert Fehler
  - Anfrage an Proxy liefert keinen Fehler, da Proxy trotzdem erreichbar, kann aber am Server nichts senden

18.3. In Memory Datagrids

- Applikationen skalieren
- Daten über Cluster verteilen
- Daten partitionieren
- Nachrichten senden und empfangen
- Lasten verteilen
- Parallele Tasks verarbeiten

18.3.1. Technologien

- HazelCast: hazelcast.com
- Apache Ignite: ignite.apache.org
- Oracle Coherence: oreil.ly/XOUJL

18.3.2. Architektur

- **Verarbeitungseinheiten**: Halten Teile der Daten im Hauptspeicher bereit
- **Datenbank**: Speichert alle Daten dauerhaft
- **Middleware**: Kommunikation zwischen Verarbeitungseinheiten

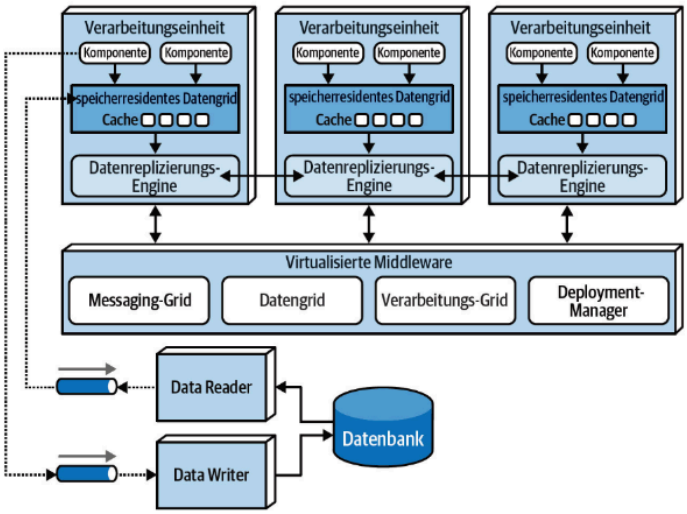


Abbildung 57: Data Grid Architektur

18.3.2.1. Verarbeitungseinheiten

- **Embedded-Topologie**: Logik und Daten liegen zusammen auf dem Node
- **RAM-basierte Datenspeicherung**:
  - Schneller als festplattenoptimierte Datenbanken
  - Verbesserte Reaktionszeiten bei kritischen Anwendungen

18.3.2.2. 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

18.3.2.3. Datenpartitionierung

- Feste Anzahl an Partitionen
- Partitionen werden durch Hash-Funktion zugewiesen:
  - `partitionId = hash(keyData) % PARTITION_COUNT`
- **Ziel**: Gleichverteilung aller Partitionen auf die Cluster-Nodes
- Jede Partition besitzt Backups → Redundanz

Beispiel

- 12 aktive Partitionen
- 12 Backup-Partitionen (zur Ausfallsicherheit)
- Beim **Hinzufügen eines neuen Knotens** (z. B. Knoten D):
  - Bestehende Partitionen und Backups werden **migriert**
  - Ziel: Last gleichmässig verteilen und Redundanz erhalten
- Beim **Ausfall eines Knotens**:
  - Backup-Partitionen übernehmen Rolle der Primärpartitionen
  - Andere Knoten erstellen neue Backups (Selbstheilung)
- Ergebnis: **12 Partitionen + 12 Backups** stets im Cluster verteilt

18.3.3. HazelCast

- Parallel, verteilt und **thread-safe**
- **Mehrere Instanzen** pro JVM möglich
- Verteilte Objekte müssen **DataSerializable** implementieren

18.3.3.1. Distributed Map

- **Idee**: Transparent verteilt statt lokale Maps

```java
import com.hazelcast.core.Hazelcast;
import com.hazelcast.core.HazelcastInstance;
import com.hazelcast.nio.serialization.DataSerializable;
import java.io.IOException;
import java.io.ObjectInput;
import java.io.ObjectOutput;
import java.util.Map;

public class HazelcastExample {
    public static void main(String[] args) {
        HazelcastInstance client = Hazelcast.newHazelcastInstance();
        Map<String, Employee> map = client.getMap("employees");
        map.put("e1", new Employee("Alice", 30));
        Employee result = map.get("e1");
        System.out.println("Result: " + result);
    }

    public static class Employee implements DataSerializable {
        private String name;
        private int age;

        public Employee() {}

        public Employee(String name, int age) {
            this.name = name;
            this.age = age;
        }

        @Override
        public void writeData(ObjectOutput out) throws IOException {
            out.writeUTF(name);
            out.writeInt(age);
        }

        @Override
        public void readData(ObjectInput in) throws IOException {
            name = in.readUTF();
            age = in.readInt();
        }
    }
}
```

18.3.3.2. Distributed Lock

- Verteilter Lock für sicheren Zugriff
- Nur ein Client zur Zeit erlaubt
- Lock wird am Ende freigegeben

```java
String mylockobject = "MyLock";
FencedLock mylock = client.getCPSubsystem().getLock(mylockobject);
mylock.lock();
try {
    // do something
} finally {
    mylock.unlock();
}
```

18.3.3.3. Distributed Concurrent Map

- Verteilte ConcurrentMap zur parallelen Datennutzung

- Speicherung und Zugriff auf `Customer`-Objekte
- Methoden wie `putIfAbsent`, `replace`, `get` unterstützt
- Ideal für synchronisierten Zugriff im Cluster

```java
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);

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);
```

18.3.3.4. Desitributed Queue

- Verteilte Queue zur Task-Verarbeitung im Cluster
- Unterstützt normale, zeitlich begrenzte und blockierende Operationen
- Gut geeignet für Producer/Consumer-Modelle

```java
HazelcastInstance client = Hazelcast.newHazelcastInstance();
BlockingQueue<Task> queue = client.getQueue("tasks");

// Queue operations
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);
```

18.3.3.5. Distributed Topic

- Verteiltes Topic für Publish/Subscribe-Kommunikation
- Nachrichten werden an alle registrierten Listener gesendet
- Ideal für Broadcasts an mehrere Empfänger

```java
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"));
```

18.4. Zusammenfassung

TL;DR:

- **Topologie**:
  - Client/Server = getrennte Verteilung
    - Embedded = Daten und Logik gemeinsam
- Lastverteilung ermöglicht Skalierung verteilter Systeme
- Temporärer/persistenter Zustand erschwert Skalierung
- **Reverse-Proxy**:
  - Middleware zur Lastverteilung mit verschiedenen Methoden
- **In-Memory Data Grids**:
  - Speicherung in Partitionen im Hauptspeicher
  - Embedded-Topologie
    - Gleichmässige Verteilung über Nodes
  - Redundante Speicherung zur Ausfallsicherheit
- Einsatz zur Kommunikation und verteilten Datenspeicherung

19. Continuous Integration (CI) ♻️

19.1. Hauptziele

- **Lauffähiges Produkt** jederzeit verfügbar
  - Kontinuierlich testbar
- **Schnelles Feedback bei Fehlern**
  - **Automatisierte** Unit-/Integrationstests
  - Compiler, Classpath, statische Codeprüfung
- **Paralleles Arbeiten im Team**
  - Überblick behalten
  - Geringer Integrationsaufwand (small steps)
  - Aktueller Stand stets bekannt
- **Moderne, zeitgemässe, agile Software-Entwicklung**
- **Grundidee**: Kent Beck (XP, JUnit), Martin Fowler (Refactoring)

19.2. 10 Praktiken der CI

1. Zentrales Versionskontrollsystem
2. Automatisierter Buildprozess
3. Automatisierte Testfälle
4. Arbeiten am Hauptzweig
5. Automatischer Build bei Änderungen
6. Schneller Buildprozess
7. Tests mit produktionsnaher Umgebung
8. Einfacher Zugriff auf Buildartefakte
9. Transparenz über aktuellen Zustand
10. Automatisches Deployment

19.2.1. 1. Versionskontrollsystem

- **Zentrales VCS** wie Git verwenden
- Alle build-relevanten Artefakte **versionieren**
- **Gute Commit-Kommentare**, Tags für Releases
- **Feature-Branches** für parallele Entwicklung

19.2.2. 2. Automatisierter Buildprozess

- **Build läuft automatisch**, reproduzierbar, auf sauberem System
- Nur aus VCS-Quellen, **keine lokalen Abhängigkeiten**
- Führt auch **Tests** und **Codeanalysen** aus
  - Laufen überall
  - Keine side effects
  - Gute Codequalität

19.2.3. 3. Automatisierte Testfälle

- **Unit Tests** für Logik, **Integrationstests** für Abhängigkeiten (z.B. DB)
- **Fehler früh erkennen**, auch bei Seiteneffekten
- **Performance-Tests** je nach Bedarf
- **Tests müssen stabil laufen** und schnell gefixt werden

19.2.4. 4. Arbeiten am Hauptzweig

- Änderungen **früh** in den Hauptzweig **integrieren**
- **Kurze, gezielte Branches** statt langem Abseitsentwickeln
- **GitFlow oder GitHub-Flow** als Modelle für Teams
- **Ziel**: häufige, einfache Integration

19.2.5. 5. Automatischer Build bei Änderungen

- Buildserver erkennt Änderungen und startet **automatisch Build**

- **Push oder Polling** durch das VCS
- **Ergebnisse** (Tests, Metriken, Status) werden **sofort angezeigt**
- **Buildfehler sofort** und gemeinsam **beheben**

19.2.6. 6. Schneller Buildprozess

- **Kurze Feedbackzeit** für Entwickler wichtig
- **Aufteilung** in schnellen und langsamen **Build** (z. B. nightly)
- Build-Pipelines mit **Fail-fast-Strategie**
- Nur **relevante Tests zuerst** ausführen

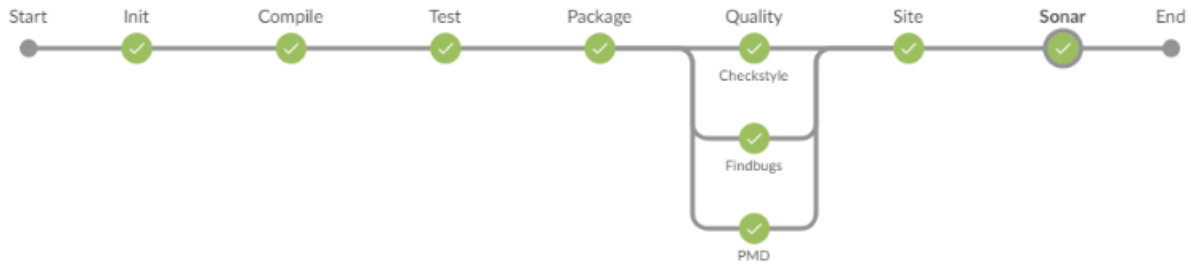


Abbildung 58: Buildprozess

19.2.7. 7. Tests auf produktionsnaher Umgebung

- Testumgebung möglichst **ähnlich wie Produktion**
  - Gleiches OS, Laufzeit, Netzwerk, Datenmenge
- **Nutzung von Containern** (z. B. Docker) empfohlen
- **Datenschutz und Kosten** im Blick behalten

19.2.8. 8. Zugriff auf Buildartefakte

- Build-Ergebnisse **jederzeit abrufbar** (z. B. über Jenkins)
- **Speicherung in Artefakt-Repositories** wie Nexus, Artifactory
  - Ermöglicht schnelle Weiterverwendung (z. B. für Tests)

19.2.9. 9. Offene Information

- Änderungen, Buildstatus, Verantwortliche sind einsehbar
- **Transparenz** stärkt Teamarbeit und Verantwortung
- Grundlage für **gemeinsame Codeverantwortung**
- Kein Kontrollinstrument, sondern **Unterstützung**

19.2.10. 10. Automatisches Deployment

- Neue Builds **automatisch installiert** oder bereitgestellt
- **Regelmässige Aktualisierung**, z. B. täglich
- Unterstützt **schnelle Rückmeldung** durch Tester
- **DevOps-Prinzipien** wie Container, IaC und Cloud **hilfreich**

20. Sicherheit in Verteilten Systemen

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.

20.1. Fokus: sichere Kommunikation zwischen verteilten System

A und B sollen sicher kommunizieren (Daten über Drittsystem, lange Verbindung oder drahtlos).

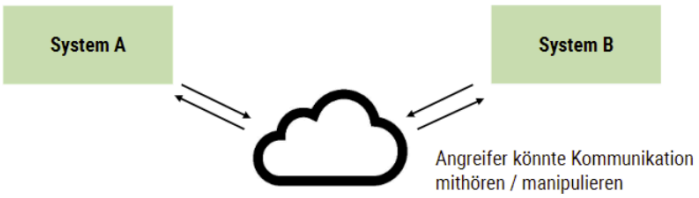


Abbildung 59: Kommunikation zwischen A und B

Sichere Kommunikation beinhaltet:

- A muss sicherstellen, dass B das korrekte System ist.
- B muss sicherstellen, dass A ein berechtigtes System ist.
- Kein Drittsystem soll die Kommunikation mithören.
- Kein Drittsystem soll die Kommunikation manipulieren.

–> gilt für gesamte Dauer der Kommunikation (Sitzung)

20.2. Cyber-Security - Massnahmen

Ziel	Beschreibung	Massnahme
Zugriffschutz	nur berechtigte Benutzer und System können zugreifen	Authentifizierung und Autorisierung
Manipulationssicherheit	Daten können nicht unerlaubt manipuliert werden	Signatur
Abhörsicherheit	keine geschützte Informationen gelangen nach aussen	Verschlüsselung
Nachvollziehbarkeit	Wer wann was am System gemacht hat	Logs / Audit-Trails

20.3. Welche Informationen verbergen?

Tatsache, dass Kommunikation stattfindet, lässt sich nur schwer verbergen (Achtung: Seitenkanäle).

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

20.4. Symmetrische Verschlüsselung

- gleicher Schlüssel zum Ver-und Entschlüssel
- effizienter als asymmetrische Verschlüsselung
- Einsatz: Verschlüsselung von Datenströmen

20.5. Asymmetrische Verschlüsselung

Erzeugung eines Schlüsselpaars:

- privater Schlüssel
- öffentlicher Schlüssel

Einsatz zum Verschlüsseln (B sendet an A eine Nachricht):

- B verschlüsselt mit öffentlichem Schlüssel von A
- nur A kann mit privatem Schlüssel

Signatur und Zertifizierung:
- Signatur: A verschlüsselt Hash einer Information mit privatem Schlüssel
- B entschlüsselt Signatur mit öffentlichen Schlüssel
- entspricht Entschlüsselung dem Hashwert, ist die Signatur erstellt

20.6. Transportlayer Security (TLS)
- Standard Protokoll für verschlüsselte Übertragung
- transparente Verschlüsselung der **Anwendungskommunikation**
- Konzept einer Transportschicht (z.B. TCP)
- implementiert als Protokoll in Anwendungsschicht

Verortung von TLS innerhalb des Internetschichtenmodells:

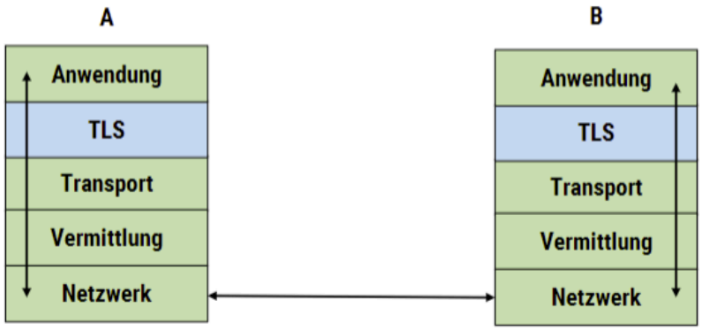


Abbildung 60: Verortung von TLS innerhalb des Internetschichtemodells

20.6.1. 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	Aktuell. Wenn möglich diese Version verwenden. (RFC 8446)

Abbildung 61: TLS Versionen

20.6.2. Verbindungsaufbau (TLS 1.3 Handshake)

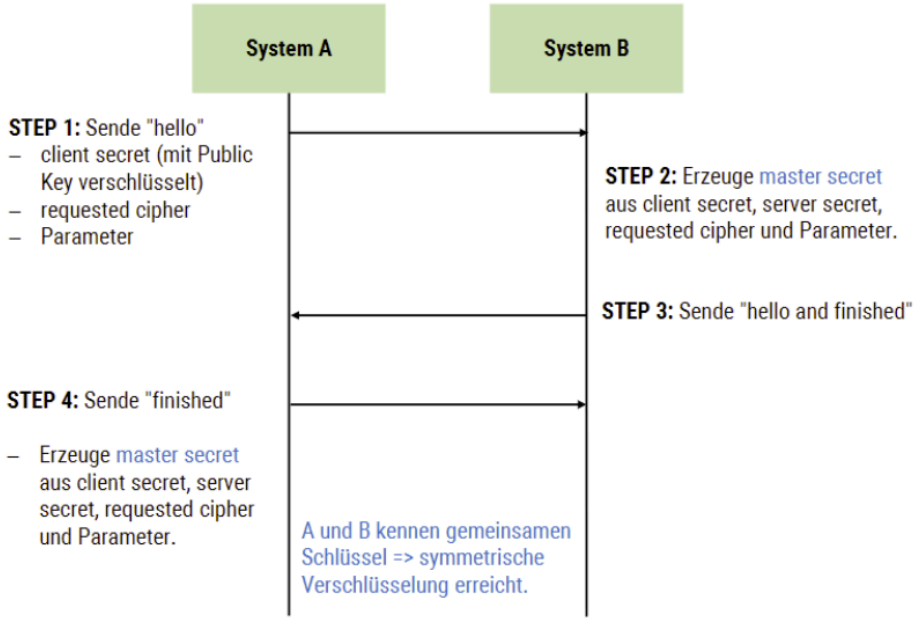


Abbildung 62: Handshake, Verbindungsaufbau

Falls der Private-Key geleakt wird, kann man nur die letzte Session sehen (nicht alles)

20.6.3. TLS gekoppelt mit Zertifikatsprüfung

- TLS verschlüsselt in Kombination mit Authentifizierung der Gegenstelle -> **X.509 Zertifikat**
- Verhinderung von „Man-in-the-Middle“ Attacken (C gibt sich gegenüber A als B aus und gegenüber B als A)

20.6.3.1. 509 Zertifikat

- binden eine Identität anhand digitaler Signatur zu dem öffentlichen Schlüssel
- Zertifikat ist zertifiziert durch Zertifizierungsstelle (oder selbstzertifiziert)
- Kategorien
  - Zertifizierungsstelle (CA): kann weitere Zertifikate erteilen, mehrere Hierarchiestufen. Benötigt Vertrauen in oberste Hierarchie (Root-CA).
  - Endstelle: identifiziert eine Entität (Domain, Person, Organisation, etc.)

20.6.3.2. Chain of Trust

- Root-CA stellt keine Endstellen-Zertifikate aus (nur Entwicklungs-/Testszenarios) –> Sicherheitsgründe
- Mehrere Schichten von Zwischen-Zertifizierungsstellen (Intermediate-CA) erhöhen Sicherheit.
- Falls eine Zertifizierungsstelle Z kompromittiert wurde (Zugriff auf Private- Key war möglich) werden alle von Z ausgestellten Zertifikate ungültig.
- Server liefert immer sein Zertifikat und alle Zertifikate der Zwischenstelle aus (von Blatt Richtung Root-CA, aber ohne Root-CA)

20.6.3.2.1. Beispiel: Client C greift mittels TLS auf Endstelle 4 zu

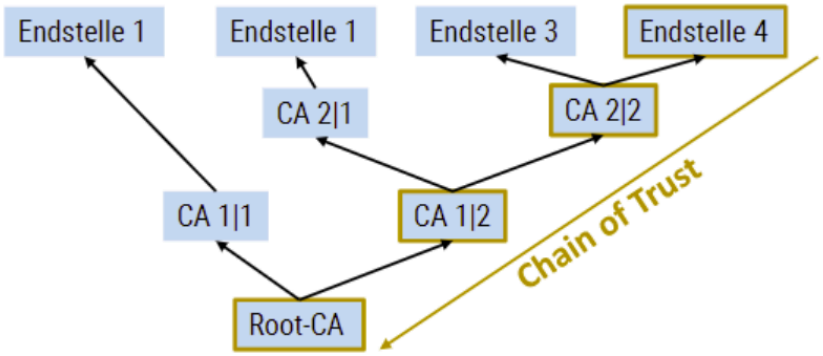


Abbildung 63: Beispiel

- Endstelle 4 liefert in folgender Reihenfolge alle Zertifikate zu Client C: Endstelle 4 -> CA 2|2 -> CA 1|2.
- Client C muss selbst über Root-CA verfügen.

20.6.3.3. Erstellung eines Certificate Sign Request (CSR)

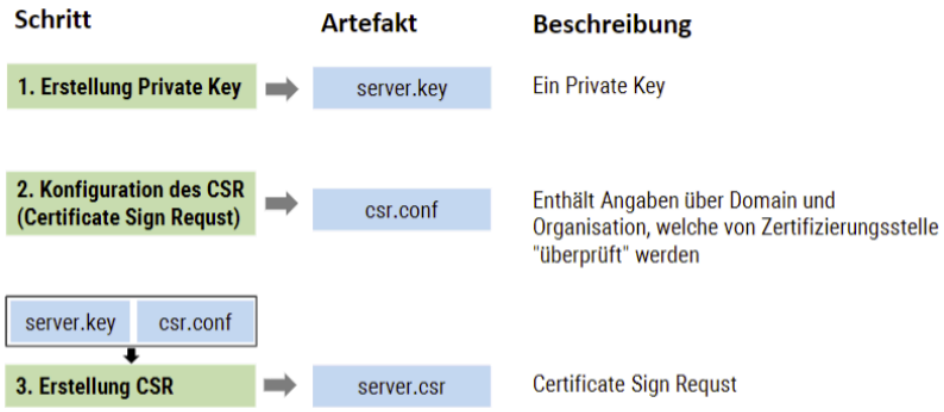


Abbildung 64: Erstellung eines Certifcate Sign Request (CSR)

20.6.3.4. Zertifikatsausstellung durch Zertifizierungsstelle

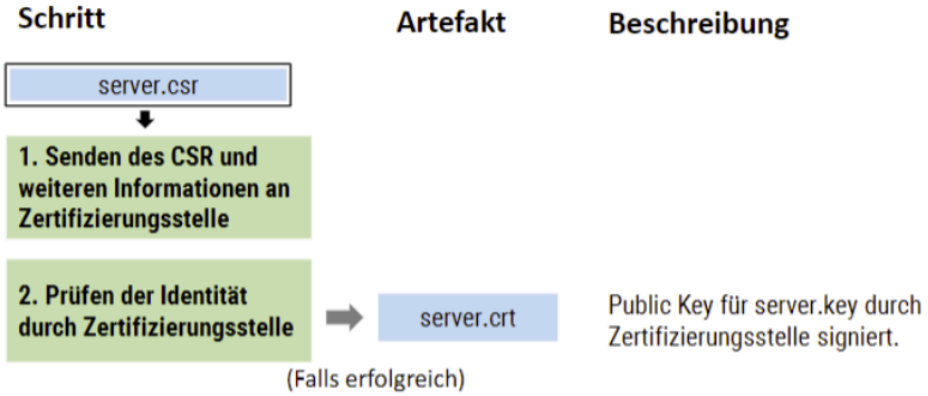


Abbildung 65: Zertifikatsausstellung durch Zertifizierungsstelle

20.6.3.5. Zertifikatsausstellung (Selbstzertifiziert)

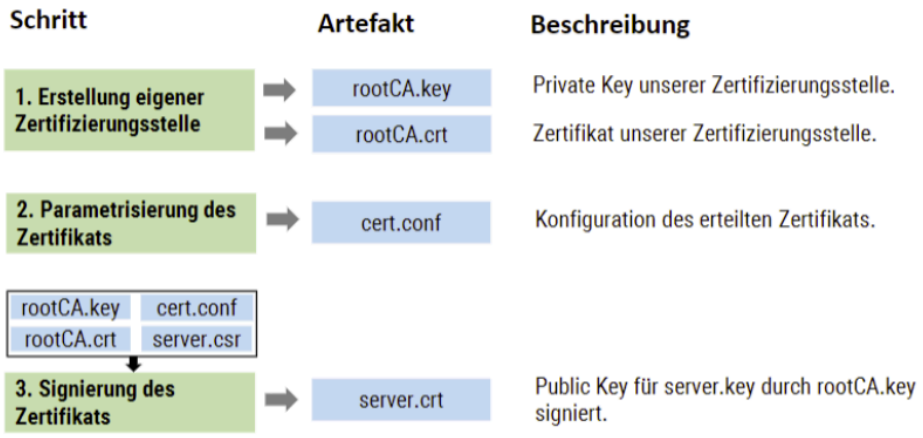


Abbildung 66: Zertifikatsausstellung (Selbstzertifiziert)

20.6.3.6. Aufbau einer sicheren Verbindung mit Java (Keystore)

Java erwartet Zertifikate und Schlüssel in einem Keystore (Java spezifisch)

- jks ist ein File

```bash
# Schritt 1: Erstelle ein PKCS#12-Bundle (.p12-Datei)
# Dieses Bundle enthält den privaten Server-Schlüssel, das Server-Zertifikat und das
Root-CA-Zertifikat
openssl pkcs12 -export \
  -in server.crt \          # Eingabe: Server-Zertifikat
  -inkey server.key \       # Eingabe: Privater Schlüssel des Servers
  -chain \                  # Inklusive Zertifikatskette (z.B. CA-Zertifikat)
  -CAfile rootCA.crt \      # CA-Zertifikat, das den Server beglaubigt
  -name localhost \         # Alias/Name für die Einträge im Bundle
  -out server.p12           # Ausgabe: PKCS#12-Datei (enthält Zertifikat + Key + CA)

# Schritt 2: Importiere das PKCS#12-Bundle in einen Java Keystore (.jks)
# Damit kann eine Java-Anwendung das Zertifikat und den Schlüssel nutzen
keytool -importkeystore \
  -deststorepass myServerPass \   # Passwort für den Ziel-Keystore (server.jks)
  -destkeystore server.jks \      # Ziel: Java Keystore-Datei (enthält Schlüssel und
Zertifikate)
  -srckeystore server.p12 \       # Quelle: PKCS#12-Datei, die wir eben erzeugt haben
  -srcstoretype PKCS12            # Typ der Quelle: PKCS#12-Format

# Schritt 3: Importiere das Root-CA-Zertifikat separat in einen Truststore
# Damit vertraut die Java-Anwendung Zertifikaten, die von dieser Root-CA ausgestellt
wurden
keytool -import -v \
  -trustcacerts \                 # Gibt an, dass das Zertifikat als vertrauenswürdig
behandelt wird
  -alias server-alias \            # Alias für das CA-Zertifikat im Keystore
  -file rootCA.crt \               # Eingabedatei: Unser Root-CA-Zertifikat
  -keystore cacerts.jks \          # Ziel-Keystore: Neuer Truststore für CA-Zertifikate
  -keypass myCaCertsPass \         # Passwort für den privaten Schlüssel (bei Truststores
optional)
  -storepass myCaCertsPass         # Passwort für den Keystore (Truststore)
```

20.6.4. Aufbau einer sicheren Verbindung mit Java (Client)

1	**Erzeuge eine `SSLSocketFactory`**	`SSLSocketFactory.getDefault()`

2	**Erstelle ein sicheres Socket**	`factory.createSocket(HOST, PORT)`
3	**Definiere erlaubte Protokolle und Cipher Suites**	`setEnabledProtocols`, `setEnabledCipherSuites`

```java
// Definiere den Hostnamen oder die IP-Adresse des Servers
private static final String HOST = ...;

public static void main(final String[] args) {

    // Schritt 1: Erstellung einer SSLSocketFactory
    // Eine SSLSocketFactory wird verwendet, um SSL-Sockets zu erstellen,
    // die sichere Verbindungen (TLS/SSL) unterstützen.
    SSLSocketFactory factory = (SSLSocketFactory) SSLSocketFactory.getDefault();

    // Schritt 2: Erstellung eines SSLSockets
    // Mit der SSLSocketFactory wird nun eine gesicherte Verbindung zu einem Server
aufgebaut.
    // HOST: Adresse des Servers
    // 1234: Port des Servers
    try (SSLSocket socket = (SSLSocket) factory.createSocket(HOST, 1234)) {

        // Schritt 3: Setzen unterstützter Protokoll-Versionen und Cipher Suites
        // Hier wird festgelegt, welche TLS-Protokollversion und welche
Verschlüsselungsverfahren
        // für die Verbindung erlaubt sein sollen.

        // Nur TLS 1.3 zulassen
        socket.setEnabledProtocols(new String[] {"TLSv1.3"});

        // Nur die Cipher Suite "TLS_AES_128_GCM_SHA256" erlauben
        socket.setEnabledCipherSuites(new String[] {"TLS_AES_128_GCM_SHA256"});

        // Ab hier kann das SSLSocket wie ein normales TCP-Socket genutzt werden:
        // Schreiben/Lesen von Daten, Streams öffnen usw.
        //
        // Beispiel (nicht im Bild): OutputStream out = socket.getOutputStream();
        //                           InputStream in = socket.getInputStream();

    } catch (IOException e) {
        // Fehlerbehandlung: Verbindungsfehler, TLS-Handshakes, etc.
        e.printStackTrace();
    }
}
```

20.6.4.1. Beispiel: Starten von Client und Server mit TLS

Server: Keystore (Name ist beliebig, hier server.jks) hier mit Private Key und Zertifikat (immer):

```bash
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 selbsterstellten Zertifikaten):

```bash
java -Djavax.net.ssl.trustStore=cacerts.jks \
  -Djavax.net.ssl.trustStorePassword=myCaCertsPass \
  myClientClassName
```

Debug-Modus: Falls SSL-Modus nicht funktioniert:

```bash
-Djavax.net.debug=ssl
```

20.7. Sessions

Definition:

- zeitlich beschränkte Zweiweg-Kommunikation (Anmelden bis Abmelden)
- Typisch zwischen Client und Server (Request-Response)

20.7.1. Session-Secret

20.7.1.1. Entkoppelung der Session von TCP-Connection

Session-Secret ist nur dem Server und dem Client bekannt

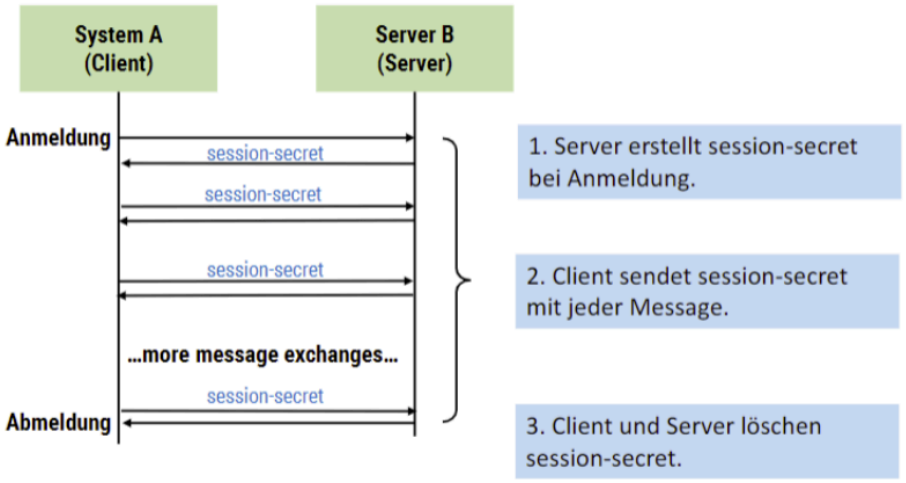


Abbildung 67: Session-Secret

20.7.1.2. Varianten von Session-Secrets

Zufällige Zahl ohne Informationsgehalt:

- ist am billigsten
- 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

20.7.1.2.1. Beispiel: Berechnung eines Session-Secrets (Zufallszahl)

- Java: `SecureRandom`

- Unter Linux bezieht `SecureRandom` per Default seine Zahlen von /dev/random (blockiert, falls nicht genügend Entropie).

```java
private byte[] generateSessionKey() {
  SecureRandom secureRandom = new SecureRandom();
  byte[] sessionKey = new byte[16];
  secureRandom.nextBytes(sessionKey); // may block
  return sessionKey; }
```

20.8. Authentifizierung und Autorisierung

20.8.1. Terminologie

Authentisieren: eine Partei P (user/System) weisst sich gegenüber einem System S mittels eines Geheimnisses aus –> anhand Kenntnis einer Information, einem Zugang oder einem Besitz, welche nur Partei P hat (Passwort, AccessToken, Smartcard, Empfang einer Mail, etc.)

Authentifizieren: System S prüft ob Partei P die ist, die es vorgibt zu sein. Geheimnis wird von S überprüft.

Autorisierung: Prüfung, ob eine Partei berechtig ist, auf eine Ressource zuzugreifen.

20.8.2. Authentifizierung einer Session (mit Passwort)

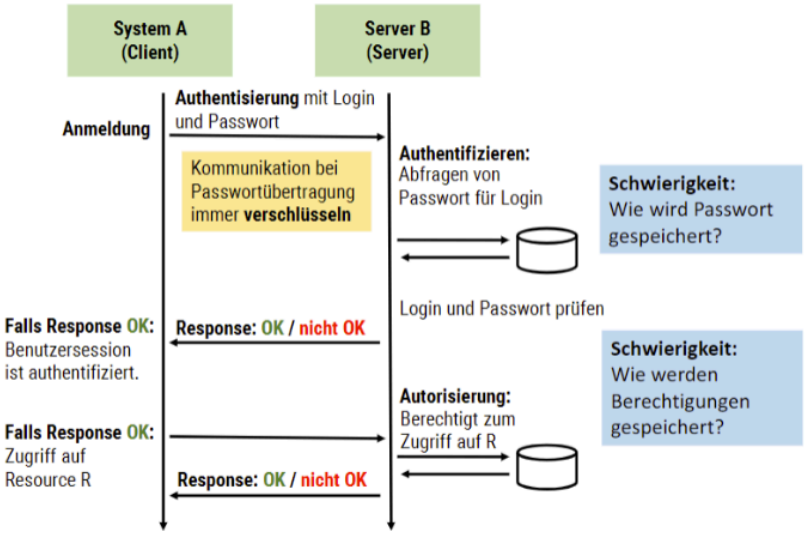


Abbildung 68: Authentifzierung einer Session (mit Passwort)

- eine Session, die für eine Partei P steht wird authentifiziert
- nach erfolgreicher Authentifizierung wird eine Session mit dem Konto der Partei P verknüpft.
- wesentlicher Teil des Logins ist die Passwortprüfung

```java
void login(Message message) {
  String account = message.getAccount();
  String password = message.getPassword();
  PasswordRecord record = PasswordManager.getAccount(account);

  if (record.matches(password)) { // Überprüfung des Passworts
    session.setAccount(account); // Verknüpfung der Benutzersession
  }
}
```

20.8.2.1. Passwortprüfung

- Passwörter als Hash speichern

- vorgefertigte Funktionen und Libraries verwenden

geeignete Hashfunktionen
- keine generischen Hash-Funktionen verwenden
  - sind auf Geschwindigkeit optimiert, mit beschränkter Lebendsdauer
- Passwort-Hashfunktionen verwenden: PBKDF2, Bcrypt, Scrypt
  - langsamer in Berechnung, was ein Vorteil ist
  - PBKDF2 in Kombination mit HMAC-SHA256 empfohlen von NIST (https://pages.nist.gov/800-63-3/sp800-63b.html#memsecretver)

weitere Massnahmen
- **Cost**: anpassen der Kosten der Hashwerberechnung an aktuelle Hardware
  - Kosten für Login ca. 100ms (schnell genug, teuer für Angriff)
- **Salt**: pro Passwort eine zufällige Zahl, die mit dem PW gehasht wird, verhindert Brute-Force- oder Wörterbuchattacken
- **Pepper**: zusätzlich alle Passwörter mit weiterer (für alle identisch) Zahl hashen. Diese Zahl separat von Passwort-DB speichern.

Beispiel: Generierung von Salt und Hash in Java:

```java
// Erzeugt ein zufälliges Salt zur Absicherung des Passwort-Hashings
private byte[] generateSalt() {
    SecureRandom random = new SecureRandom(); // Kryptografisch sicherer
Zufallszahlengenerator
    byte[] salt = new byte[16]; // Erzeugt ein 16-Byte (128 Bit) langes Salt
    random.nextBytes(salt); // Füllt das Salt-Array mit zufälligen Bytes
    return salt; // Gibt das generierte Salt zurück
}

// Legt die Anzahl der Iterationen für das Hashing fest (je höher, desto sicherer, aber
langsamer)
private static final int ITERATION_COUNT = 65536;

// Definiert die Schlüssellänge und die zu verwendende Hashfunktion (PBKDF2 mit HMAC-
SHA512)
private static final int KEY_LENGTH = 512; // Schlüssellänge in Bits
private static final String CRYPTO = "PBKDF2WithHmacSHA512"; // Hash-Algorithmus

// Erzeugt einen Hash aus Passwort und Salt unter Verwendung von PBKDF2
public byte[] generateHash(String password, byte[] salt) {
    // Definiert die Spezifikation für das Schlüsselderivat: Passwort, Salt,
Iterationszahl und Schlüssellänge
    KeySpec spec = new PBEKeySpec(
        password.toCharArray(), // Umwandlung des Passworts in ein char-Array
        salt,                   // Das verwendete Salt
        ITERATION_COUNT,         // Anzahl der Iterationen
        KEY_LENGTH               // Länge des erzeugten Schlüssels
    );

    try {
        // Initialisiert eine SecretKeyFactory mit dem angegebenen Algorithmus
        SecretKeyFactory factory = SecretKeyFactory.getInstance(CRYPTO);
        // Erzeugt das Hash-Bytearray basierend auf den Spezifikationen
        return factory.generateSecret(spec).getEncoded();
    } catch (Exception e) {
        // Fehlerbehandlung: sollte nicht auftreten, wenn Algorithmusname korrekt ist
        throw new RuntimeException("Fehler beim Erzeugen des Password-Hashes", e);
    }
}
```

Beispiel: Erstellung eines Passwortrecords / Passwordprüfung:

```java
// Record enthält mind. Hash und Salt. Ideal wäre noch Parameter der Hashfunktion zu
speichern
public static class PasswordRecord {
  private final byte[] hash;
  private final byte[] salt;

  public PasswordRecord(byte[] hash, byte[] salt) {
    this.hash = hash;
    this.salt = salt;
  }

public PasswordRecord create(String password) {
  byte[] salt = generateSalt(); //salt erstellen
  byte[] hash = generateHash(password, salt); //hash mit pw und salt erstellen
  return new PasswordRecord(hash, salt); //record mit hash und salt zurückgeben
}

//Erstellung eines Passwordhash mit eingegebenem pw und salt aus dem record.
anschliessend Vergleich
public boolean compare(String password, PasswordRecord record) {
  byte[] newPasswordHash = generateHash(password, record.salt);
  return Arrays.equals(newPasswordHash, record.hash);
}
```

20.8.2.2. Autorisierung

Nach Verknüpfung mit Konto ist eine Session autorisiert auf bestimmte Ressourcen oder Funktionalitäten zuzugreifen.

```java
if (session.hasRole("logger")) {
  while (true) {
    LogMessage msg = receiveLogMsg();
    ...
  }
}
```

Typische Verfahren zur Zugriffskontrolle:

- **Role Based Access Control**: Definition von Rollen und prüfen, ob Benutzersession für die benötigte Rolle Rechte hat
- **Row Level Security**: Funktionalität von Datenbanken: Definiert pro Ressource (oft in Table-Row gespeichert), wer darauf zugreifen darf.

21. Deployment 🚀

21.1. Deployment Grundlagen

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, prüfen, ob das Zielsystem für die Anwendung geeignet ist (Hardwareausstattung, Betriebssystemversion etc.).

Konfiguration: Einstellungen der Anwendung auf User, Netzwerkumgebung, Hardware etc.

Organisation: Planung, Produktion, Information (Marketing), Schulung (intern), Support bereitstellen

21.1.1. Wann findet Deployment statt?

- aufgrund iterativer und agiler Entwicklungsmodellen soll das Deployment kontinuierlich und früh stattfinden (CI/CD erlaubt auch Continuous Deployment)
- einzelne Build-/Sprint-/Iterationsergebnisse fortlaufend deployen, z.B. auf interne Testumgebung oder direkt beim Kunden (alpha, beta, release candidate, etc.)
- **Continuous Delivery** -> Grundlage für Continuous Deployment (Analogie zum Testen, vgl. Continuous Integration), Nutzung von DevOps Technologien, Infrastructure as Code
- Staging: Deployment auf unterschiedliche Umgebungen (Entwicklung, Test, Integration, Vor-Produktion, Produktion

21.1.2. 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!

21.2. Deployment - Aspekte

21.2.1. Installation und Deinstallation

Ziel:

- Installation und Update einer Software möglichst automatisiert
- saubere Deinstallation
- vollautomatisierte Verteilung (Software-Management)

Bedürfnisse:

- End-User: grafische, interaktive GUI-Installation für Endanwender auf dem Desktop.
- Admin: möglichst script-basiert, durch Parametrisierung voll automatisierte Installation auf dem Server für Admins
- Entwickler / Tester: spezielle Distributionen, entweder manuell (download) oder über (zentrale) Repositories

21.2.2. Konfiguration von Anwendungen

Die neu installierte Anwendung soll:

1. möglichst sofort „out-of-the-box“ lauffähig sein,…
2. sich an verschiedene Umsysteme anpassen können, und…

3. trotzdem möglichst einfach aktualisierbar sein.

-> Zielkonflikt

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.).

21.2.3. Konfigurationsmanagement

Herausforderungen
- verschiedene Kunden haben unterschiedliche Versionen
- verschiedene Kunden haben unterschiedliche Produkte und Versionen der Umsysteme (z.B. Datenbank) und Hardware (Einfluss auf Performance)
- Update von jeder existierenden Konfiguration

21.2.4. Deploymentdokumentation / Manuals

Release Notes: erste Quelle über
- neue Funktionen
- veränderte / zusätzliche Vorbedingungen
- neue / veränderte Datenformate oder Protokolle
- etc.

Installationsanleitung:
- Veränderung an HW- oder SW- Voraussetzungen
- Varianten von unterschiedlichen Konfigurationen berücksichtigen
- Abfolgen, falls eingehalten werden muss
- Tipp: möglichst automatisierte Installation -> wenig Doku

Bedienungsanleitung / User-Manual

21.3. Releases und Versionierung

Ziel: 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

Wichtig:
- eindeutige Bezeichnung und Version
- technische Version ist die eindeutige Identifikation
- Tagging im Versionskontrollsystem
- Marketing-Version unbedingt trennen (ist eine Ergänzung)

Bewährte Versionierung: Dreistellige Version x.y.z (z.B. 7.2.3) mit Semantik -> Semantic Versioning http://semver.org/

21.3.1. Semantic Versioning - Repetition

- **Major-Release (X.x.x):**
  - Veränderungen in der API, in der fachlichen Funktion oder in der Konfiguration
  - zu früheren Versionen nicht mehr kompatibel
  - machen Anpassungen notwendig
- **Minor-Release (x.X.x):**
  - Erweiterungen in der API, der fachlichen Funktion oder der Konfiguration,
  - sind vollständig Rückwärtskompatibel
  - 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

21.3.2. Time-Based Release Versioning

- festen Takt an Release Terminen
- (sinnvollerweise) Versionsnummer macht Zeitpunkt deutlich
- Einsatz bei grossen Produkten oder / und Marketing-Versionen sinnvoll, aber nicht bei Libraries / Komponenten
- Empfehlung: Versionierung mit Vernunft

21.3.2.1. bei Oracle / Java

–> nicht so schönes Beispiel

Java-Versionen seit September 2017 im Format: `$FEATURE.$INTERIM.$UPDATE.$PATCH`

Bedeutung der Stellen (21.0.3):

- `FEATURE` : Inkrementeller Counter, aktuell bei 21.
- `INTERIM` : Bleibt derzeit bei 0.
- `UPDATE` : Inkrementeller Counter für Updates, derzeit bei 3.
- `PATCH` : Optionale Patch-Nummer (war bei 17.0.6.1 mal nötig)

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: 21.0.7).
- Nächster LTS für Herbst 2025 angekündigt, somit 25.0.x

21.3.3. Release Notes

Ziel:

- saubere Nachführung von allen Änderungen, Erweiterungen und Korrekturen
- Nachvollziehbare Entwicklungsgeschichte

Erstellung:

- meist manuell
- evtl. unterstützt durch Issue-Tracking-Systeme
  - Bugzilla, JIRA, Mantis, Trac, Redmine etc.
  - Direkter Bezug auf Change-Request oder Bug.

21.4. Technisches Deployment (Java)

21.4.1. Techniken und Methoden

- 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`)

21.4.2. Deploymentziele und Projektarten

-> je nach Produkt unterschiedliche Techniken zur Verteilung

Applikation / Anwendung für End-User
- Verteilung in binärer Form
- evtl. mit automatischem Installationsprogramm (`setup.exe`) und Anleitung / Manual
- separierte JAR-Dateien (Komponenten) oder Single-JAR

Server-Applikation für Operating:
- klassisch
- meist mit separierter JAR-Datei für gezieltes Update
- bei Microservices eher Single-JAR
- immer beliebter als Alternative: Container-Deployment

Library / Komponente für Entwickler:
- 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`).

Vorteile einer Menge an einzelnen JARs:
- bei Dockerimages macht es Sinn
- Austausch einer einzelnen JAR Datei sehr einfach möglich

Vorteil Single JAR:
- einfacher zu verteilen

21.4.3. JAR-Datei (Java ARchiev)
- ist eine vollständige Applikation
  - kann selber aus 1..n einzelnen JARs bestehen
  - kann zusätzlich 0..n Dependency-JARs benötigen
- JAR's können extern (Name) oder intern (Manifest) versioniert sein.
  - Bei >1 JAR-Dateien ist ein Classpath notwendig
- eine Menge von JARs:
  - 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.)

21.4.4. Modularisierung seit Java 9 (Project Jigsaw)
- seit Java 9 (September 2017) echte Modularisierung implementiert

Ziele:
- Reliable Configuration
  - Der fehleranfälligen Classpath wurde 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.

21.4.4.1. 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 Imports, Exports und Abhängigkeiten definiert.
  - Somit wird eine Designverifikation zur Compile-Time möglich.
- Zusätzlich wird beim Start die Applikation eine Laufzeitprüfung durchgeführt, ob alle notwendigen Komponenten vorhanden sind.
  - `ClassNotFound` Exception nicht mehr möglich
- neues Format `jmod`, Classpath wurde von Modul-Path abgelöst
- rückwärtskompatibel

21.4.5. Beispiel `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.

```java
module ch.hslu.sort.quicksort {
  exports ch.hslu.sort.quicksort.impl;
  requires ch.hslu.sort.api;
}
```

21.4.6. Verteilung über Binär-Repositories

- Binär-Repository
  - hat nichts mit einem VSC/SCM (z.B. git) zu tun
  - strukturierte Ablage von Binaries AR, WAR, EAR, JMOD etc.)
  - Primär für Entwicklungsprozess (Dependency-Management)
- ursprünglich als „Maven-Repository“ entstanden
- 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

21.4.6.1. Verteilung über GitLab Package Repository und Regisity

- Eine Alternative stellt die Verteilung über die Plattformen wie z.B. http://gitlab.com zur Verfügung gestellten Repositories und Registries dar.
- Vorteile: Hohe Integration, Transparent, einfache Authentifizierung, sehr einfach in Pipelines zu integrieren.
- Nachteile: Funktionalität nicht mit «professionellen» Repository-Servern zu vergleichen, feingranulare Repositories.

21.4.6.2. 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.
  - Entwicklungswerkzeuge (JDK etc.) vorausgesetzt
- Alle Distributionen werden sauber versioniert
  - Ideal / Empfehlung: Semantic Versioning

21.5. Deployment Arten in Java

21.5.1. Applikation Starten

- Jede Applikation bzw. Komponente bzw. Library ist ein eigenes (Teil-)Projekt und baut je ein eigenes, versioniertes JAR.
  - 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 Resourcen.
    - Einfacher Austausch einzelner JAR-Dateien (z.B. Bugfix).
  - Variante 1b: Class-Path-Eintrag in `META-INF/MANIFEST.MF` in der JAR-Datei der 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.

Befehle

- Liste der Abhängigkeiten anzeigen (ohne test-Scope): `mvn dependency:list –DincludeScope=compile`
- Dependencies aus dem binär-Repository zusammenkopieren:
  `mvn dependency:copy-dependencies -DincludeScope=compile`
  - Resultat im Verzeichnis `./target/dependency`

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-Arbeitsplätze nutzbar/sinnvoll.
  - Wegen der Abhängigkeit zum Buildtool sicher keine Lösung für Endbenutzer

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 (;).

21.5.2. JavaFX Applikation

- Maven: Plugin von OpenJFX
  - automatisiert den korrekten Start einer (teil-)modularisierten JavaFX-Applikation
    - `org.openjfx:javafx-maven-plugin:0.0.8` (Apr. 2024)
    - Befehle `mvn javafx:run [-debug]` oder `mvn javafx:jlink`

Mittels -debug-Option von Maven kann man sich den Startbefehl (sinngemäss) herausholen. Beispiel (konzeptionell):

```shell
java
  --module-path
  javafx-base-21.0.3-win.jar;javafx-base-21.0.3.jar;
  javafx-controls-21.0.3-win.jar;javafx-controls-21.0.3.jar;
  …
  --add-modules javafx.base,javafx.controls,…
  -cp app.jar;slf4j-2.0.13.jar;logback-1.5.6; …
  ch.hslu.demo.GuiStartClass
```

Info

- Java FX nutzt seit Version 11 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 IDEs der Realität etwas hinterher.

21.5.3. Deployment als Container

Deployment als FAT-JAR in Container

Vorteile
- Bau des Images und Start Befehl sehr einfach
- simples Dockerfile zur Konfiguration
- FAT-JAR auch ausserhalb des Containers einfach nutzbar

Nachteil
- FAT-JAR kann gross werden -> Builprozess wird langsamer
- bei kleinen Änderungen muss das ganze JAR/Layer neu erstellt werden
- Problematik mit den Manifesten und Konfigurationen

Deployment als „layered“ Container

Vorteile
- Feingranulare Trennung der Applikation, Dependencies und Konfigurationen in getrennten Layern
- Unterschiedliche Änderungshäufigkeit (App. versus Deps.) beschleunigt den Build massiv und senkt Ressourcenbedarf
- Automatisierung z.B. durch Google JIB-Plugin

Nachteil
- 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

22. SOLID

Durch S.O.L.I.D erreicht man qualitativ besseres und schöneres Code-Design

Konkret:

- Höhere Wiederverwendbarkeit
- Leichtere Verständlichkeit / bessere Lesbarkeit
- Verbesserte Testbarkeit
- Vereinfachte Wartung
- Verbesserte Erweiterbarkeit
- Leichteres Refactoring

22.1. Single Responsibility Principle

- **Eine Klasse** soll **nur eine Verantwortung** haben.
- **Nur ein Grund** zur Änderung pro Klasse.

22.1.1. Vorteile

- **Hohe Kohäsion**: Was zusammengehört, bleibt zusammen.
- **Niedrige Kopplung**: Bessere Wart und Erweiterbarkeit.
- **Mehr, kleinere Klassen** im Design.
- Bessere Wiederverwendbarkeit.

22.1.2. Probleme bei Missachtung

- **Hohe Komplexität**
- **Grosse „Gott“-Klassen**
- **Schlechte Wartbarkeit**
- Häufige Verletzung in der Praxis!

22.1.3. Beispiel

Nicht gut:

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

Gut:

```java
interface Transmit {
  void send(char character);
  char receive();
}

interface Connection {
  void dial(String phoneNumber);
  void hangup();
}
```

22.2. Open Closed Principle

- Eine Klasse soll **offen für Erweiterungen**, aber **geschlossen gegenüber Modifikationen** sein.

22.2.1. Offen für Erweiterungen

- Design erlaubt einfache/sichere Erweiterung
  - Beispiel: Strategy-Pattern (GoF)
  - Erweiterung durch neue Klassen

22.2.2. Geschlossen für Änderungen

- Bestehende Klassen/Methoden bleiben unverändert.
- Reduziert das Risiko, neue Fehler einzubauen.

22.2.3. Vorteil

- Reduktion des Risikos neue Fehler einzubauen
- eliminiert ie `if/switch` statements
- Code wird verständlicher

22.2.4. Beispiel

Schlecht:

```java
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;
}
```

Gut:

```java
interface Operation {
  double calc(double arg1, double arg2);
}

double calc(Operation op, double arg1, double arg2) {
  return op.calc(arg1, arg2);
}
```

22.3. Liskov Substitution Principle

- Subtypen müssen sich wie ihre Basistypen verhalten.
- Spezialisierungen dürfen Verhalten **erweitern**, aber **nicht fundamental verändern**.
- Ziel: Austauschbarkeit ohne Überraschungen oder Fehler im Code.

Vererbung **kritisch hinterfragen**:

- Subtyp **ist ein** (is-a) Basistyp?
- Subtyp **verhält sich wie** (behaves-as) der Basistyp?
- Favor Composition over Inheritance

22.3.1. Beispiel

Schlecht:

```java
public class Bird {
    public void fly() {
        System.out.println("Bird fly!");
    }
}

public class Penguin extends Bird {
```

```
    public void fly() {
        throw new UnsupportedOperationException("Penguins can't fly!");
    }
}
```

Gut:

```java
public abstract class Bird {
    public abstract void move();
}

public class FlyingBird extends Bird {
    public void move() {
        System.out.println("Flies!");
    }
}

public class Penguin extends Bird {
    public void move() {
        System.out.println("Swims!");
    }
}
```

22.4. Interface Segregation Principle

- Schnittstellen (Interfaces) **klar von Implementierungsdetails trennen**
- **Keine „fetten" Interfaces**, lieber **viele kleine**
- **Nur relevante Methoden** anbieten ⟶ hohe **Kohäsion**
- **Minimale Kopplung** zwischen Komponenten
- **Klienten sollen nur abhängen**, was sie wirklich brauchen

22.4.1. Umsetzung

- **Feingranulare Interfaces** entwerfen
- Bei verschiedenen Klient-Typen ⟶ **eigene Interfaces**
- **Abhängigkeiten reduzieren**, erhöht auch **Sicherheit**
- **Abstrakte Basisklassen** als Interfaces nutzen
- **Vererbung von Interfaces möglichst vermeiden**

22.4.2. Bei Refactorings

- **Superklasse extrahieren** ⟶ Achtung: keine überladene Schnittstelle!
- **Interfaces extrahieren** ⟶ fördert **Komposition statt Vererbung**
- **Bestehende Interfaces aufsplitten** (Seperation of Concerns, Single Responsiblity Prinicple)

22.5. Dependency Inversion Principle

- **Änderungen voneinander isolieren**
- Abhängigkeiten **nur zu stabileren Komponenten** (in Richtung höherer Abstraktion)
- **High-Level-Klassen** sollen **nicht** von **Low-Level-Klassen** abhängen
  - **Beide** sollen von **Interface** abhängen
- **Interfaces** sollen **nicht von Details** abhängen, sondern **Details von Interfaces**
- Abstraktionsschicht dazwischen ⟶ entkoppelt Konzept von konkreter Umsetzung

22.5.1. Vorteile

- **Geringe Kopplung**, **hohe Flexibilität**
- **Einfachere Testbarkeit** (z. B. durch **Test Doubles**)
- Ermöglicht **Austauschbarkeit** der Implementierung
- Nutzt oft **Dependency Injection (DI)**

- Entspricht häufig dem **Adapter-Pattern** (GoF)

23. CUPID

Idee hinter C.U.P.I.D.-Eigenschaften

- Entwickelt als Ergänzung zu S.O.L.I.D. von Daniel Terhorst-North und 2021 veröffentlicht.
- Keine «sturen» Regeln und Richtlinien (die nicht eingehalten werden) definieren, sondern positive Eigenschaften formulieren, die mehr oder weniger erfüllt werden können.

23.1. Composable

lässt sich gut mit anderem nutzen

Guter Code hat eine kleine „Oberfläche“

- Nur relevante Teile von aussen sichtbar
- Wiederverwendung soll möglichst **einfach** sein
- Weniger Fehler & Konflikte bei Änderungen
- **Hohe Kohäsion**, **schmale & einfache Schnittstelle**

Eindeutig definierte Schnittstellen

- Code leichter zu **verstehen** und **nutzen**
- **Intention** des Codes soll klar erkennbar sein
- Wissen über interne Details ist **nicht nötig**

23.2. Unix Philosophy

kümmert sich genau um eine Sache

- **Guter Code** erledigt **eine Aufgabe**, aber **richtig gut**
- Jede Softwareeinheit (Programm, Klasse, Methode …) hat **nur eine klar definierte Verantwortung**
- **UNIX-Prinzip** Viele kleine Tools, die sich **gut kombinieren** lassen
- Parallele zum **SRP (Single Responsibility Principle)**

23.3. Predictable

macht das, was man erwartet

- **Keine Überraschungen**: Code verhält sich wie erwartet
  - Fachlich: **Keine Nebeneffekte**
  - **Aussagekräftige Namensgebung** – keine falschen Versprechen
- **Deterministischer Ablauf**, keine „Magie“
  - Verhalten ist **beobachtbar**
  - Vereinfacht **Debugging** und **Fehlersuche**
- **Interner Zustand** ist **nicht direkt veränderbar**
  - Nur aus dem Verhalten ableitbar
  - → **Datenkapselung**

23.4. Idiomatic

fühlt sich natürlich an

- Code ist leicht **verständlich**
- Es werden die Idiome des Kontextes (Domäne, Firma, Team) sowie des Ökosystems der Sprache, in der er geschrieben ist, verwendet.
  - Namenskonventionen einhalten

23.5. Domain based

sowohl in Sprache als auch Struktur

- **Guter Code nutzt die Sprache & Struktur der Domäne**
  - Nicht durch Frameworks oder technische Konzepte dominiert
  - → Lösung statt Technik im Fokus
- **Domänenbegriffe statt technischer Begriffe verwenden**
  - z. B. „Dokument“ statt „Model“, „Zahlung“ statt „Controller“
  - Verbessert Verständlichkeit und Wartbarkeit

- **Struktur des Codes spiegelt die fachliche Lösung**
  - Domänengrenzen = **Modul und Deployment-Grenzen**
  - Architektonisch: **vertikale statt horizontale Trennung**

24. Komponentenmodell

DEFINITION: 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.

Ein Komponentenmodell:
- Legt grundlegende Eigenschaften der Komponenten fest,
- Erstellt einen Rahmen für die Entwicklung, oft in Zusammenhang mit der Laufzeitumgebung,
- Ist Basis für die Interaktion zwischen Komponenten.

Minimal Anforderungen an ein Komponentenmodell:

Eigenschaft	Inhalt
Schnittstellendefinition	Wie wird die Schnittstelle einer Komponente beschrieben?
Kommunikation und Verteilung	Wie kommuniziert die Komponente mit anderen Komponenten? Ist eine Verteilung auf mehrere Prozesse bzw. physische System möglich?
Komposition und Auffindbarkeit	Können zwei Komponenten miteinander verbunden werden? Wie wird dies gemacht?
Deployment	Wie werden Komponenten ausgeliefert und gebündelt?

Weiter Eigenschaften können: ‚Sicherheit', ‚Logging', ‚Monitoring' sein.

Beispiel im Logger-Projekt

Eigenschaft	Inhalt
Schnittstellendefinition	Java-Interfaces, gRPC
Kommunikation und Verteilung	Lokal: SharedMemory / Methodenaufrufe Verteilt: RPC (via TCP/IP)
Komposition und Auffindbarkeit	• ServiceProvider • Spring • Hostname / Port
Deployment	Container und Jar

24.1. OSGI-Komponentenmodell

OSGI ist ein **leichtgewichtiges** Java Modul- und Servicesystem und definiert ein vollständiges Komponentenmodell.

OSGI ist verbreitet und wird z.B. von NetBeans, Confluence und Jira, etc. verwendet.

24.1.1. Bundle

In OSGI ist Bundle die Bezeichnung von Komponenten bzw. Modulen.

- Komponenten sowie Schnittstellen werden als Bundles ausgeliefert.
- Ein Bundle wird immer als JAR mit einem Manifest `META-INF/MANIFEST.MF` ausgeliefert.

Beispiel Manifest

```yaml
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
```

Im Manifest-File wird definiert, was exportiert wird und was die Requirements sind.

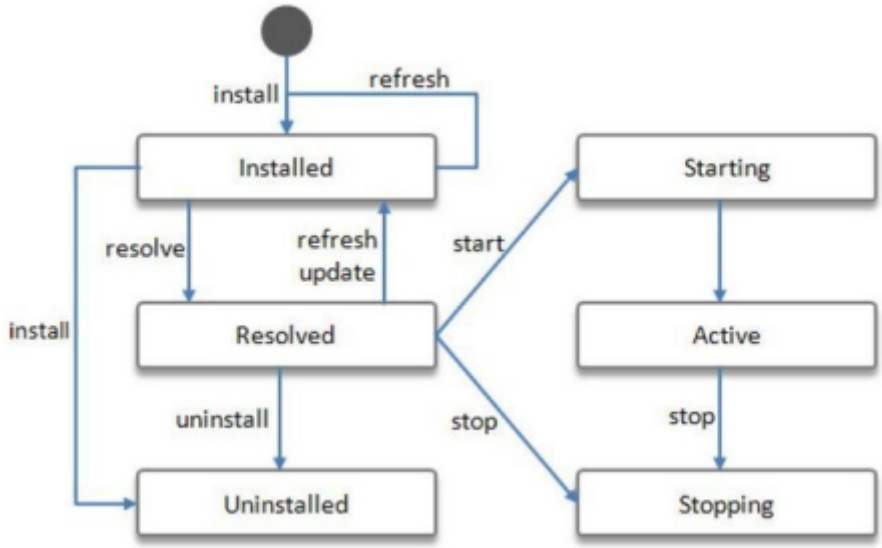
Nur die Exportierten Packages sind für andere Bundles ersichtlich -> **Eliminiert** die Problematik, mit sich **überschneidenden Klassen und Methoden**.

=> Schnittstellen werden als separate Bundles ausgeliefert

24.1.2. Isolation durch Eigene Classloaders

In OSGI wird für jedes Bundle ein eigener Classloader genutzt. So wird eine Art ‚Namespace' für die verschiedenen Bundles erzeugt.

24.1.3. Lebenszyklus



- Dynamisches Laden von Modulen
- Ideal für Plugins (kein Neustart, aber keine garantierte Verfügbarkeit)
- Bundle wird via Bundleactivator benachrichtigt, wenn es aktiviert wird.

Bundleactivator:

```java
public class Activator implements BundeActivator {

  // Aufruf wenn Bundle aktiviert wird
  public void start(BundeContext context) {

  }

    // Aufruf wenn Bundle deaktiviert wird
  public void stop(BundeContext context) {

  }
}
```

24.1.4. Service Registry

Die Service-Registry ist ein zentrales Verzeichnis, welches alle Services enthält:

```
                            -----------
 -----------------          | OSGI      |              -----------------
| Consumer Bundle |---get-->| Service   |<--register--| Provider Bundle |
 ----------------           |  Registry |              -----------------
        |                    -----------
        |                        ^
         -------listen-----------|
```

Ein Service ist eine Objektinstanz, welche eine bestimmte API implementiert und in der Registry publiziert wird.

Service registrieen

```java
Hashtable<String, String> properties = new Hashtable<>();
properties.put("name", "Lowercase Transform");
context.registerService(TextService.class.getName(), new TextServiceImpl(),
                        properties);
```

Service finden

```java
// Abrufen aller Service-Instanzen, die TextService implementieren:
ServiceReference[] refs = context.getServiceReferences(TextService.class.getName(),
                                                       null);
// Überprüfen, dass mindestens eine Service-Instanz gefunden wurde
if (refs.length == 0) throw new RuntimeException("not services found");
// Referenz zu erster gefundenen Service-Instanz
ServiceReference serviceReference = refs[0]
```

Service verwenden

=> Services müssen vor Verwendung gepinned werden (sie sind volatil) und nach Verwendung freigegeben werden

```java
ServiceReference serviceReference = ...; // enthält Referenz zu Service
// Service-Instanz pinnen
TextService textService = (TextService) context.getService(serviceReference);
// Service-Instanz verwenden
output = textService.translate(text);
```

```
// Service-Instanz unpinnen
context.ungetService(serviceReference);
```

24.1.5. OSGI-Komponenten Modell

Eigenschaft	Inhalt
Schnittstellendefinition	Java-Interfaces
Kommunikation und Verteilung	Method Calls
Komposition und Auffindbarkeit	• Import/Export von Interfaces • Service-Registry • Isolation der nicht exportierten Packages
Deployment	• Einzelne JARs • Bundle JAR

24.2. Microservices

DEFINITION: „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)

Ziel: bessere Aufteilung auf verschiedene Entwicklungsteams (Konzentration auf Fachdomäne)

Eigenschaften
- **Aufteilung in Services:** Anwendung in kleine Services, welche als Komponenten verwendet werden
- **Schichten übergreifend**: enthält alle Schichten einer Funktionalität
- **Unabhängiges Deployment:**
- **Kommunikation über Netzwerk:** REST, MQ, SOAP, etc.
- **Entkopplung:** Microservice nur über Schnittstelle bekannt.
- **Technologische Entkopplung:** pro Microservice kann z.B. eine andere Sprache verwendet werden.

24.2.1. Komponentenmodell

Eigenschaft	Inhalt
Schnittstellendefinition	OpenAPI
Kommunikation und Verteilung	• HTTP Ressourcen API (REST) • Verteilung über virtuelle und physische Systeme möglich
Komposition und Auffindbarkeit	• Komposition mittels Docker-Compose • Automatische Vergabe von Hostname/Port an Service-Anbieter. • Automatische Übergabe von Hostname/Port an Konsumenten z.B. mit Umgebungsvariablen • Isolation durch unabhängige Prozesse
Deployment	• Gebündelt mit HTTP-Server und allen Abhängigkeiten als Container Image • Abgelegt in Binary-Repo

Kommunikation:
- Muss Interprozess-Kommunikation sein
- „often an HTTP resource API“

Schnittstellendefinition Beispiel:

```yaml
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
```

Verbinden von Services:

- Regulär via Hostname/Port
  - Statische Zuweisung von Hostname/Port per Konfig File nicht praktikabel
- Umgebung sehr variablel

Möglichkeiten zur Konfiguration:

- Komposition mittels Skripts (selten) oder Orchestrierungstools (Docker Compose, Kubernetes)
- Export: Orchestrierungstools/Skripts vergeben Hostname/Port an Container z.B. mittels Umgebungsvariablen
- Import: Orchestrierungstools/Skripts injizieren Hostnamen/Port in Konsumenten z.B. mittels Umgebungsvariable oder Programmargumenten

Möglichkeiten zum Deployment

- Kein eigener Prozess
  - Normale Webservices im Applikationsserver
- Bündeln mit vollwertigem Applikationsserver
  - Komplex benötigt viele Ressourcen
- Ausführbares JAR inklusive Server
  - nicht Technologie neutral
- Container-Images mit allen Abhängigkeiten

25. Technische Schuld

25.1. Code Qualität

Es gibt viele Ausreden, warum Code-Qualität bei einem Projekt nicht wichtig ist:
- Dokumentieren kann man (wenn überhaupt) Später. -> Ist ja „nur“ Beispiel/Test-Projekt
- Code kommt von ChatGPT, muss man nicht testen.
- Code sollte gestern fertig sein, keine Zeit für statische Code-Analyse oder Q-Ratings.
- Projekt ist zu klein, lohnt sich nicht
- etc.

Professionelle Entwickler*innen
- Nutzen selbständig vorhandene Werkzeuge
- Sind offen für alle Hinweise aus allen Tools
- Lassen sich von „false-positiv“ nicht ärgern
- Sehen Potenzial zur laufenden Verbesserung.
- Motivieren auch andere für das Thema

25.2. Statische Codeanalyse

Für viele Sprachen gibt es Tools, welche Hinweise zur Codequalität liefern. Meistens sind diese:
- Freistehend -> In CI/CD nutzbar.
- Individuell und projektspezifisch konfigurierbar
- Plugins für IDE

Beispiele: Checkstyle, PMD, Spotbugs, SonarQube

Verwendung über: `mvn site`

25.2.1. Herausforderungen:

Es gibt viele verschiedene Tools, welche alle andere Schwerpunkte legen und auch verschiedene Qualitäten haben.

Die Standard-Herausforderungen sind:
- Pflege der Konfiguration (im Projekt oder Zentral)
- Unterschiedliches Reporting, Doppelbefunde.
- Umgang mit „false-positiv“
- Integration in Build- und Entwicklungsumgebung
- Wie werden Ergebnisse von verschiedenen Tools zusammengeführt?

Die Integration dieser Tools wird am einfachsten über Reporting gemacht, dies führt aber zu **langen Feedbackloops**. (Integration ist auch in IDE möglich, meist komplizierter.)

25.3. Konzept der Technischen Schuld

Man definiert für jede Art und Kategorie der Befunde einen Preis (monetär oder zeitlich), welcher für die Verbesserung aufgewendet werden muss.
- Z.b. Umbenennen einer Variable -> 5min

Die Summe aller Preise ergibt nun die **technische Schuld**. Die technische Schuld ist **nicht unbedingt genau,** dafür lassen sich Projekte vergleichen.

Durch geschickte Kategorisierung kann die technische Schuld gut aufgeschlüsselt werden. Man kann Befunde z.B. mit Einfluss auf:
- Wartbarkeit,
- Sicherheit,
- Testbarkeit

gruppieren. Daraus folgt, dass man z.B. n Stunden aufwenden muss, um die grössten Sicherheitslücken zu beheben.

25.4. SQALE Methodik

SQALE wurde 2009 entwickelt und steht für **S**oftware **Q**ality **A**ssesment based on **L**ifecycle **E**xpectations. Es werden 9 aufeinanderfolgende Kategorien definiert.

```
         ------------------------------------              ^
        / Reusability - Wiederverwendbarkeit   \           |
       /  Portability - Portabilität            \          |
      /   Maintainability - Wartbarkeit          \         |
     /    Sceurity -  Sicherheit                  \        |
    /     Usability - Benutzerfreundlichkeit       \       |
   /      Efficiency - Effizienz                    \      |
  /       Changeability - Änderbarkeit               \     |
 /        Reliability- Zuverlässigkeit                \    |
/         Testability - Testbarkeit                    \   |
```

Die Pyramide ist stufenweise aufgebaut. **Testability** als Fundament und **Reusability** als höchstes Ziel.

Viele Befunde der statischen Codeanalyse lassen sich in diese Kategorien aufteilen, daher kann die Bewertung **weitgehend automatisch** erfolgen. -> Technische Schuld kann so auch als Trend aufgezeichnet werden.

25.5. SonarQube

SonarQube ist eine Webapplikation mit zentralen Regelsets. Dadurch lassen die Projekte dezentral analysieren

- Vereinfachte SQALE-Methodik -> berücksichtigt auch dynamische Analysen
- Kann als harte Q-Gate genutzt werden
- SQALE Methodik als Plugin

Es gibt eine SonarQube Community Edition -> am besten in einem Docker Container mit DB verwenden

Folgendes Docker Compose File kann für den Einsatz von SonarQube verwendet werden

```
services:
  sonarqube:
    depends_on:
      - db
    image: sonarqube:community
    restart: unless-stopped
    ports:
      - "9000:9000"
      - "9092:9092"
    networks:
      - sonarnet
    environment:
      - SONAR_JDBC_URL=jdbc:postgresql://db:5432/sonarqube
      - SONAR_JDBC_USERNAME=sonarqube
      - SONAR_JDBC_PASSWORD=sonarpass
    volumes:
      - vsk_sonarqube_conf:/opt/sonarqube/conf
      - vsk_sonarqube_data:/opt/sonarqube/data
      - vsk_sonarqube_extensions:/opt/sonarqube/extensions
      - vsk_sonarqube_plugins:/opt/sonarqube/lib/bundled-plugins
  db:
    image: postgres:16
    networks:
      - sonarnet
    environment:
      - POSTGRES_DB=sonarqube
      - POSTGRES_USER=sonarqube
      - POSTGRES_PASSWORD=sonarpass
    volumes:
```

```
      - vsk_sonarqube_db:/var/lib/postgresql
      - vsk_postgresql_data:/var/lib/postgresql/data
networks:
  sonarnet:
    driver: bridge
volumes:
  vsk_sonarqube_conf:
  vsk_sonarqube_data:
  vsk_sonarqube_extensions:
  vsk_sonarqube_plugins:
  vsk_sonarqube_db:
  vsk_postgresql_data:
```

=> Es werden auch gleich Netzwerke und persistente Volumes definiert

Um den Docker zu starten/stoppen/loggen:

```
docker compose up –d
docker compose down
docker compose logs -f
```

Um eine Analyse durchzuführen

- Dafür muss aber eine token unter http://localhost:9000/account/security gelöst werden was als Parameter an mvn übergeben wird
- `mvn sonar:sonar -D"sonar.host.url=http://localhost:9000" -D"sonar.token=sqa_token`

IDE Plugin SonarLint ist frei verfügbar

26. Softwarekonfigurationsmanagement

Ziel: Änderungen von Konfigurationen über gesamten Systemlebenszyklus kontrolliert unter Einhaltung von Integrität und Rückverfolgbarkeit durchführen.

Methodik: Identifikation der Konfiguration eines Systems
- auf verschiedenen Ebenen
- zu verschiedenen, definierten Zeitpunkten

Während der Entwicklung und im Betrieb eines Systems sind Bestandteile Änderungen unterworfen. Dazu gehören:
- Software
- Hardware
- Konfigurationen (Einstellungen)
- Firmware
- Dokumentation

⟹ Nicht alle Versionen aller Bestandteile sind miteinander kompatibel. Softwarekonfigurationsmanagement stellt diese Kompatibilität sicher.

26.1. Konzept & Begriffe

Konfiguration: Sammlung bestimmter Versionen von Bestandteilen einer Software, welche gemäss einem festgelegten Verfahren kombiniert werden, um einen bestimmten Zweck zu erreichen.

Konfigurationselement: Aggregation von Software, welche im Rahmen des Konfigurationsmanagements, als einzelne Einheit behandelt wird.z.B.:
- Pläne
- Spezifikationen
- Bibliotheken
- Daten und Verzeichnisse
- Dokumentation für Installation, Wartung und Betrieb

Version: Spezifisches, identifizierbares Artefakt eines Konfigurationselements auf einem bestimmten Entwicklunsstand.

Revision: Neue Version eines Artefakts mit dem Zweck, eine ältere abzulösen.

Baseline: Ein Satz von Revisionen, d.h. ein Snapshot der Konfiguration

Release: Eine getestete und freigegebene Baseline

26.2. Klassisches Konfigurationsmanagement nach IEEE

<table>
<tr><td rowspan="2">Management und Planung</td><td>Kontrolle
Umgang mit Änderungen*</td><td>Accounting
Unterstützung für Projektmanagement und Entwicklung</td><td>Auditing
Nachvollziehbarkeit</td></tr>
<tr><td colspan="3">Identifikation von Konfigurationen
Basis für das Konfigurationsmanagent</td></tr>
</table>

* Managment Autorisiert, Entwicklungsteam koordiniert

26.2.1. Konfigurationsmanagementplan nach IEEE

Struktur und Inhalte eines Konfigurationsmanagementplans nach IEEE 828-1990:

SCM = Softwarekonfigurationsmanagement

Abschnitt	Inhalte
1. Einführung	Zweck, Gültigkeitsbereich, Begriffsdefinitionen.

Abschnitt	Inhalte
2. SCM Management	**(WER)** - Zuständigkeiten und verantwortliche Stelle für die Umsetzung der geplanten SCM-Aufgaben
3. SCM Tätigkeiten	**(WAS)** - Alle SCM-Aufgaben, die im Rahmen des Projektes zu erfüllen sind.
4. SCM Termine	**(WANN)** Termine für die SCM-Aufgaben in Abstimmung mit dem Projektablauf.
5. SCM Ressourcen	**(WIE)** Benötigte Werkzeuge, Personal und Sachmittel zur Umsetzung der geplanten SCM-Aufgaben.
6. SCM Aktualisierung	Wie wird sichergestellt, dass dieser Plan dem Projektablauf entsprechend nachgeführt wird.

26.3. Modernes Softwarekonfigurationsmanagement

Modernes Softwarekonfigurationsmanagement besteht aus folgenden Teilen:

- Source Code Management
- Build Engineering
- Environment Configuration
- Change Control
- Release Management
- Deployment

Ein Softwareentwicklungsprozess wird oft durch eine Pipeline mit verschiedenen Stages repräsentiert. Ziel ist, dass alle **Input-** und **Output-Artefakte** exakt **identifizierbar** und **reproduzierbar sind**. (Pipeline ist möglichst vollständig automatisiert, siehe Dev-Ops)

Beispiel Pipeline

Entwicklung -> Test -> Staging / QA -> Produktion/Release

26.3.1. Source Code Management

Version Control Systems bilden die Basis für Konfigurationsmanagement. Alle Bestandteile müssen identifizierbar sein. (Code, Doku, Binärdateien, etc.)

Verschiedene Codeprojekte können in separaten Repositories verwaltet werden oder in einem einzigen.

Falls mehrere Repositories verwendet werden, muss eine **Verbindung** zwischen den **Versionen der Repositories** hergestellt werden. Dies wird durch **Tags** sichergestellt. Der Tag enthält dann die Versionsnummer.

Oft werden auch verschiedene Repository-Typen eingesetzt:

Source-Code	Git
Builds / Binärdateien	Nexus
Container-Images	GitLab
Umgebungs-Konfiguration	Git
Manuelle Testfälle	Spez. Tools
Benutzer-Dokumentation	Spez. Tools
Release-Konfiguration	Git

26.3.2. Build Engineering

Durch das Build-Engineering werden **identifizierbare** und **reproduzierbare** Builds sichergestellt.

Dabei sind:
- Builds verständlich, wiederholbar, schnell und zuverlässig.
- Jede Konfiguration identifizierbar.
- Source- und Compile Abhängigkeiten leicht zu ermitteln.
- Build-Anomalien leicht zu identifizieren und auf akzeptable Weise zu verwalten
- Ursachen für fehlerhafte Builds schnell und einfach zu identifizieren und beheben.

Dies wird sichergestellt durch:
- Verwendung **exakter Versionen** (kein SNAPSHOT oder latest)
- Deterministischer Builds (Gleicher Input => gleicher Output)
  - Ursachen für nicht deterministische Builds:
    - Timestamps
    - Parallele Kompilation -> Linking Reihenfolge bei C (=> Alphabetische Sortierung)
    - Code-Generierung

26.3.2.1. Build-ID

Mithilfe einer automatisierten „immutable[1]“ Build-ID welche sich **im Buildartefakt** befindet, kann ein Build zuverlässig identifiziert werden.

Beispiel für das Einbetten einer Build-ID in C Artefakt

1. Zuerst wird die Build-ID (Commit-Hash) dem Build-Tool (hier Makefiles) mitgegeben:

```yaml
build:
  stage: build
  script:
    - make BUILD_ID=\"$CI_COMMIT_SHA"
  artifacts:
    paths:
      - logger
```

2. Die Build-ID wird an den Code als Parameter weitergereicht

```shell
BUILD_ID="manual"
logger: main.c
        gcc -o logger main.c -D BUILD_ID='$(BUILD_ID)'
```

3. Einbettung der ID im Build-Artefakt:

```c
#include <stdio.h>
int main(int argc, char* argv[]) {
  printf("Distributed Logging System (build ID: %s)\n", BUILD_ID);
  return 0;
}
```

Es ist wichtig, dass auch das **Dockerimage** oder die **Infrastruktur**, auf welchem der Build entstanden ist, **wiederherstellbar** ist. Dies sollte in regelmässigen Abständen **getestet** werden.

Alternativen zu Build IDs:
- Disk einer Entwicklungsmaschine abspeichern
- Exaktes Manual, wie eine Umgebung aufgesetzt werden kann

26.3.3. Envirnoment Konfiguration

Die Ziele der Envirnomen Konfiguration sind:

[1] Echte Immutabilität nur möglich mit Signaturen, oft reicht schwierig änderbar

- Pro Komponente existiert **nur ein Buildartefakt**. (Selber Build für Development-, Test- und Produktive-Systeme)
- **Umgebungen** sind zur Laufzeit **identifizierbar**.
- Umgebungskonfigurationen sind **dokumentiert** und **identifizierbar**.
- Umgebungskonfigurationen sind **versioniert**.
- **Reproduzierbare** Entwicklungs-, Test-, QA-, und Produktionsumgebungen.

Entwicklung	Test	Staging	Produktion
Code / Komponenten			
Daten	Daten	Daten	Daten
Konfiguration	Konfiguration	Konfiguration	Konfiguration

- Daten der Umgebung sind immer spezifisch
- Konfigurationen sind ebenfalls spezifisch, aber versioniert

Entkopplung von Code und Umgebungsinformationen
Umgebungsinformationen dürfen nicht hardcodiert werden, da sich ansonsten der Build für die verschiedenen Systeme unterscheidet. Daher werden sie entweder über ein **Konfigurationsfile** konfiguriert oder über **Umgebungsvariable injiziert**.

Beispiel 1: Umgebungsvariable durch Skripting Wird angewendet wenn keine Containervirtualisierung zur Verfügung steht

```sh
## set_env.sh ##
export MAIL_HOST=testmail.example.com
export MAIL_PORT=443
export PAYMENT_HOST=testpay.service.com
export PAYMENT_PORT=443
export BACKEND_HOST=test1.example.com
export BACKEND_PORT=3003
export DATABASE_HOST=test1.example.com
export DATABASE_PORT=2031

## start_test.sh ##
~$ source ./set_test_env.sh
~$ java –jar mybackend.jar \
      –port BACKEND_PORT \
      -mailHost=${MAIL_HOST} \
      –mailPort ${MAIL_PORT} \
      -dbHost=${DATABASE_HOST} \
      -dbPort=${DATABASE_PORT} \
      -payHost=${PAYMENT_HOST} \
      -payPort=${PAYMENT_PORT
```

Beispiel 2: Umgebungsvariable durch Docker-Compose

```yaml
services:
  web:
    image: myserver
    ports:
      - "433:5000"
    environment:
      LISTEN_PORT: 5000
      REDIS_HOST: redis
      REDIS_PORT: "6379"
```

```
      EMAIL: smtp.test.ch
    volumes:
      - logvolume01:/var/log
    links:
      - redis
  redis:
    image: redis
    envirnoment:
      LISTEN_PORT: 6379
```

Hier werden alle Umgebungsvariablen unter dem Punkt ‚environment' gesetzt. Pro Umgebung gibt es ein docker-compose.yaml File.

Die einzelnen Services lesen die Variablen dann aus:

```python
import time
import os
import redis
from flask import Flask

app = redis.Redis(
  host=os.environ["REDIS_HOST"],
  port=os.environ["REDIS_PORT"]
)
```

Weiter Tools zur Umgebungskonfiguration:
- Docker Swarm / Kubernetes / Open Shift: Transpartente Verteilung auf mehreren Systemen
  - Verschlüsselung / Lastverteilung / Redundanz
  - Separate Tools (Trafik / haproxy / etc.) sind meistens mächtiger
  - Redundanz muss von der Datenbank unterstützt werden
- Helm: „Package Manager" für Kubernetes

⟹ Konfiguration muss versioniert werden können.
- Ideal ist Infrastructure as Code (z.B. Docker-Compose)

26.3.4. Change Control

Das Ziel von Change Control ist „Kontrollierte und nachvollziehbare Änderungen" an allen Softwarekonfigurationselementen durchzuführen. Das beinhaltete folgende Fragen:

- Darf und soll eine Änderung gemacht werden?
- Wer darf Codeänderungen machen
- Wo sollen Codeänderungen stattfinden (in welchem Release)
- Wer darf auf welchem Environment deployen?

In GitLab können viele solcher Regeln definiert werden.

26.3.4.1. Change Control Board (CCB)

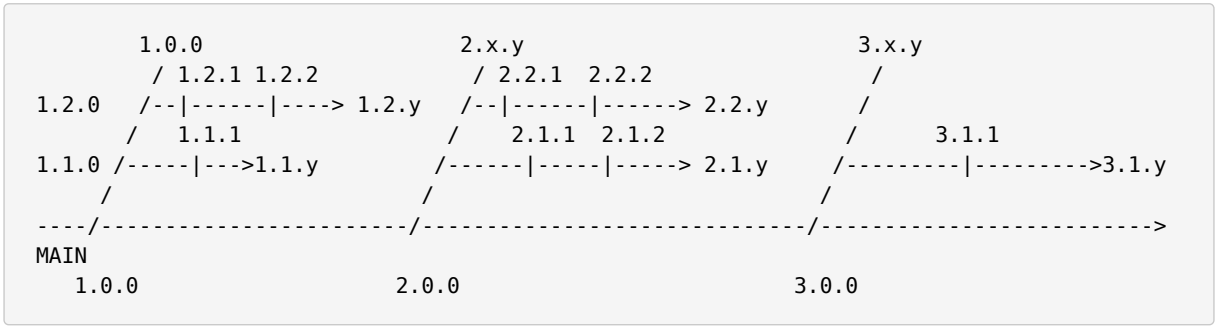
Das CCB koordiniert Änderungen pro Release und Umgebung. Das CCB kann verschieden zusammengestellt werden:

- Kleiner Kreis von Seniors
- Releasemanager pro Release mit Stellvertreterregelung
- Senior pro Komponente mit Stellvertreterregelung
- N Peers

Das CCB legt fest, welche Änderungen, in Anbetracht des Risikos, in welchem Release gemacht werden: **Gatekeeping**.

Das CCB legt auch die Standardkorrekturregeln fest, z.B.:
- Unkritische Korrekturen von Releases mit gemeldeten Fehlern nach oben beheben
  (keine Regression nach oben, keine unnötigen Korrekturen nach unten)
- Kritische Korrekturen: Branches im Vorfeld nach unten zu allen noch von Kunden verwendeten Releases
  sowie Kommunikation mittels Critical-Known-Issue-Prozess (CKI) aufsetzen.

Beispiel:



1. Unkritischer Fehler gemeldet in `2.2.2`
   -> Korrektur in `2.2.2`, `2.2.y`, `2.x.y`, `3.1.y`, `3.x.y`, `MAIN`
2. Kritischer Fehler gemeldet in Release `3.1.1`, eingeführt auf dem MAIN zwischen `2.0.0` und `3.0.0`
   -> Korrektur in `MAIN`, `3.1.y` -> Benachrichtigung aller Kunden welche `3.1.1` einsetzen

26.3.5. Release Management

Ziel: Schneller, planbarer und wiederholbare automatisierter Prozess, um einen Release zu paketieren. Alle Komponenten sind dabei eindeutig identifizierbar.

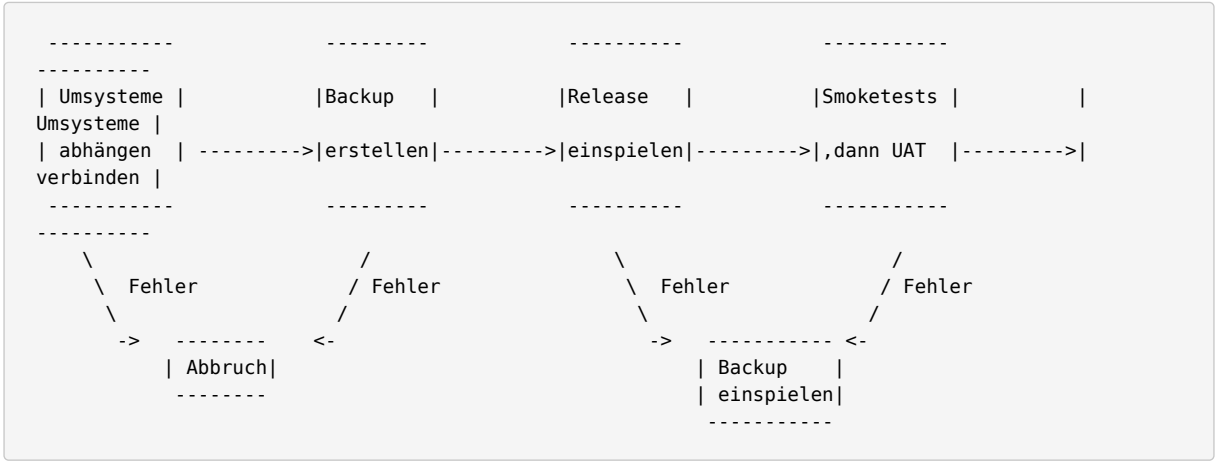
Die Tätigkeiten des Release Managements sind: Releaseplan erstellen und Releasestruktur in allen Repos abbilden.

Typischerweise wird ein Skript benutzt, welches alle Bestandteile von verschiedenen Repositories bündelt und paketiert.

26.3.6. Deployment

Ziel des Deployments ist die problemlose Einführung eines bestimmten Releases auf Productions (ggf. QA)-Environment.

Das Einspielen des Releases soll wie ein Lichtschalter funktionieren:
- Verantwortliche Personen wissen zu jedem Zeitpunkt, welcher Release auf der Produktionsumgebung läuft.
- Verantwortliche Personen müssen das Deployment bei Problemen rückgängig machen können.



- Umsysteme müssen abgehängt werden, damit Backups problemlos eingespielt werden können.

27. Koordination verteilter Systeme

Heute wird die Zeit mittels der TAI-International Atomic Time gemessen. Sie misst seit 1.1.1958 die Anzahl Ticks der Cäsium 133-Uhren.

Warum muss es eine Einigung auf die Zeit geben?

```
Computer, auf            2144    2145    2146    2147     <- Zeit gemäss lokaler Uhr
dem der Compiler    -------|-------|-------|-------|------
ausgeführt wird             \
                             -> output.o erzeugt

Computer, auf            2142    2143    2144    2145      <- Zeit gemäss lokaler Uhr
dem das C           -------|-------|-------|-------|-------
Programm                    \
geschrieben wird             -> output.c verändert
```

Dies hat zur Folge, dass `output.c` nicht neu kompiliert wird, weil `output.c` scheinbar früher erstellt wurde

Lokale Uhren Ein Timer ist eine Schaltung im Computer, welche die Zeit verwaltet. Er misst die **Schwingungen eines Quarzkristalls unter Spannung** und erzeugt Interrupts in bestimmten Intervallen (Tick).

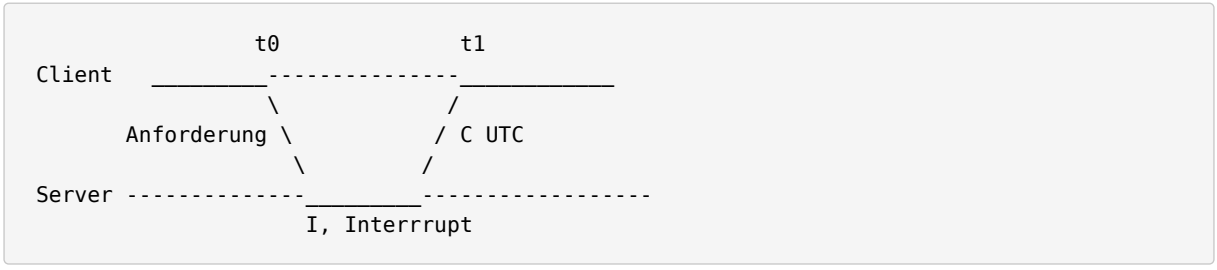
-> Diese Zeitwerte laufen auseinander (auch wenn anfänglich synchronisiert) da Quarzkristalle mit unterschiedlicher Qualität verwendet werden (-> Unterschiedliche Frequenzen). Das nennt man Uhrasymmetrie.

27.1. Algorithmus von Cristian

Zeitserver: Maschine mit Zeitzeichenempfänger[2], synchronisiert alle anderen Maschinen. **UTC**: Universal Coordinated Time: Zeitmessung in Beziehung mit dem Sonnenstand mit Schaltsekunden

Ablauf

1. Client `P` erfragt Zeit von Server `S` zum Zeitpunkt t_0
2. Die Anfrage wird von `S` bearbeitet innerhalb der Zeitspanne I
3. Die Antwort $C_{\mathrm{UTC}}(t_0)$ wird von `P` zum Zeitpunkt t_1 empfangen
4. `P` wird auf die Zeit $C_{\mathrm{UTC}}(t_0) + f\frac{\mathrm{RTT}}{2}$ gesetzt, d.h. die vom Server gemeldete Zeit plus die Rücklaufzeit des Pakets.
   - die Round Trip Time (RTT) wird dabei berechnet durch $\mathrm{RTT} = t_1 - t_0$.
   - ist die Zeitspanne I bekannt, kann die Berechnung verbessert werden: $\mathrm{RTT} = t_1 - t_0 - I$
5. Für genauere Werte wird die Laufzeit öfters gemessen, Messungen ausserhalb eines Bereiches werden verworfen und eine Mitteilung der restlichen Werte durchgeführt.



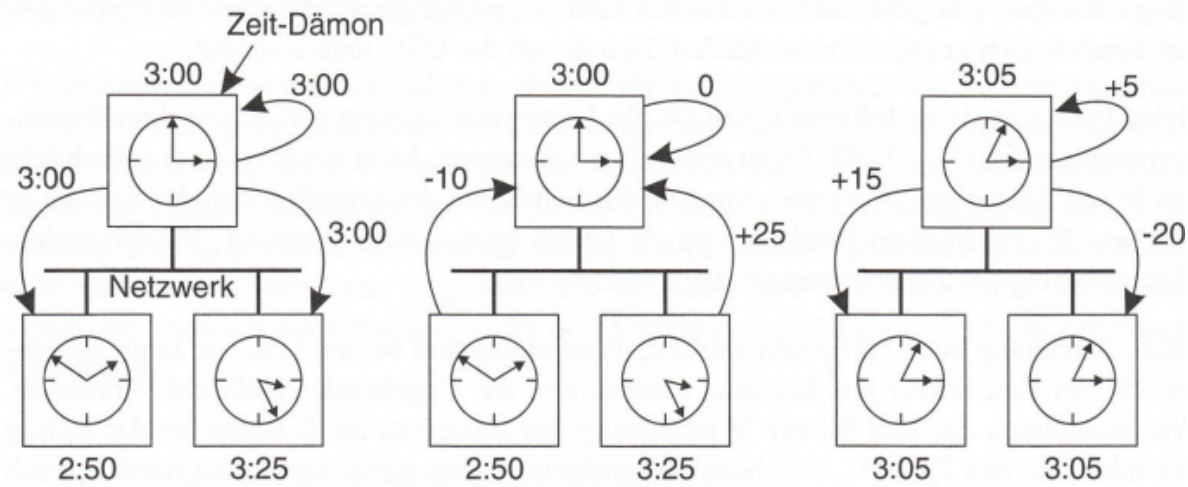
Probleme

- Zeit vom Zeitserver liegt in der Vergangenheit der lokalen Zeit
  -> Zeit kann nicht zurückgedreht werden, da inkonsistente Zustände
  => Lösung: Verlangsamung der lokalen Zeit, bis Zeitdifferenz ausgeglichen

[2] z.B. Langwellensender DCF77 https://de.wikipedia.org/wiki/DCF77

- Laufzeit der Anfrage kann nicht genau bestimmt werden, abhängig von Netzwerklast
  => Kompensation durch mehrfache Messungen der Dauer der Anfrage und Adaption des vom Zeitserver gelieferten Werts

27.2. Berkley-Algorithmus



- Wird verwendet wenn keine Maschine einen Zeitzeichenempfänger hat (z.B. abgeschottete Netze).
- Der Zeitserver (Zeit-Daemon) 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.

27.3. Network Time Protocol - NTP

NTP wurde 1982 unter der Leitung von David Mills entwickelt und befindet sich seit 1994 in der Version 4.

Eingenschaften

- NTP-Daemon auf fast allen Rechnerplattformen verfügbar. (PC bis Crays)
- Erreichbare Genauigkeit von ca 10ms in WANs und > 1 ms in LANs.
- Fehlertolerant

Struktur

- **Stratum 1:** primärer Zeitgeber, über Funk oder Standleitungen an amtliche Zeitstandards angebunden
- **Stratum > 1:** synchronisiert mit Zeitgeber des Stratums $N - 1$
- Stratum kann dynamisch wechseln

```
                Atom Uhr                    Atom Uhr
                  |                           |
-------------------------------------------------------------------
              NTP Server                    NTP server              Stratum 1
-------------------------------------------------------------------
            /           \                /       \
           /             \              /         \
-------------------------------------------------------------------
  NTP Server & Cleint    NTP Server & Cleint    NTP Server & Cleint    Stratum 2
-------------------------------------------------------------------
     /           \       /           \      /
    /             \     /             \    /
   /               \   /               \  /
-------------------------------------------------------------------
NTP Client       NTP Client         NTP Client                      Stratum 3
```

Datenpaket

<table>
<tr><td rowspan="11">Cryptosum</td><td>LI</td><td>VN</td><td>Mode</td><td>Strat</td><td>Poll</td><td>Prec</td><td rowspan="12">LI = leap indicator
VN = version number
strat = Stratum (0-15)
Poll = poll intervall
Prec = Precision</td></tr>
<tr><td colspan="6">Root Delay</td></tr>
<tr><td colspan="6">Root Despersion</td></tr>
<tr><td colspan="6">Reference Identifier</td></tr>
<tr><td colspan="6">Referece Timestamp
Seconds (32)
Fration (32)</td></tr>
<tr><td colspan="6">Originate Timestamp
Seconds (32)
Fration (32)</td></tr>
<tr><td colspan="6">Receive Timestamp
Seconds (32)
Fration (32)</td></tr>
<tr><td colspan="6">Transmit Timestamp
Seconds (32)
Fration (32)</td></tr>
<tr><td colspan="6">Ext. Filed 1 Key Identifier (optinal)</td></tr>
<tr><td colspan="6">Ext. Filed 2 Message Digest (optinal)</td></tr>
<tr><td rowspan="2">Authenticator</td><td colspan="6">Key/Algorithm Identifier</td></tr>
<tr><td colspan="6">Message Hash (64 or 128)</td></tr>
</table>

Ablauf

1. Client sendet NPT-Message an Server.
2. Server verarbeitet Paket
3. Server sendet Paket zurück
4. Client hat nun vier Timestamps ($t_1 - t_4$) und leitet davon Offset unt Delay ab:

Client	Connection	Server
Zeitstempel 1	—>	Zeitstempel 2
Zeitstempel 3	<—	Zeitstempel 4

$$\text{Offset} = \frac{(t_2 - t_1) + (t_3 - t_4)}{2}$$
$$\text{Delay} = (t_4 - t_1) - (t_2 - t_3)$$

Offset: Zeitdifferenz der Rechenuhren (gemittelt) -> Ändern um diesen Wert
Delay: Zeit während der das Paket unterwegs war -> Paket mit geringstem Delay nutzen

27.4. Logische Zeit

Leslie Lamport zeigte, dass es ausreichend ist, wenn alle Maschinen innerhalb eines Systems über dieselbe Zeit einig sind. Eine **Übereinstimmung mit der Zeit ausserhalb des Systems ist nicht notwendig**. Diese Form von Zeit wird **Logische Zeit** genannt.

Logische Zeit findet vor allem in Bereichen Anwendung, in denen Kausalitäten und Verlässlichkeit wichtig ist. Verfahren zur **Synchronisation von logischen Uhren** sind in grossen Systemen **im Allgemeinen ineffizient**.

27.4.1. Happended Before Relation

Der Ausdruck $a \to b$ wird gelesen als „a passiert vor b“ und bedeutet, dass **alle Prozesse einig** sind, dass **zuerst** Ereignis **a** stattfindet und **dann b**.

1. Wenn a und b Ereignisse im selben Prozess sind, und a vor b auftritt, gilt $a \to b$

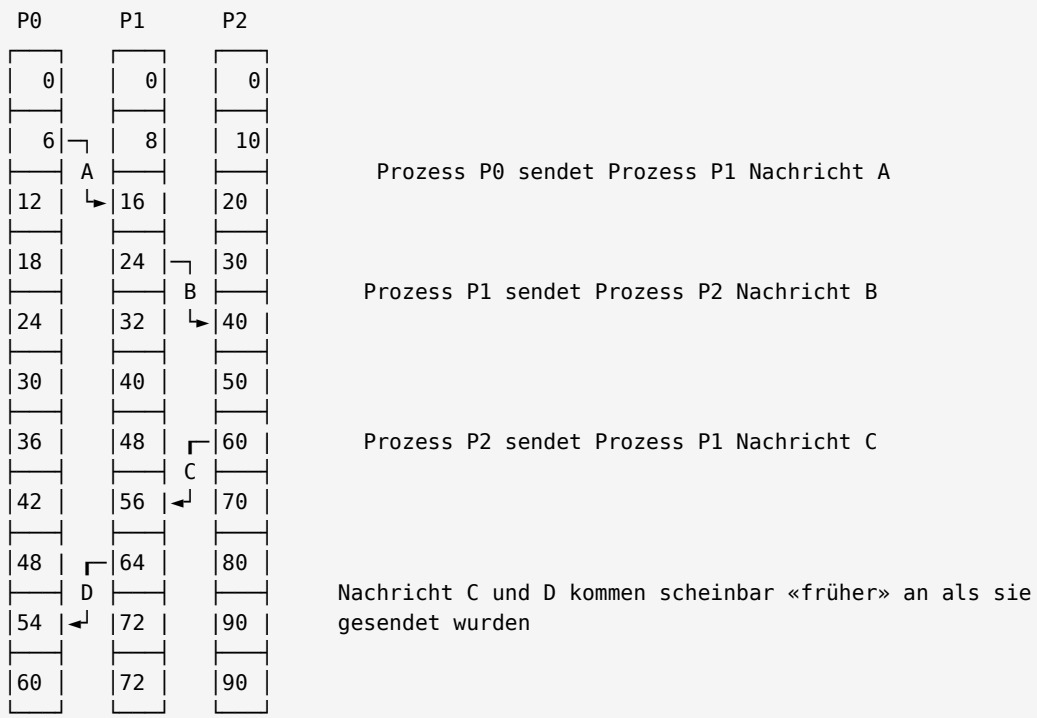
2. wenn a das Senden einer Nachricht bei einem Prozess und b das Empfangen derselben Nachricht bei einem anderen Prozess ist, gilt $a \to b$

Zwei Ereignisse $a \neq b$ sind kausal unabhängig, geschrieben als $a \parallel b$, wenn weder $a \to b$ oder $b \to a$ gilt.

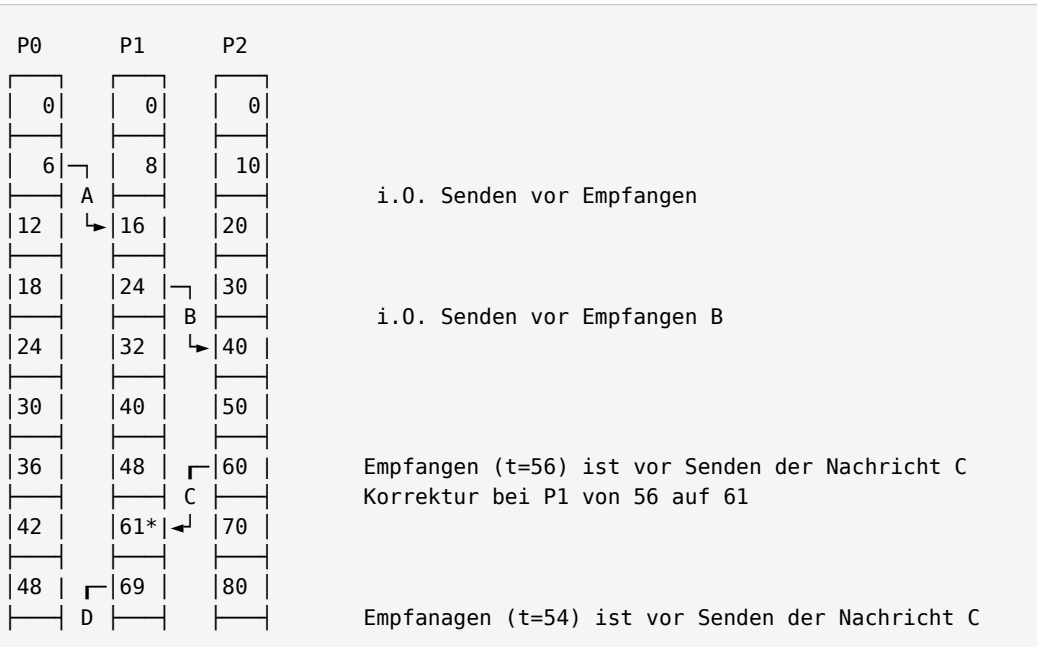
Die Happended Before-Relation ist **transitiv**: $(a \to b \land b \to c) \to (a \to c)$

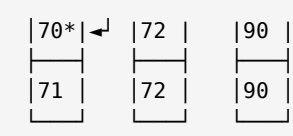
27.4.2. Lamport Zeitstempel

Jede Maschine hat eine eigene Zeit mit konstanten aber unterschiedlichen Geschwindigkeiten.



- Ein Prozess sendet eine Nachricht mit dem Zeitstempel (eigen Zeit) an einen anderen Prozess
- Einem Ereignis a wird ein Zeitwert $C(a)$ zugeordnet
  - Alle Prozesse sind sich über den Zeitwert einig
  - Wenn $a \rightarrow$ gilt auch $C(a) \rightarrow C(b)$
- Ein Prozess, welcher eine Nachricht zur eigenen Zeit b empfängt, dann müssen $C(a)$ und $C(b)$ so zugewiesen werden, dass $C(a) < C(b)$ ist.
- Die Zeit **muss immer vorwärtslaufen**
- Korrektur durch Addition von positiven Werten.





Weiter dürfen zwei Ereignisse nie zu genau derselben (logischen) Zeit auftreten. Daher wird der Zeitstempel um die Prozessnummer ergänzt.

Eigenschaften
- Lamports Uhren erfüllen die Happened Before Relation: $a \to b \Rightarrow C(a) \to C(b)$
- Definiert eine partielle Ordnung auf der Menge der Ereignisse, welche kausalen Zusammenhang erhält.
- Ergänzung zu einer totalen Ordnung ist möglich

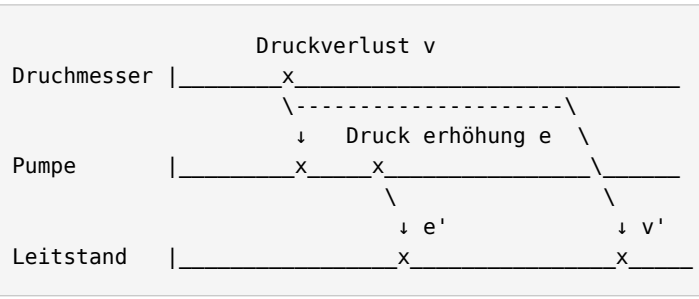
Einschränkung
- Anhand der Zeitstempel lässt sich nicht sagen, ob zwei Ereignisse kausal voneinander abhängen.

Beispiel Generationclock (GCL)
- Ein logischer Zeitstempel, welcher monoton inkrementiert wird pro Leader Election
- Persistente und transaktionssichere Speicherung
- Ziel: Nachrichten vergangener Leader können erkannt und ignoriert werden

Ablauf:
1. Jeder Prozess P startet als Follower
2. P liest seine GCL von der Disk
3. Gibt es einen Leader übernimmt P dessen GCL
4. Ansonsten gibt es eine Leader Election (P inkrementiert sein GCL)
5. Leader sendet GCL bei jeder Request

27.4.3. Vektorzeitstempel

Problem

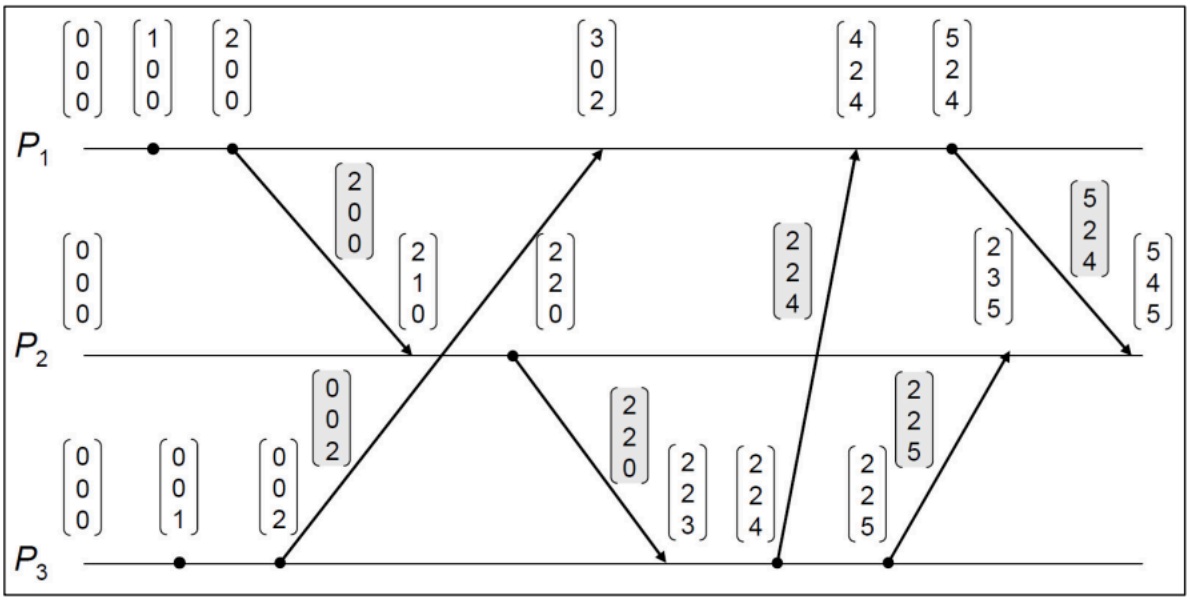


Da beim Leitstand das Signal „Druck erhöhen“ schneller ankommt als „Druckverlust“ macht er eine falsche Schlussfolgerung.

Definition
- Ein Vektorzeitstempel $\mathrm{VT}(a)$, der einem Ereignis a zugewiesen wurde, hat die Eigenschaft, dass das Ereignis a dem Ereignis b kausal vorausgeht, wenn: $\mathrm{VT}(a) < \mathrm{VT}(b)$ gilt.
- Jeder Prozess P_i besitzt einen Vektor V_i für jeden Prozess im System die Anzahl der Ereignisse enthält mit den Eigenschaften
  - V_i ist die Anzahl der Ereignisse, welche in P_i bisher 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
1. Jeder Prozess P_i hält Vektor bestehend aus n Zeilen (n = Anzahl Prozesse)
2. Initial ist der Vektorzeitstempel der Null-Vektor
3. Tritt beim Prozess P_i ein Ereignis auf, so inkrementiert er die i-te Komponente seines Vektors
4. Sendet P_i eine Nachricht, so wird der Vektor mitgesendet

5. Empfängt P_i eine Nachricht
   1. Bildet neuen Vektor mit dem komponentenweise Maximum
   2. Inkrementiert beim neuen Vektor die Komponenten i



Der Vektorzeitstempel enthält:
- 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.

Ereignis A ist eine Ursache von Ereignis B wenn:
- der Zähler für jeden Prozess im Zeitstempel $\mathrm{VT}(A)$ kleiner oder gleich dem Zähler im Zeitstempel $\mathrm{VT}(B)$ für den korrespondierenden Prozess ist.
- Mindestens ein Zähler kleiner ist.

Beispiel
- $\begin{pmatrix}2\\0\\0\end{pmatrix}$ 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 $\begin{pmatrix}1\\3\\4\end{pmatrix}$