
VSK

Verteilte Systeme und Komponenten

Author:
Eldar Omerovic
<https://www.github.com/omeldar>

Project Repository:
<https://www.github.com/omeldar/hslu>

Credits:
I completed these exercises during my VSK class
at the Lucerne University of Applied Sciences and Arts.

July 2024

Inhaltsverzeichnis

1 SW01 - CleanCode und Serverprozesse mittels Sockets und RPC	6
1.1 Clean Code	6
1.1.1 Kommentare im Code	6
1.1.2 Namensgebung	6
1.1.3 Clean Code Developer	7
1.2 Serverprozesse mittels Sockets und RPC	8
1.2.1 Herausforderungen RPC	8
1.2.2 TCP Client- und Serverprogramme mittels Sockets	10
1.2.3 gRPC	11
2 SW02 - Komponenten, Interfacedesign, SCM und Buildautomatisation (CI)	12
2.1 Komponenten	12
2.1.1 Nutzen von Komponenten	13
2.1.2 Entwurf mittels Komponenten	13
2.1.3 Begriff und Konzept der Schnittstelle	14
2.1.4 Design by Contract	14
2.2 Versionskontrollsysteme (SCM, VCS)	15
2.2.1 Grundlagen	15
2.2.2 Arbeit mit einem VCS	16
2.2.3 Konzeptionelle Unterschiede	17
2.2.4 Drei konkrete Produkte	18
2.2.5 Benutzerschnittstellen (Clients)	18
2.3 Buildautomatisation	19
2.3.1 Automatisation des Buildprozesses	19
2.3.2 Build-Werkzeuge	19
2.3.3 Build-Automation mit Apache Maven	20
3 SW03 - Messageorientierte Kommunikation, Dependecy-Management und Buildserver	21
3.1 Messageorientierte Kommunikation	21
3.1.1 Grundlagen der messageorientierten Kommunikation	21
3.1.2 Kommunikationsmuster	22
3.1.3 Transiente Kommunikation	23
3.1.4 Persistente Kommunikation	24
3.2 Dependency Management	25
3.2.1 Maven Repository	25
3.2.2 Depdendecy Management für Java	25
3.2.3 Dependency Management mit Apache Maven	25
3.2.4 Dependency Scopes	26
3.2.5 Transitive Dependencies	26
3.2.6 Versionierung und Snapshots	26
3.2.7 Managed Dependencies in Multimodul-Projekten	27
3.2.8 Deployment	27
3.3 Buildserver	27
3.3.1 Was ist ein Buildserver	27

3.3.2	Positive Effekte eines Buildservers	27
3.3.3	Buildserver Beispiele und Dienste	27
3.3.4	Konfiguration von Buildservern	28
3.3.5	Einsatz von Buildservern	28
3.3.6	Gitlab	28
4	SW04 - Architekturbeschreibung und Modularisierung	29
4.1	Architekturbeschreibung	29
4.1.1	Inhalt einer Architekturbeschreibung	29
4.2	Modularisierung und Schichtenarchitektur	30
4.2.1	Modularisierung: Konzepte und Vorgehen	30
4.2.2	Schichtenarchitektur	32
5	Entwicklung mit Container und Docker	34
5.0.1	Einsatzszenarien von Docker für die Entwicklung	34
5.0.2	Erstellen eigener Docker Images	35
5.0.3	Integration Tests mit Containern	35
5.0.4	Tool-Tipps für Docker	35
6	SW06 - Integraion- und Systemtests, Automatisiertes Testing und Test Doubles	36
6.1	Integrations- und Systemtests	36
6.1.1	Teststrategie	36
6.1.2	Integrationstests	37
6.1.3	Systemtest	38
6.1.4	Testing in Scrum	39
6.2	Automatisiertes Testing	39
6.2.1	Unit Tests	40
6.2.2	Integrationstests mit JUnit	40
6.2.3	Messung der Code Coverage	41
6.2.4	Dependency Injection	42
6.2.5	Effektives Testen mit Test Doubles	42
6.2.6	(Integrations-)Testen mit Containern	42
6.3	Testing: Test Doubles	43
6.3.1	Empfehlungen	44
7	SW08 - Fehlertoleranz, Resilienz und Entwurfsmuster	45
7.1	Fehlertoleranz und Resilienz	45
7.1.1	Fehlertolerante Systeme	46
7.1.2	Auslieferungsgarantien	46
7.1.3	Resilienz durch Wiederherstellbarkeit	47
7.1.4	Verteilte Transaktionen	48
7.2	Entwurfsmuster (Patterns)	50
7.2.1	Patterns - Klassifikation	51
7.2.2	Singleton	52
7.2.3	Fassade	52
7.2.4	Strategy-Pattern	52
7.2.5	Observer-Pattern	53
7.2.6	Adapter	53

8 SW09 - Continuous Integration (CI)	54
8.1 Continuous Integration	54
8.1.1 Die 10 Praktiken der CI	54
9 SW10 - Konsistenz, Replikation, Skalierung und Verteilung	57
9.1 Konsistenz und Replikation	57
9.1.1 CAP-Theorem	57
9.1.2 Konsistenz und Konsistenzgarantien	57
9.1.3 Replikation	58
9.1.4 Ausfallsicherheit mittels Replikation	60
9.2 Skalierung und Verteilung	61
9.2.1 Lastverteilung mittels Reverse-Proxys	62
9.2.2 In-Memory Datagrids	62
10 SW11 - Sicherheit in verteilten Systemen und Deployoment	64
10.1 Sicherheit in verteilten Systemen	64
10.1.1 Transport-Layer-Security (TLS)	64
10.1.2 Sessions	66
10.1.3 Authentifizierung und Autorisierung	66
10.2 Deployment	67
10.2.1 Deployment-Diagramme	68
10.2.2 Deployoment - Aspekte	68
10.2.3 Releases und Versionierung	69
10.2.4 Technisches Deployment (Java)	69
11 SW12 - Code-Qualität	71
11.1 S.O.L.I.D.-Prinzipien	71
11.1.1 Single Responsibility Principle (SRP)	71
11.1.2 Open Closed Principle (OCP)	71
11.1.3 Liskov Substitution Principle (LSP)	71
11.1.4 Interface Segregation Principle (ISP)	72
11.1.5 Dependency Inversion Principle (DIP)	72
11.2 C.U.P.I.D.-Eigenschaften	72
11.2.1 Composable	72
11.2.2 Unix Philosophy	72
11.2.3 Predictable	73
11.2.4 Idiomatic	73
11.2.5 Domain-based	73
12 SW13 - Komponentenmodelle uind Technische Schulden	74
12.1 Komponentenmodelle	74
12.1.1 Übersicht Komponentenmodelle	74
12.1.2 Das OSGi-Komponentenmodell	74
12.1.3 Ein Komponentenmodell für Microservices	74
12.2 Technische Schulden	75
12.2.1 Herausforderungen der statischen Codeanalysen	75
12.2.2 Konzept der technischen Schulden	75
12.2.3 SQALE Methodik	76

12.2.4	SonarQube	76
13	SW14 - Softwarekonfigurationsmanagement	77
13.1	Softwarekonfigurations management	77
13.1.1	Konzept und Begriffe	77
13.1.2	Klassisches Konfigurationsmanagement nach IEEE	77
13.1.3	Modernes Softwarekonfigurationsmanagement	78
14	SW15 - Koordination verteilter Systeme	81
14.1	Koordination verteilter Systeme	81
14.1.1	Physische Zeit	81
14.1.2	Logische Zeit	82
14.1.3	Lamport-Zeitstempel	82
14.1.4	Vektorzeitstempel	83
15	Anhang	84
15.1	ZeroMQ Code Beispiele	84
15.1.1	Half-Duplex	84
15.1.2	Data Distribution	85
15.1.3	Aufgabenverteilung	86

1 SW01 - CleanCode und Serverprozesse mittels Sockets und RPC

1.1 Clean Code

1.1.1 Kommentare im Code

- Kommentare machen keinen guten Code. Schlechter Code soll nicht kommentiert sondern umgeschrieben werden.
- Kommentare und auch Dokumentationen lügen. Nur im Code liegt die Wahrheit.
- Kommentare sind ein notwendiges Übel.

Was aber unterscheidet dabei jetzt gute und schlechte Kommentare? Generell gilt der Grundsatz: Der beste Kommentar ist derjenige, den man gar nicht braucht. Notwendige aber akzeptable Kommentare können sein:

- juristische Kommentare (Copyright, etc)
- TODO-Kommentare (aber nur temporär)
- verstärkende, unterstreichende Kommentare, welche Dinge hervorheben, die sonst zu unauffällig wären
- Kommentare zur (Er-)Klärung der übergeordneten Absicht oder zur expliziten Warnung vor Konsequenzen (Context & Warning).

Von schlechten Kommentaren gibt es viele Beispiele und Eigenschaften:

- Redundante Kommentare (CCD:DRY). Also reine Wiederholungen dessen, was schon der Code sagt.
- Irreführende Kommentare: Falsche oder Unpräzise Formulierungen
- Vorgeschrriebene oder erzwungene Kommentare: sture JavaDoc Kommentare, nur damit sie da sind.
- Tagebuch- oder Changelog-Kommentare: Dies braucht es nicht da wir Versionskontrolle haben.

- Positionsbezeichner und Banner: zur optimalen Unterteilung von grossen Quellcode-dateien.
- Zuschreibungen und Nebenbemerkungen: Hinweise auf Autor oder sinnlose Zusatzbemerkungen.
- Auskommentierter Code: Diesen wenn man ihn vorfindet immer löschen. Im Versionskontrollsystem kann nachgeschaut werden wenn es etwas braucht.
- HTML-formatierte Kommentare: Kommentar muss im Code direkt lesbar sein, nicht erst im JavaDoc.
- Zu viele Kommentare / Informationen: Oft nicht relevante / unnötige Informationen.

Ein Beispiel für einen schlechten Kommentar und wie man dies beheben kann sehen wir folgend:

```
1 // Timeout in Millisekunden  
2 int time = 100;
```

Dies kann auch ganz einfach zu folgendem Code geändert werden:

```
1 int timeoutMilliseconds = 100;
```

Hier hat eine bessere Namensgebung uns den Kommentar erspart. Das ist in der Regel oft der Fall.

1.1.2 Namensgebung

Gute Namensgebung ist eine Herausforderung für viele SW-Entwickler. Besonders heikel zum Thema Namensgebung sind Interfaces. Denn diese werden dann oft von dritten Entwicklern oder sogar Systemen verwendet. Und einmal in Systeme eingebunden benötigt es sehr viel Aufwand diese in allen eingebundenen Systemen zu ändern.

Ein guter Name sollte:

- absolut zweckbeschreibend sein
- Fehlinformationen vermeiden

- Unterschiede deutlich machen (differenziert sein)
- gut aussprechbar und gut suchbar sein
- möglichst keine Codierung enthalten

Dazu: Heuristiken N1-N7:

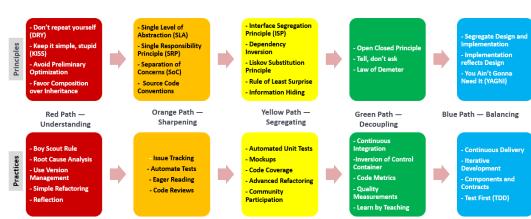
- **N1:** Beschreibende Namen wählen
- **N2:** Namen passend zur Abstraktionsebene wählen
- **N3:** Standardnomenklatur verwenden
- **N4:** Eindeutige Namen wählen
- **N5:** Namenlänge abhängig vom Geltungsbereich
- **N6:** Codierungen vermeiden
- **N7:** Namen sollten auch Nebeneffekte beschreiben

1.1.3 Clean Code Developer

Clean Code Developer ist eine wohldefinierte Auswahl von Prinzipien und Praktiken. Basis ist ein Wertesystem:

- Evolvierbarkeit
- Korrektheit
- Produktionseffizienz
- Reflexion

Die Clean Code Developer Grade



Der Schwarze Grad

Zu wissen, was Clean Code Developer ist, ist der schwarze Grad.

Der zweite Grad (Rot)

- Prinzipien
 - Don't Repeat Yourself (DRY)
 - Keep it simple, stupid (KISS)
 - Vorsicht vor Optimierung
 - Favour Composition over Inheritance (FCoI)
- Praktiken
 - Boyscout Rule
 - Root Cause Analysis (RCA)
 - Ein Versionskontrollsyste einsetzen
 - Einfache Refaktorisierungsmuster anwenden
 - Täglich reflektieren

Der dritte Grad (Orange)

- Prinzipien
 - Single Level of Abstraction (SLA)
 - Single Responsibility Principle (SRP)
 - Separation of Concerns (SoC)
 - Source Code Konventionen: Namensregeln, Kommentare
- Praktiken
 - Issue Tracking
 - Automatisierte Integrationstests
 - Lesen, Lesen, Lesen
 - Reviews

Der vierte Grad (Gelb)

- Prinzipien
 - Interface Segregation Principle (ISP)
 - Dependency Inversion Principle (DIP)
 - Liskov Substitution Principle (LSP)
 - Principle of Least Astonishment
 - Information Hiding Principle (IHP)

- Praktiken
 - Automatisierte Unit Tests
 - Mockups (Testattrappen)
 - Code Coverage Analyse
 - Teilnahme an Fachveranstaltungen
 - Komplexe Refaktorisierungen

Der fünfte Grad (Grün)

- Prinzipien
 - Open Closed Principle (OCP)
 - Tell, don't ask
 - Law of Demeter
- Praktiken
 - Continous Integration (CI) 1
 - Statische Codeanalyse (Metriken)
 - Inversion of Control Container
 - Erfahrung weitergeben
 - Messen von Fehlern

Der fünfte Grad (Blau)

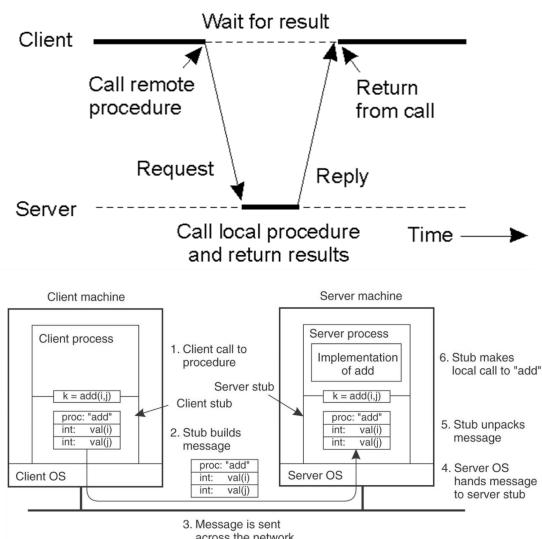
- Prinzipien
 - Implementation spiegelt Entwurf
 - Entwurf und Implemenetation überlappen nicht
 - You Ain't Gonna Need It (YAGNI)
- Praktiken
 - Continous Integration (CI) 2
 - Iterative Entwicklung
 - Komponentenorientierung
 - Test First

Der siebte Grad (Weiss)

- Weisser Grad vereint alle Prinzipien und Praktiken der farbigen Grade.

- Eine gleichschwebende Aufmerksamkeit ist jedoch sehr schwer zu halten. Darum beginnt der Clean Code Developer im Gradesystem nach einiger Zeit wieder von vorne.
- Die zyklische Wiederholung bringt stetige Verbesserung auf der Basis von überschaubaren Schwerpunkten.

1.2 Serverprozesse mittels Sockets und RPC



Vielfach verwendete Konzepte sind: gRPC, DCE, RMI, etc. Was sind die Herausforderungen?

1.2.1 Herausforderungen RPC

Herausforderung 1: Das Kommunikationsprotokoll

Das Kommunikationsprotokoll ist notwendig zur Verständigung zwischen zwei Maschinen (z.B. Client und Server). Die wesentlichen Bestandteile eines Kommunikationsprotokolls sind:

- Festlegung der Anforderungen an den Übertragungskanal. Beispiele: Zuverlässigkeit (Datenverlust möglich), Bandbreite, Latenz
- Definition der generellen Nachrichtenstruktur. Beispiele: Binär/Text, Länge, Stopnzeichen, Nachrichten ID, usw.

- Definition der Nachrichten und Zustände für die Kommunikation.

Zur Nachrichtenstruktur gibt es unterschiedliche Formate die man verfolgen kann. Zum Beispiel entweder mit Längenangabe oder mit Stopnzeichen.

Mit Längenangabe:

Länge (n)	ID	Inhalt
a bytes	b bytes	(n - a - b) bytes

Mit Stopnzeichen:

ID	Inhalt (ohne Stopnzeichen)	Stoppz.
a bytes	b bytes	c bytes

Nachrichten sind auch ein Teil des Kommunikationsprotokolls. Die Struktur folgt praktisch immer folgendem Aufbau:

- **ID:** Identifiziert die Nachricht (z.B. GET, POST, ... bei HTTP). Die ID wird bei strikten Interaktionen zwischen Kommunikationspartnern nicht benötigt. (z.B. Schach).
- **Argumente, oft abhängig von der ID:** Dies sind einfache Datentypen wie Integer, Strings oder Strukturen und Arrays. Sie können zusätzliche Informationen enthalten, basierend auf der Art der Nachricht.

Dies ist ein Beispiel des Aufbaus einer HTTP Nachricht:

HTTP Request:

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

HTTP Response:

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

Herausforderung 2: Parameterübergabe

Ein Beispiel, wie man einen Datensatz (`data`) an eine Liste auf einem entfernten System (`dbList`) anhängt. Als Rückgabe erhalten wir in diesem Beispiel `newlist`.

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

Die Parameterübergabe erfolgt dann

- Lokal:

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

- Remote:

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

Dabei gibt es einiges zu beachten:

- **Kosten:** Die Kosten beziehen sich auf die Ressourcen, die für die Übertragung von Daten benötigt werden. Bei einer Übergabe durch Kopie (by-value) entstehen höhere Netzwerk- und Verarbeitungskosten, da die gesamten Daten übertragen werden müssen. Bei einer Referenzübergabe (by-reference) wären die Kosten niedriger, aber die Umsetzung ist komplexer und stellt besondere Herausforderungen in verteilten Systemen dar.

- **Wo sind Primitive oder Strukturen gespeichert?** Primitive Daten und Strukturen können entweder im lokalen Speicher oder im entfernten Speicher gespeichert sein, abhängig davon, ob die Übergabe lokal oder remote erfolgt. Bei lokaler Übergabe sind die Daten im Speicher des aufrufenden Systems, bei remote Übergabe müssen sie auf das entfernte System übertragen und dort im Speicher abgelegt werden.

- **Wie identifizierte ich Primitive oder Strukturen?** Primitive Datentypen (wie integer, Float, etc.) und Strukturen (wie Arrays, Listen, etc.) werden durch ihre Typdefinitionen in der Programmiersprache identifiziert. Bei der Implementierung von RPC muss klar definiert sein, welche Typen wie übertragen werden sollen, um Missverständnisse und Fehler zu vermeiden.

Herausforderung 3: Parallele Verarbeitung

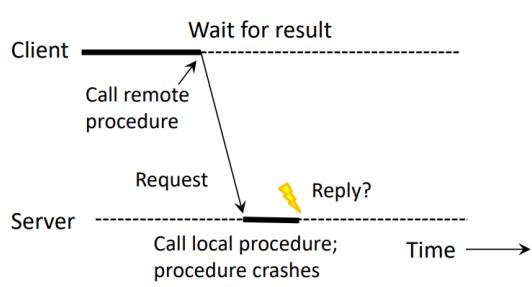
Folgende Varianten zur parallelen Verarbeitung gibt es:

Variante	Einsatzgebiet
– Blocking I/O	Eine langlebige Verbindung oder wenige kurze Verbindungen.
– Single-Threaded	
– Blocking I/O	Wenige langlebige Verbindungen.
– Mehrere Prozesse (Fork)	
– Blocking I/O	Wenige langlebige Verbindungen.
– Mehrere Threads	
– Blocking I/O	Viele kurze Verbindungen.
– Thread-Pools	
– Non-Blocking I/O	Viele Verbindungen (kurz- oder langlebig) mit wenig Bearbeitungszeit.
– Single-Threaded	
– Non-Blocking I/O	Viele Verbindungen (kurz- oder langlebig) mit wenig bis viel Bearbeitungszeit.
– Mehrere Threads	
– Blocking I/O	Viele Verbindungen (kurz- oder langlebig) mit wenig bis viel Bearbeitungszeit.
– Mehrere virtuelle Threads	
(Ab Java 22)	

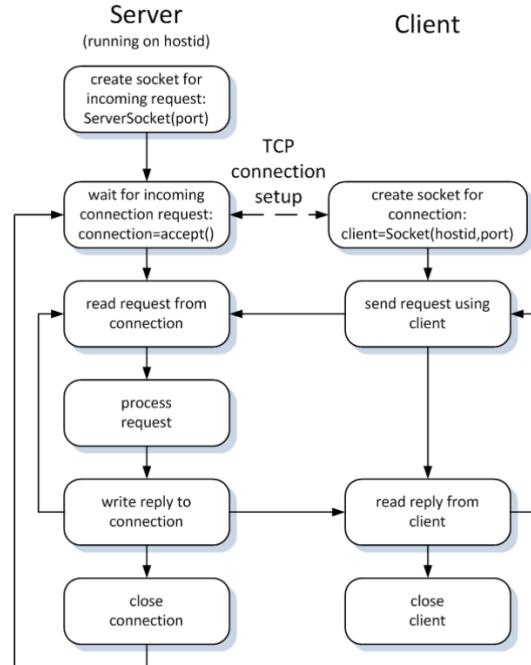
Herausforderung 4: Fehlerbehandlung

Wie erfolgt die Fehlerbehandlung?

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



1.2.2 TCP Client- und Serverprogramme mittels Sockets



Ablauf beim Client

1. Socket erzeugen
2. Socket an einen lokalen Port binden (Serverseitig)
3. Verbindung mit Zieladresse herstellen
4. Daten über Socket lesen / schreiben
5. Socket schliessen

Socket für den Server

Server hören an ihrem zugewiesenen Port auf Anfragen. Sockets bekommen als Argument Portnummer, zu der sich Clients verbinden können. Es gibt einige Regeln für Portnummern:

- Darf nicht in Benutzung sein
- Kann nach Benutzung z.T. für einen bestimmten Zeitraum nicht verwendet werden (typisch: 4 Minuten)

- Falls < 1024 nur nutzbar durch Rootbenutzer bei Unix-Systemen (Linux, MacOS, etc.)

Lebenszyklus eines TCP Servers

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

Einfacher DayTime-Client mit Blocking I/O:

```

1 static void getTime(String host, int port)
2     throws IOException {
3     Socket socket = new Socket(host, port);
4
5     DataInputStream is;
6     is = new DataInputStream(socket.
7         getInputStream());
8
9     byte[] bytes = is.readAllBytes();
10    socket.close();
11
12    String time = new String(bytes);
13    System.out.println(time);
14 }
```

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

```

1 public class SimpleDayTimeServer {
2     public static void main(final String[]
3             args) {
4         try {
5             final ServerSocket listen = new
6                 ServerSocket(1300);
7
8             while(true) {
9                 try (final Socket client =
10                     listen.accept()) {
11                     final DataOutputStream dout
12                         = new
13                             DataOutputStream(
14                             client.getOutputStream(
15                             ));
16                     final Date date = new Date
17                         ());
```

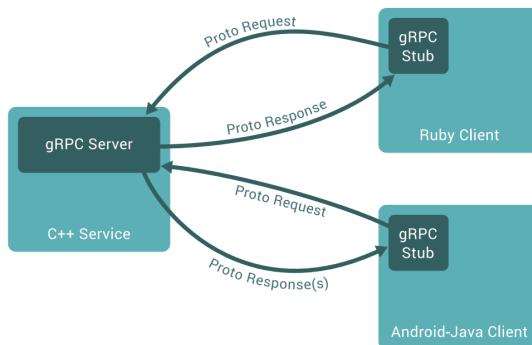
```

10                     dout.write((date.toString()
11                         ).getBytes());
12                 }
13             } catch (IOException ex) {
14                 LOG.debug(ex.getMessage());
15             }
16         }
17     }
```

Für einen nicht blockierenden EchoServer würde man einen **ExecutorService** nutzen um für jeden Client einen eigenen EchoHandler zu erstellen um den Main-Thread frei zu halten um neue Verbindungen zu akzeptieren. Dieser EchoHandler muss dann das **Runnable**-Interface implementieren und der Konstruktor die Socketverbindung zum Client empfangen.

1.2.3 gRPC

gRPC ist ein Plattform- und sprachübergreifendes RPC-Framework. Es ist für Java, C#, Python, C++, Go, Node und weitere Sprachen verfügbar. Es verwendet die Google Protocol Buffers für die Serialisierung und Deserialisierung der Daten und basiert auf HTTP/2.



Der Server-Loop von gRPC führt gleichzeitige Requests als Tasks in verschiedenen Threads aus. Per Default wird ein **CachedThreadPoolExecutor** verwendet. Bei Bedarf kann der **TaskExecutor** mittels **executor(...)** beim Erstellen überschrieben werden:

```
1 ServerBuilder.forPort(PORT).addService(new
2     EchoService()).executor(...).build();
```

2 SW02 - Komponenten, Interfacedesign, SCM und Buildautomatisierung (CI)

2.1 Komponenten

Was ist eine Komponente?

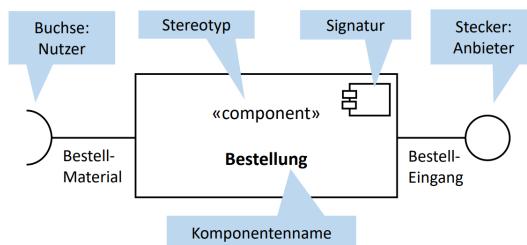
“Eine Software-Komponente ist ein Software-Element, das zu einem bestimmten Komponentenmodell passt und entsprechend einem Composition-Standard ohne Änderungen mit anderen Komponenten verknüpft und ausgeführt werden kann”

- *Council, Heinemann: Component-Based Software Engineering, Addison-Wesley, 2001*

Eigenschaften von Komponenten

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

Modellierung von Software Komponenten in UML:



Um ein Subsystem zu modellieren kann man im UML den Stereotyp **«subsystem»** verwenden. Dessen Komponenten können den Stereotyp **«component»** tragen.

Neben der genauen Form und den Eigenschaften einer Komponente muss das Komponentenmodell einen Interaction-Standard und einen Composition-Standard festlegen.

Interaction-Standard

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

- verteilt oder lokal
- verteilte Objekte, Remote-Procedure-Calls, Unix-Pipes (Streams).
- konkrete Technologien wie SOAP / REST (HTTP).

Der Interaction-Standard legt auch fest, wie innerhalb einer Komponente die Schnittstelle festgelegt wird. Beispiel hierzu wären: Interface Definition Language (CORBA) oder WSDL (SOAP).

Composition-Standard

Beschreibt wie der Entwickler Komponenten zusammensetzt um grössere Einheiten zu bilden und 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.)

2.1.1 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 Komponen zu verwenden: Wettbewerb und Markt.

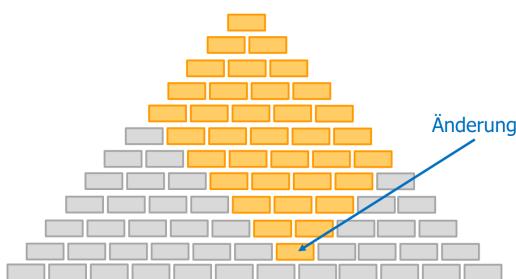
Erbringt vereinbarte Dienstleistung

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

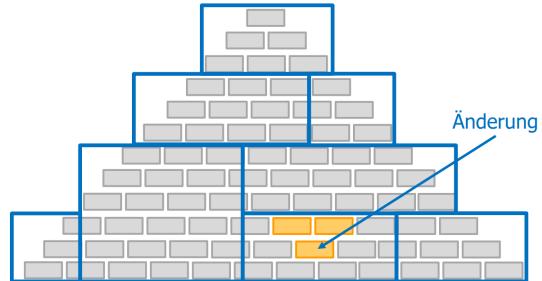
Vollständigkeit

- Komponente als Ganzes ersetzbar.
- Parallele und verteilte Entwicklung.
- Präzise Spezifikation und verwaltete Abhängigkeiten.
- Verbesserte Wartung. Kapselung limitiert Auswirkung von Veränderung.

Auswirkung in einem monolithischen System:



Auswirkung in einem Komponentenbasierten System:



2.1.2 Entwurf mittels Komponenten

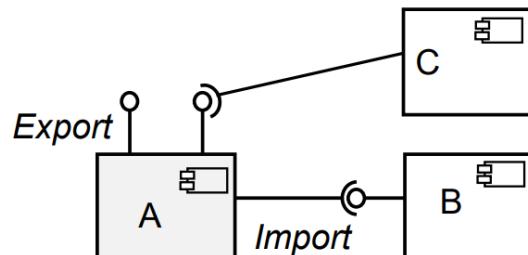
Spezifikation von Komponenten:

Export: unterstützte Interfaces, die andere Komponenten nutzen können.

Import: benötigte / benutzte Interfaces von anderen Komponenten.

Verhalten: Verhalten der Komponente.

Kontext: Rahmenbedingungen im Betrieb der Komponente.



Verhaltenssicht:

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

Components & Connectors: ausführbare Einheiten und gemeinsame Daten.

Datenfluss: Datenfluss zwischen Komponenten.

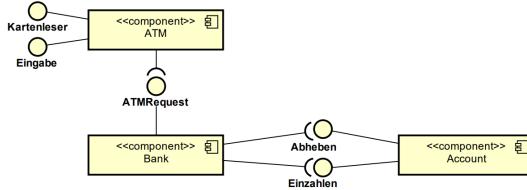
Kontrollfluss: Wird angestoßen von ...

Prozess: Welche Komponenten laufen parallel?

Verteilung: Zuordnung der Komponenten zur HW.

UML-Komponentendiagramm - Verdrahtung:

Hier an einem Beispiel: Bankautomat (ATM)



Sichtbar sind hier Komponenten und Konnektoren.

2.1.3 Begriff und Konzept der Schnittstellen

Schnittstellen: Begriff und Konzept

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

Bedeutung des Schnittstellenkonzepts

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

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

Breite einer Schnittstelle

Die Breite einer Schnittstelle wird bestimmt über

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

Schmälere Schnittstellen haben weniger Abhängigkeiten.

Kriterien für gute Schnittstellen

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

2.1.4 Design by Contract

Das Zusammenspiel der Komponenten wird durch einen Contract erreicht, dieser besteht aus:

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

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

Verantwortlichkeiten bei Design by Contract

	Nutzer	Anbieter
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)

Spezifikation von Schnittstellen

Zur Programmschnittstelle gehört alles, was für die Benutzung der Komponente wichtig ist und was der Programmierer verstehen und beachten muss.

Jede Programmschnittstelle definiert eine Menge von Operationen mit folgenden Eigenschaften:

- Syntax (Rückgabewerte, Argumente, in/out, Typen).
- Semantik (Was bewirkt die Methode)
- Protokoll (z.B. synchron, asynchron)
- Nichtfunktionale Eigenschaften (Performance, RObustheit, Verfügbarkeit, bei Web-Anwendungen möglicherweise auch Kosten).

Wie dokumentiert man Schnittstellen?

- 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.
- Zustandsänderung der Komponenten.
- Spezifikation inwiefern sich Eingabeparameter auf die Zustandsänderung und die Ausgabeparameter auswirken.

In Java werden Schnittstellen mit Java-Interfaces deklariert. Schnittstellen werden (wie Klassen) zu `class`-Dateien kompiliert. Schnittstellen an der Systemgrenze und zwischen Systemen sind architekturelevant und werden in der Architekturbeschreibung dokumentiert. Sonst werden Schnittstellen sinnvollerweise besonders gut mit JavaDoc dokumentiert (woraus eine Dokumentation im HTML-Format erzeugt werden kann).

Öffentliche Schnittstellen werden oft als API (Application Programming Interface) bezeichnet.

2.2 Versionskontrollsysteme (SCM, VCS)

Versionskontrollsysteme (engl. Version Control System, VSC) ist ein Teil des Source Code Managements (SCM).

2.2.1 Grundlagen

Versionskontrollsysteme werden vorwiegend (aber nicht nur) in der Softwareentwicklung genutzt. Alternativ als SCM bezeichnet. Sie sind für eine eher technisch orientierte Nutzung konzipiert. Das Hauptziel ist: **Die Abfolge aller Bearbeitungsschritte an Artefakten benutzerbezogen und detailliert zeitlich nachvollziehen zu können.** Vereinfacht formuliert: "Wer hat wann und aus welchem Grund in welcher Datei welche Änderung vorgenommen?" Dies schliesst Neuerstellung, Verschieben und Löschen mit ein.

Ein VCS erlaubt es jederzeit einen Rückgriff auf alte Änderungsstände zu machen. Außerdem ist es für konkurrenzierende Zugriffe und Modifikationen ausgelegt.

- Teamarbeit auf gemeinsamen Quellen.
- Automatisches Merging bei Konflikten (so weit möglich).
- Entweder zentrale Datenhaltung oder auch verteilt!
- VSC ist aber kein Ersatz für fehlende Koordination.

Dabei gibt es eine ganz klare Abgrenzung zu Filesharing-Diensten.

- Beispiele: Dropbox, OneDrive, etc.
- Synchronisieren Dateien (inkl. Verzeichnistrukturen) zwischen verschiedenen Rechnern und/oder der Cloud.
- Artefakte werden bei jeder Veränderung bzw. bei jedem Speichern sofort übertragen. Jedoch keine Garantie, es kann auch verzögert erfolgen.
- Erzeugen teilweise auch Revisionen (pro Artefakte). Meist stark beschränkte Anzahl an Revisionen.

Wie sollen Versionskontrollsysteme genutzt werden und welchen Fokus haben sie?

- VSCs sind weder ein Backup noch ein Filesharing-Dienst (und auch kein DMS!).
- Bewusster Umgang: Sie als Entwickler bestimmen, wann eine neue Version festgeschrieben wird!
- VSCs haben einen anderen Fokus: Nachvollziehbarkeit von Änderungen. Workflows und Koordination in (verteilten) Teams.
- Änderungen werden als sogenannte **Changegesets** innerhalb einer Transaktion gespeichert. **1..n** Dateiarbeitsfakte, die von einem konsistenten Zustand z_1 zum nächsten konsistenten Zustand z_2 führen.
- Wir entscheiden sehr bewusst, wann wir was versionieren.

2.2.2 Arbeit mit einem VCS

Generell können mehrere Personen gleichzeitig an einem Projekt und somit an der selben Source arbeiten. Jeder Entwickler hat lokal eine Kopie der ganzen Source und kann mit dieser entwickeln.

- **checkout:** Von einem Projekt eine lokale Arbeitskopie erstellen. Auf dieser wird gearbeitet.
- **update:** Änderungen anderer Entwickler in lokaler Arbeitskopie aktualisieren. Periodisch oder nach Bedarf, aber unbedingt vor einem Commit.
- **log:** Bearbeitungsgeschichte der Artefakte ansehen. Also wer hat wann und warum welche Artefakte geändert?
- **diff:** Verschiedene Revisionen miteinander vergleichen. Fremde (oder auch eigene) Änderungen nachvollziehen.
- **commit:** Artefakte in das Repository schreiben (auch: checkin). Lokale Veränderungen in das Repository zurückschreiben. Veränderungen werden dann für die anderen Entwickler nach einem **update** sichtbar.

Simple lokale Versionsverwaltung mit git

Wenn git installiert ist, funktioniert dies auch rein lokal. Dies wird oft vergessen. Lokal wird ebenfalls ein Repository erstellt, welches die Änderungen und Artefakte speichert. Hier ein einfaches Beispiel:

1. **git init:** Erzeugt ein neues Repo.
2. **git add .:** Bezieht (neue) Dateien mit ein.
3. **git commit -a -m "Message":** Bestätigt Änderungen im Repository (Changegeset).
4. **git log:** Zeigt die History des Repos an.
5. **git status:** Zeigt den lokalen Änderungsstatus an.

Tagging - Markieren von Revisionsständen

Tags in Versionskontrollsystmen markieren einen gewissen Stand mit einem Namen. Dieser Name ist eine textuelle, einfach identifizierbare Marke oder Version. Dies erleichtert die spätere Selektion dieses Standes z.B. für einen Release.

Es kann auch genutzt werden um einen Meilenstein, Testversion oder Auslieferungsstand zu markieren. Das Tagging selbst wird von den Systemen sehr unterschiedlich realisiert.

- Nur eine Markierung in jeder Datei (z.B. bei CSV)
- Kopie aller Artefakte in ein anderes Verzeichnis (z.B. bei Subversion)
- Identifikation eines Änderungsstandes des gesamten Dateisystems. (vereinfacht formuliert, z.B. bei git, hg)

Branching

Branches sind voneinander getrennte Entwicklungszweige, lokal oder zentral für:

- Prototypen, Tests und/oder Experimente.
- Bugfixing bereits geschlossener Versionen
- Professionelle (Änderungs-)Workflows

Wenn es sich nicht um Wegwerf-Entwicklungen handelt, ist ein späteres Merging möglich, bzw. notwendig. Dies läuft idealerweise halbautomatisch ab. Kann aber auch sehr aufwändig (manuelles merging) werden.

Was verwaltet man in einem VSC

In einem VSC sollen ausschliesslich Quell-Artefakte eingecheckt werden. Sourcen, Konfigurationen, ggf. Dokumentation. Beispiele für Java: `*.java`, `*.properties`, etc. Es soll aber nicht Artefakte die generiert werden bzw. erzeugt werden können eingecheckt werden (z.B. `*.class`, `./target/**`).

2.2.3 Konzeptionelle Unterschiede

VSCs werden in banalen Vergleichen häufig in einen Topf geworfen, zumal die Befehle zum Teil identisch lauten. Gefährliche Vereinfachung, weil zwar die Ziele identisch, aber die Lösungswege grundverschieden sind.

- Zentrale oder verteilte Systeme
- Optimistische und pessimistische Lockverfahren
- Versionierung auf der Basis einer Datei, der Verzeichnisstruktur (FS) oder der Änderung (changeset).
- Transaktionsunterstützung (vorhanden oder nicht)
- Verschiedene Zugriffsprotokolle und Sicherheitsmechanismen.
- Integration in Webserver (vorhanden oder nicht).

Wenn wir mit einem gemeinsamen Repositoryserver arbeiten, müssen wir das bewusst selber synchronisieren:

- `git clone <url>`: Lokales klonen (kopieren) eines entfernten Repos und Workspace einrichten (einmalig, Initialisierung).
- `git pull`: Änderungen vom entfernten Repo lokal nachführen und in Workspace mergen (fetch/merge).
- `git push`: Lokale Änderungen (im Repo) in entferntes Repo übertragen.

2.2.4 Drei konkrete Produkte

CVS - nicht mehr zeitgemäß

Altes Ur-Versionskontrollsystem. Sehr robust, die Verbreitung hat jedoch stark abgenommen da es als veraltet gilt. CVS basiert auf einem zentralen Server.

Vorteile und Nachteile:

- Vorteile

- Sehr stabil, nur noch sehr wenig Fehler
- Einfache Anwendung, gut überschaubar.
- Repository-Konzept strukturell vorgegeben.

- Nachteile

- Nur dateibasierend (Verzeichnisstruktur nicht versioniert).
- Unterscheidung zwischen Text- und Binärdateien.
- Ablage von Binärdateien sehr ineffizient (Platzintensiv).
- Keine Transaktionen!

Subversion

Wurde offiziell als CVS-Nachfolger eingeführt. Verbreitung mittlerweile sehr stark reduziert, aber noch immer gute und breite Unterstützung. Basiert auf einem zentralen Server.

Vorteile und Nachteile:

- Vorteile

- Transaktionsorientiert (Commit in Transaktion, LUW)
- Versioniert die ganze Verzeichnisstruktur.
- Optimierte/effiziente Speicherung und Übertragung
- Repositorystruktur frei wählbar (für Experten flexibler).
- Integration in / mit Webserver möglich.

- Nachteile

- Repositorystruktur frei wählbar (für Anfänger schwieriger).
- Branching und Tagging technisch eigentlich Kopien/Links.

Git - sehr populäres, zeitgemäßes System

Ist ein verteiltes System, wird u.a. für Linux-Codeverwaltung verwendet. Entwickelt von Linus Torvalds, sehr flexibles Konzept. Ausgelegt für massives, billiges Branching. Operationen möglichst billig und schnell, weil es primär lokal ist.

Vorteile und Nachteile:

- Vorteile

- Verteiltes System, beliebig viele Server / Repos möglich.
- Sehr flexibel, sehr einfach lokal einsetzbar.
- Skaliert (Funktionsumfang, Verteilung, Grösse)
- Meist mit Integration in/mit zusätzlichen Web-Applikationen.

- Nachteile

- Erfordert bei verteiltem Einsatz ein solides Konzept, das organisiert und verstanden werden muss.
- Für Einsteiger anspruchsvoll, weil sehr mächtig.

2.2.5 Benutzerschnittstellen (Clients)

Im wesentlichen gibt es drei verschiedene Varianten:

- Kommandozeile in der Shell (CLI): Ideal für einfache, kurze Befehle (z.B. `git clone`).
- Spezialisierte VCS-GUI-Clients: In der Regel deutlich einfacher zu Bedienen. Unterstützen das spezifische VCS optimal.
- Integrierte VCS-Clients, z.B. in der Entwicklungsumgebung: Sehr bequem, aber je nach Integration etwas verwirrend.

2.3 Buildautomatisation

Wie wird Software typischerweise entwickelt?
Nicht nur das reine arbeiten am Code gehört zur Entwicklung einer Software. Auch folgende Aufgaben gehören dazu:

- Binaries erstellen für Deployment
- Testfälle ausführen, Test-Reports erstellen.
- Qualitätssicherung (Codeanalyse, Metriken, etc.).
- Distributionen/Release erstellen und Deployment.
- Aktualisieren der Dependencies in den Projekten.

Die Gesamtheit all dieser Tätigkeiten, um aus Quellartefakten ein fertiges Produkt zu erstellen, bezeichnet man als **Buildprozess**.

2.3.1 Automatisation des Buildprozesses

Anstelle diese Schritte manuell zu machen, könnte dies ein Script übernehmen.

- Vorteile:
 - Vollautomatisierter Ablauf, keinerlei Interaktion mehr.
 - Jederzeit reproduzierbare Ergebnisse.
 - Lange Builds können über bzw. auch in der Nacht laufen.
 - Vollständige Unabhängigkeit von der Entwicklungsumgebung.
- Nachteil:
 - Eher sturer, unflexibler Ablauf (oder komplizierte Scripte).
 - Abhängigkeit von Shell und Plattform (OS).
 - Aufwändige Wartung und Erweiterung.

2.3.2 Build-Werkzeuge

Build-Werkzeug make

Dieses Build-Tool wird hauptsächlich für C/C++ Projekte verwendet. Es existiert seit vielen Jahren und wird noch immer genutzt. Bietet eine sehr hohe Flexibilität.

Java Build-Tools

Im Java-Ökosystem existiert mittlerweile eine ganze Palette von Tools. Viele davon basieren selber auf Java (mindestens der JRE). Dazu gehören unteranderem:

- Ant: Altes und bewährtes Werkzeug, Java mit XML.
- Maven: populäres, etabliertes Werkzeug, Java mit XML.
- Buildr: neueres Werkzeug, Ruby-Script.
- Gradle: populäres, neues Werkzeug, Groovy-Script mit DSL.
- Bazel: Build-tool von Google, Java mit Python-like Scripts.

Die gemeinsamen Ziele dieser Tools sind:

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

2.3.3 Build-Automation mit Apache Maven

Apache Maven ist selber in Java implementiert und somit plattformunabhängig.

Apache Maven ist deklarativ (nicht imperativ!) implementiert. Das zentrale Element pro Projekt stellt das Project Object Model (POM) dar: `pom.xml`. Dieses definiert alle Metainformationen, Plugins und Dependencies.

Es ist ein bewährtes und robustes Buildtool für Java. Integration ist in praktisch jeder IDE vorhanden. Es sind nur minimale Ressourcen notwendig (nur JDK) und es basiert auf einem globalen, binären Repository.

Apache Maven - Konzept

- Maven selber kann eigentlich sehr wenig. Es definiert im Wesentlichen nur die POM-Struktur und die Lifecycle-Phasen.
- Alles wird dann von dynamisch geladenen Plugins (mit eigenen Releases) erledigt. Extrem grosse Vielfalt an Funktionen.
- Somit ist Maven extrem flexibel und nahezu beliebig erweiterbar.
- Vorischt: Neben den Core-Plugins (die von Apache Maven selber entwickelt wurden), existieren sehr viele weitere (Dritt-)Plugins, deren Qualität sehr unterschiedlich ist. Maven Plugins sind auch Abhängigkeiten.
- Man kann natürlich auch eigene Plugins schreiben. Aber nur wohlüberlegt (Lohnt es sich wirklich?).

Apache Maven - lifecycle phases

Maven definiert einen generalisierten Ablauf eines Buildprozesses, den so genannten lifecycle:

- **validate**: Validiert die Projektdefinition
- **compile**: Kompilation der Quellen.

- **test**: Ausführen der Unit-Tests.
- **package**: Packen der Distributionen (JAR, EAR, etc.)
- **verify**: Ausführen der Integrations-Tests.
- **install**: Deployment im lokalen Repository.
- **deploy**: Deployment im zentralen Repository.

Die per Deklaration aktivierten Plugins registrieren sich und ihre Aufgaben (häufig) automatisch in den jeweiligen Phasen. Buildprozess passt sich quasi dynamisch der Projektart an.

Apache Maven - Binäre Repositories

Maven integriert auch ein Dependency Management. Im POM deklarierte Abhängigkeiten sowohl von Plugins als auch von Libraries werden aus einem zentralen Repository geladen. Alle heruntergeladenen Artefakte werden in einem lokalen Repository auf dem Rechner gespeichert (gecached). Dieses lokale Repository findet man im `$HOME/.m2/repository` Ordner.

Firmen und Organisationen nutzen typisch ein eigenes Repository als lokaler Speicher von eigenen Artefakten.

Apache Maven - Single- und Multimodulprojekte

Apache Maven unterstützt sowohl auch Single- als auch Multimodulprojekte. Ein Single-Modul-Projekt hat eine einfache Struktur für kleine, in sich abgeschlossene Projekte (Kriterium: Release-einheit, d.h. was wollen wir einzeln unter einer Version veröffentlichen können).

Maven unterstützt auch Multimodulprojekte, das heisst ein Projekt kann beliebig viele Submodule enthalten (Modularisierung). Übergeordnete Konfiguration wird an Submodule vererbt. Außerdem können Abhängigkeiten zwischen Submodulen definiert werden.

3 SW03 - Messageorientierte Kommunikation, Dependecy-Management und Buildserver

3.1 Messageorientierte Kommunikation

3.1.1 Grundlagen der messageorientierten Kommunikation

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

Es wird verwendet:

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

Abgrenzung zu:

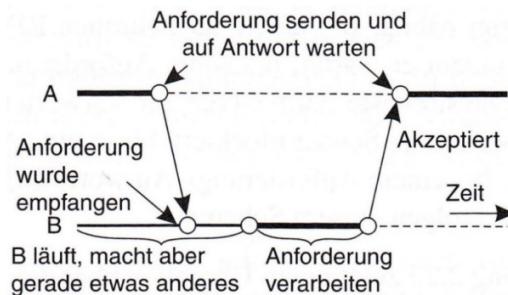
- Synchrone Remote-Procedure-Call
- Streaming (z.B. reines TCP, Unix-Pipes)

Folgende Eigenschaften hat eine messageorientierte Kommunikation:

- Zuverlässigkeit: Je nach verwendetem Protokoll, können Messages verloren gehen.
- Reihenfolge: Ob die Messages in der vorgeesehenen Reihenfolge ankommen, hängt vom Protokoll ab.
- Anzahl Empfänger: Unicast / Anycast (1:1), Multicast / Broadcast (1:n), Client-Server (n:1), Peer-to-Peer(m:n)
- Synchrone vs. Asynchrone: Messageorientierte Kommunikation ist in der Regel asynchron. Der Absender sendet eine Nachricht und fährt fort, ohne auf eine sofortige Antwort (oder auch Bestätigung) zu warten. Es ist aber beides möglich.

- Sicherheit: Nachrichten können unverschlüsselt sein. Sie können auch durch Zugangskontrollen geschützt und verschlüsselt sein um Vertraulichkeit und Integrität zu gewährleisten.

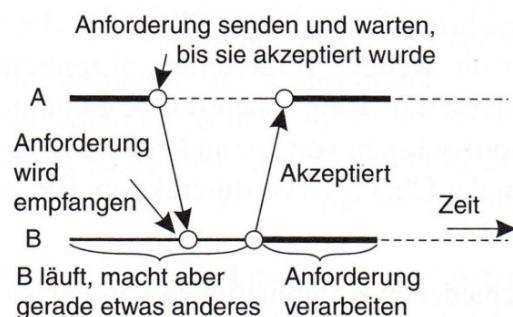
Synchrone Kommunikation



Dies wird zum Beispiel bei Remote-Procedure-Calls oder Anfragen einer Ressource mittels HTTP-Protokoll verwendet.

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

Asynchrone Kommunikation



Auch hier ist der Verwendungszweck derselbe: Remote-Procedure-Calls und Anfragen von Ressourcen per HTTP-Protokoll (hier mit Quittierung, ACK). Nun stellt sich aber die Frage wie gelangt hier das Resultat zurück zum Sender?

- Anforderung senden:** Der Absender (A) sendet eine Anforderung an den Empfänger (B) und wartet bis diese akzeptiert ist, kann aber danach etwas anderes machen.
- Verarbeitung der Anforderung:** Der Empfänger (B) empfängt die Anforderung, akzeptiert sie und beginnt mit der Verarbeitung. In dieser Zeit kann A immernoch andere Aufgaben erledigen.
- Resultat zurücksenden:** Sobald B die Anforderung verarbeitet hat, sendet B das Resultat zurück an A. Dies kann auf verschiedene Arten geschehen:
 - Callback-Funktion:** A übergibt B beim Senden der Anforderung eine Callback-Funktion. Sobald B das Resultat hat, ruft B diese Funktion auf, um das Resultat an A zurückzusenden.
 - Nachrichtenschlange (Message Queue):** B sendet das Resultat an eine vordefinierte Message Queue, die von A überwacht wird. A kann das Resultat dann aus dieser abrufen, sobald es verfügbar ist.
 - Polling:** A fragt regelmässig bei B nach, ob das Resultat verfügbar ist. Dies kann ineffizient sein, wird aber manchmal verwendet.
 - Event-basierte Benachrichtigung:** B sendet ein Ereignis oder eine Nachricht an A, um A zu informieren, dass das Resultat bereit ist. A kann das Resultat dann verarbeiten.
- Der Absender (A) empfängt das Resultat und verarbeitet es weiter. In der Zwischenzeit konnte A an anderen Aufgaben arbeiten und wird nur unterbrochen, um das Resultat zu verarbeiten.

Persistente vs. transiente Kommunikation

Persistente Kommunikation bedeutet, dass eine Nachricht persistent (i.d.R. Disk) gespeichert wird, bis Empfänger bereit ist (z.B. Email).

Transiente Kommunikation bedeutet es dass eine Nachricht nur flüchtig (i.d.R. Memory) gespeichert ist. Nachricht nur verfügbar, solange sendende und empfangende Applikation ausgeführt werden (z.B. Router, Socket).

3.1.2 Kommunikationsmuster

Kommunikation lässt sich in einige Muster einteilen. Bekannte Muster sind:

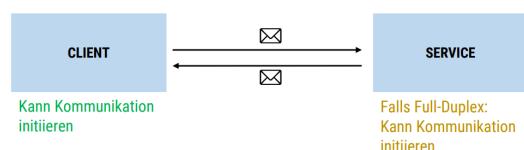
- Request-Reply:** Client-Server (One-To-One)
- Publish-Subscribe:** Datenverteilung (One-To-Many)
- Pipeline:** Verarbeitung in mehreren Schritten (One-To-Many)

Muster: Request-Reply

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

Half-Duplex: Nur Client kann Nachrichten initiieren.

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

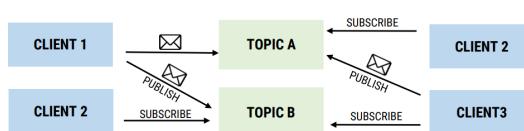


Muster: Publish-Subscribe

Ziel: Datenverteilung

Verbindet eine Menge von Publishers mit einer Menge von Subscribers.

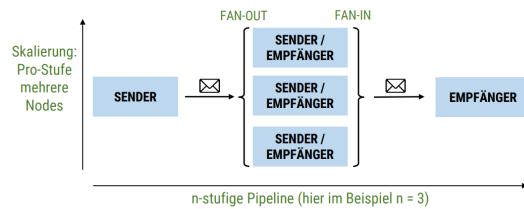
Vertreibt in Enterprise- (ActiveMQ/Web-sphere) und IoT-Umfeld (-> MTTQ).



Muster: Pipeline

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.



3.1.3 Transiente Kommunikation

Am Beispiel von ZeroMQ und WebSockets.

Technologie: ZeroMQ

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



Sockets passend zu Kommunikationsmustern im ZeroMQ:

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

Half-Duplex Beispiel

Für den Client sieht das dann so aus:

```
1 ZMQ.Socket socket = context.createSocket(
  SocketType.REQ);
```

Für den Server der auf die Requests antwortet wäre es dann:

```
1 ZMQ.Socket socket = context.createSocket(
  SocketType.REP);
```

Bei einem Half-Duplex folgt auf jedes `socket.recv()` ein `socket.send(reply)`.

Data Distribution

Für den Publisher sieht das dann so aus:

```
1 ZMQ.Socket socket = context.createSocket(
  SocketType.PUB);
```

Für den Subscriber wäre es:

```
1 ZMQ.Socket socket = context.createSocket(
  SocketType.SUB);
```

Aufgabenverbreitung

Für den Producer sieht das dann so aus:

```
1 ZMQ.Socket socket = context.createSocket(
  SocketType.PUSH);
```

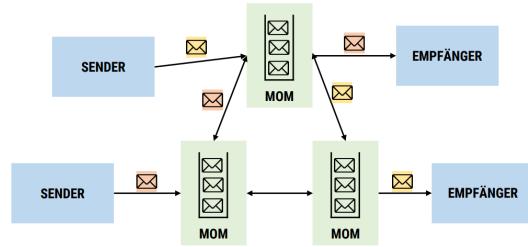
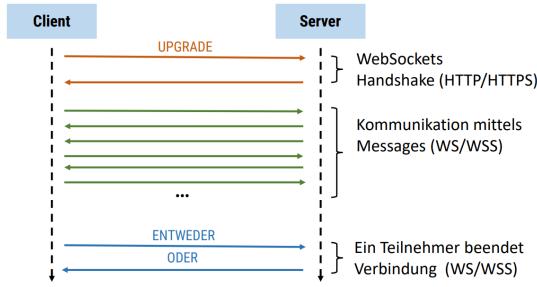
Für den Consumer wäre es:

```
1 ZMQ.Socket socket = context.createSocket(
  SocketType.PULL);
```

Die vollen Codebeispiele sind im Anhang einsehbar.

WebSockets

WebSockets ist eine Full-Duplex Kommunikation. Es ist ein Protokoll auf Application-Level (KEIN Transportprotokoll wie TCP/UDP). Es basiert auf HTTP. Es gibt viele Implementierungen (Tomcat, Jetty, Nginx, Apache, usw.).

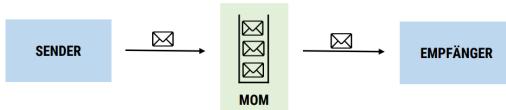


Mit dieser Methode kann man die Kommunikation zeitlich entkoppeln.

3.1.4 Persistente Kommunikation

Am Beispiel von ActiveMQ Artemis

Persistente messageorientierte Kommunikation kann zum Beispiel eine Message-Oriented-Middleware (MOM) verwenden. Diese übernimmt die Zwischenspeicherung von Messages. Dadurch ist die Kommunikation zeitversetzt.



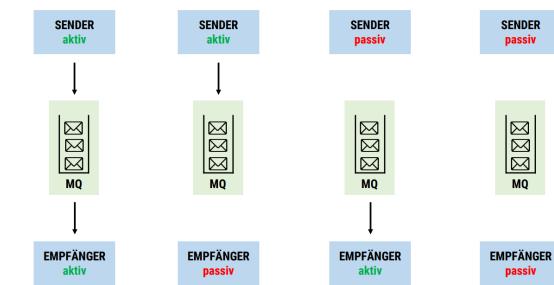
Sender oder Empfänger müssen nicht aktiv sein während der Übertragung. Diese Art von Kommunikation unterstützt Transfers, welche Minuten dauern (anstelle von ms).

Populäre MOMs sind: Apache ActiveMQ / ActiveMQ Artemis, RabbitMQ, Websphere MQ.

Message-Queuing-Modell

Hier findet die Kommunikation mittels Einfügen von Messages in spezifische Warteschlangen statt. Es gibt keine Garantien, wann eine Message gelesen wird. Messages werden von Kommunikationsservern weitergeleitet, bis zum Ziel. Oft sind die Kommunikationsserver direkt miteinander verbunden.

Typischerweise gibt es eine Queue pro Kommunikationsteilnehmer. Eine Queue kann aber geteilt werden.

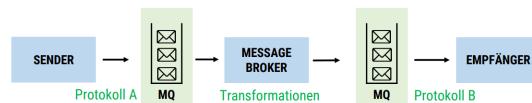


Typische Message-Queuing-Protokolle sind AMQP, MQTT, STOMP, XMPP und OpenWire.

Message-Broker

Das Ziel eines Message-Brokers ist es Systeme mit verschiedenen Protokollen und Nachrichtenformaten zu verbinden.

Alles auf den gemeinsamen Nenner bringen hätte die Komplexität $N \times N$. Ein Broker wandelt zwischen Protokollen und Nachrichtenformaten und routet Messages zu verschiedenen Zielen (ggf. mit Publish-Subscribe).



3.2 Dependency Management

Das Dependency Management (DM) beschreibt die Organisation und Techniken für den Umgang mit Abhängigkeiten zu anderen Modulen.

Modul wird in diesem Kontext vereinfacht als Überbegriff für Package, Library, Bundle oder Komponente verwendet. Häufig wird auch der Begriff **Package Management** verwendet.

Abhängigkeiten können sowohl aus internen und externen Modulen bestehen. Intern: Modul im selben Projekt. Extern: Dritt-Modul, aus anderem Projekt oder Organisation.

Abhängigkeiten werden typisch in binärer (kompilierter) Form aufgelöst. Deshalb kommen dafür so genannte Binär-Repositories und Packagemanager(-Tools) zum Einsatz.

Einige populäre System für DM und PM sind:

- NuGet: Package Manager für .NET-Plattform
- apt: Advanced Packaging Tool, Paketverwaltung für Linux
- yum: Yellowdog Updater Modified, Paketverwaltung für Linux
- npm: Node Package Manager für JavaScript / node.js
- Gems: Paketmanager für Ruby

Allen gemeinsam ist: Zentrale Ablage auf Server, ein standardisiertes Format, zusätzliche Metainformationen, typisch mit Abhängigkeiten (Dependencies) versehen, Sicherung der Konsistenz (z.B. über hash-Mechanismen), geregelte Zugriffsprotokolle, Suchmöglichkeiten etc.

3.2.1 Maven Repository

Es gibt verschiedene öffentliche Repos (OSS). Auf öffentliche Repositories hat man keine Schreibrechte. Organisation betreiben typischerweise interne Repositories.

3.2.2 Deppendency Management für Java

Binäre Module (kompilierte Projekte) werden bei Java typisch als JAR-Dateien (oder EAR, WAR, RAR, JMOD etc.) ausgetauscht. Java selber kennt zwar seit Version 9 neue Mechanismen zur Modularisierung und Definition von Abhängigkeiten, das umfasst aber nicht die Versionierung und die Ablage.

Ursprünglich wurden deshalb die JAR-Dateien einfach von Hand z.B. in /lib-Verzeichnisse direkt in die Projekte kopiert. Dies ist fehleranfällig, da es so schnell zu hoher Redundanz und so auch Platzbedarf kommt.

Grundsätzlich sollte man unterscheiden zwischen:

- Dem **Format** für die zentralen Ablage der meist binären Artefakte mit zusätzlichen Metainformationen im Repository
- Dem **Werkzeug**, das es ermöglicht, Artefakte auf einem Repository zu suchen, zu beziehen, zu deployen und ggf. auch zu verwalten.

3.2.3 Dependency Management mit Apache Maven

Ein Maven Projekt identifiziert sich über drei Attribute, die als *maven coordinates* bezeichnet werden.

- **GroupId:** Meistens zusammengesetzt aus dem *reverse domain name* der Organisation und einem Zusatz. z.B. ch.hslu.vsk
- **ArtifactId:** Entspricht häufig dem Namen des Projektes bzw. den darin enthaltenen Modulen. z.B. stringpersistor-api
- **Version:** Entholten wird eine dreistellige Versionsnummer (`major.minor.patch`. z.B. 3.1.2)

Die Identifikation eines Projektes ist somit eines der wichtigsten Pflichtelemente in einem POM (`pom.xml`). Mittels dieser Koordinaten wird eine Dependency weltweit absolut eindeutig identifiziert werden können (hier: `ch.hslu.vsk:stringpersistor-api:3.1.2`)

Eine einzelne Dependency kann genau so im POM angegeben werden (Im Bereich des Elements: <dependencies>):

```

1 <dependency>
2   <groupId>ch.hslu.vsk</groupId>
3   <artifactId>stringpersistor-api</
4     artifactId>
5   <version>8.0.0</version>
6   <scope>compile</scope>
7 </dependency>
```

Diese werden beim Build automatisch vom Repository (default: Maven Central) heruntergeladen, und im lokalen Repository (\$HOME/.m2/repository) gespeichert.

Der Buildprozess referenziert die Artefakte (typisch: JAR-Dateien) dort mit einem entsprechenden Classpath.

3.2.4 Dependency Scopes

Mit einem Scope wird der Zweck und Geltungsbereich oder Dependency qualifiziert, was unbedingt empfohlen wird.

Maven kennt verschiedene Scopes (hier nur die wichtigstens drei):

- **compile**: Dependency wird für die Kompilation und zur Laufzeit des Programmes benötigt (Default).
- **test**: Dependency wird ausschließlich für die Kompilation und Ausführung der Testfälle (Beispiele: JUnit, AssertJ, etc.) benötigt.
- **runtime**: Dependency nur für Laufzeit, aber nicht für Kompilation. z.B. für dynamisch geladene Implementationen.

IDEs gehen nicht alle perfekt damit um. NetBeans ist derzeit die einzige IDE welche die Scopes nicht nur visualisiert, sondern auch die daraus resultierenden, getrennten Klassenpfade während der Entwicklung aktiv unterhält und nutzt. Dadurch wird absolut zuverlässig vermieden, dass z.B. in einer produktiven Klasse eine Referenz auf eine Klasse aus einer Dependency vom **test**-Scope erstellt werden kann.

3.2.5 Transitive Dependencies

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



Auflösung der Dependencies:

- Modul A ist von Modul B, und dieses von Modul C abhängig.
- Modul A ist somit transitiv auch von Modul C abhängig.
- Bei Kompilation von A wird Modul C somit auch miteinbezogen.

Durch direkte oder transitive Abhängigkeiten können auch Versionskonflikte oder Zyklen auftreten. Maven erkennt solche Konflikte und meldet sie. Einfache Versionskonflikte werden automatisch aufgelöst.

3.2.6 Versionierung und Snapshots

Grundsätzlich sind alle Dependencies versioniert. Auf Basis dieser Semantik kann der Dependency Resolver diverse Automatisierungen und Vereinfachungen anbieten:

- Erkennen von neueren Versionen
- Automatische Verwendung des neusten Bugfixes
- Angabe von Versionsbereichen welche Kompatibel sind, etc.

Gute Repositories sind so konfiguriert, dass eine einmal deployte Version nicht mehr überschrieben werden kann. Das garantiert wirklich nachvollziehbare Buildprozesse.

Semantic Versioning:

- **Major-Release (X.x.x)**: Veränderungen in der API, ind er fachlichen Funktion und/oder in der Konfiguration, welche zu früheren Versionen nicht kompatibel sind.

- **Minor-Release (x.X.x):** Erweiterungen in der API, der fachlichen Funktion oder der Konfiguration, welche aber vollständig Rückwärtskompatibel sind.
- **Major-Release (x.x.X):** Reine Korrekturen oder Änderungen in der Implementierung, voll rückwärtskompatibel, keinerlei neue Funktionen, keine veränderten Funktionen.

Da aber so viel Versionen entstehen, die nie gebraucht werden, wurde das Snapshot Konzept integriert:

Sobald man einer Version den Appendix -SNAPSHOT gibt, gilt diese als erneuerbar und noch nicht stabil, sondern in Entwicklung. Sie wird bei jedem Build immer wieder vom Repository aufgelöst und aktualisiert. z.B. ist so die Version 8.0.1-SNAPSHOT die noch nicht stabile, zukünftige Version 8.0.1.

3.2.7 Managed Dependencies in Multimodul-Projekten

Bei Projekten, die aus mehreren Submodulen bestehen, können mehrere Submodule von der gleichen Dependency abhängig sein. z.B. jedes Submodul verwendet JUnit.

Es macht sehr viel Sinn (bzw. ist es technisch notwendig) dass in jedem Submodul dieselbe Version verwendet wird. Damit man diese Dependencies nicht redundant in jedem Projekt-POM führen muss, sollte man dies in einem übergeordneten (Master-)POM machen.

3.2.8 Deployment

Die häufigste Art von Deployment sind JAR-Dateien. Dass die Quellen und die JavaDoc auch als JAR geliefert werden ist Konvention und kann Unwissende verwirren. Letztlich aber egal, die Einheitlichkeit ist wichtiger.

Vorteil davon für die Entwicklungsumgebungen: Es ist implizit klar, wo die Dokumentation und ggf. der Source für ein bestimmtes JAR gefunden wird.

Das Deployment in öffentliche Repositories wird sehr restriktiv gehandhabt, weil danach

nichts mehr verändert werden darf (auch kein Löschen). Stabilität von Builds muss gewahrt bleiben.

Sehr oft dürfen Entwickler (zurecht) nicht direkt deployen. Man bedient sich stattdessen eines nachvollziehbaren, automatisierten und verifizierbaren Release-Prozesses, welcher von einem Buildserver ausgeführt wird.

3.3 Buildserver

3.3.1 Was ist ein Buildserver

Definition: Ein Buildserver ist eine Serversoftware, die einen automatisierten Build eines Softwareprojekts ausführt und die Ergebnisse allen Entwicklern im Team zur Verfügung stellt.

Build-Auslöser (Triggers) können sein:

- Automatisch durch Änderungen im Versionskontrollsyste
- Automatisch durch Zeitsteuerung
- Manuell durch den Anwender

3.3.2 Positive Effekte eines Buildservers

- Entlastung der Entwickler von repetitiven Aufgaben.
- Häufigere Verifikation durch Buildprozesse, Tests, Deployment.
- Bereitstellung statischer Informationen über den Entwicklungsprozess.
- Automatische Benachrichtigung über den Zustand der Projekte.

3.3.3 Buildserver Beispiele und Dienste

Open Source Beispiele:

- Jenkins/Hudson: Einer der populärsten Buildserver.
- GoCD: Moderne Ansätze mit Pipelines und Continuous Delivery.

Kommerzielle Server Beispiele:

- TeamCity: Funktional und gut skalierbar.

- Bamboo: Eng verknüpft mit JIRA (Issue-Tracking).

Cloud Dienste für Buildserver

Beispiele dafür sind Github, Gitlab und Bitbucket. Der Vorteil dieser ist, dass sie oft kostenlos für Open-Source-Projekte sind.

3.3.4 Konfiguration von Buildservern

Es gibt zwei komplett gegensätzliche Einsätze:

- Variante 1: typischerweise on-site Produkte.
 - Konfiguration getrennt von Projekt.
 - Interaktive Konfiguration direkt auf dem Buildserver.
 - Geschützt Infrastruktur (Buildagents).
- Variante 2: typischerweise Cloud- und Hosting-Plattform und OSS.
 - Konfiguration ist direkt im Projekt abgelegt.
 - Wird direkt durch Entwickler konfiguriert.
 - Weniger restriktiv, sehr häufig mit Docker (ad-hoc Agents).

3.3.5 Einsatz von Buildservern

Als Voraussetzung hat man, dass es automatisiert gebuilded werden kann und das man ein Versionskontrollsysteem hat.

Man sollte auf eine saubere Aufgabentrennung zwischen den Systemen und Technologien achten:

- Wann wird ein Build ausgeführt : Buildserver / Anwender
- Was wird gebaut: Versionskontrollsysteem

- Wie wird gebaut: Buildautomatisation
- Wohin gehen die Artefakte: Buildserver / Binary-Repo

Speziell das **wie** sollte nie vom Buildserver umgesetzt, sondern immer durch den automatisierten Build abgedeckt werden. Durch Buildtools wie z.B. Maven.

Verschiedene Buildarten/-szenarien

- Continous Builds: Automatisch bei Änderungen im Versionskontrollsysteem.
- Nightly Builds: Automatisch nach Zeitsteuerung (nachts). Umfangreiche Builds für zeitintensive Tests und Metriken.
- Release Build: Manuell ausgelöst. Build einer auslieferbaren Version, im VCS getagged.

Integration und Verknüpfung

Integration möglich von:

- verschiedenen Buildtools
- verschiedenen VCSs verschiedenen Kommunikationstechnologien zur Notifikation
- verschiedene Visualisierungen / Plugins für IDEs

Verknüpft wird dies oft mit Issue-Tracking Systemen und Code-Review Werkzeugen.

3.3.6 Gitlab

Im Gitlab werden die Pipelines über die Datei `.gitlab-ci.yml` im Root-Verzeichnis konfiguriert.

4 SW04 - Architekturbeschreibung und Modularisierung

4.1 Architekturbeschreibung

Eine Architekturbeschreibung beschreibt die Umsetzung der funktionalen und nicht-funktionalen Anforderungen, welche an ein System gestellt werden.

Diese Anforderungen (ggf. iterativ) können abgeleitet werden aus übergeordneten Anforderungsdokumenten (Lastenheft) oder übergeordneten Systemen (z.B. Schnittstellen der umgebenden Systeme).

Aleternativ wird eine Architekturbeschreibung auch als Systemspezifikation oder Systembeschreibung genannt.

Nutzen einer Architekturbeschreibung

Es dient als Entwurf und Dekomposition der Architektur. Es ist eine Art Grobdesign der Komponenten. Ausserdem beschreibt es die Kommunikation der Informationen der am Projekt beteiligten Akteure.

Kompatibilität mit agilen Vorgehensmodellen

Eine initiale Version der Architekturbeschreibung wird vor dem Projekt (Sprint 0) definiert. Diese Version bildet die bisher bekannten Anforderungen ab. Jeden Sprint wird dieses Modell dann angepasst. Änderungen der Architekturbeschreibung werden dann im Sprint-Review kommuniziert.

Vorlagen

Es gibt einige Vorlagen, die man auswählen kann. Darunter gehört: arc42, SA4D oder eine firmenteigene Vorlage.

arc42 ist gängiger Standard im deutschsprachigen Raum. Es beantwortet die zentralen Fragen: "Was sollen wir über unsere Architektur kommunizieren / dokumentieren?" und "Wie sollen wir kommunizieren / dokumentieren?".

SA4D hat das Ziel: Beschreibe was nicht offensichtlich aus dem Code erkennbar ist. Diese Vorlage wird kombiniert mit dem C4-Architektur-Visualisierungsmodell.

4.1.1 Inhalt einer Architekturbeschreibung

1 Einführung und Ziele

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

2 Randbedingungen

Das Ziel: Zeige zu entwickelndes System in seinem Kontext (geschäftlich / technisch).

Dies kann beispielsweise erreicht werden, indem man im Diagramm des Gesamtkontexts ein gestricheltes Rechteck um das zu entwickelnde System zeichnet.

3 Kontextabgrenzung

Was dokumentieren: Externe Schnittstellen (sowohl exportierte als auch importierte) und Benutzerschnittstellen.

Wie dokumentieren: Schnittstellen in separatem Dokument beschreiben und in diesem Kapitel exakt referenzieren.

4 Softwarearchitektur

Unterschiedliche Sichten auf ein System aus unterschiedlichen Perspektiven auf hohem Abstraktionsniveau. Zuhilfenahmen von Modellen wie z.B. arc42, 4+1-Sichten-Modell, C4-Modell von Simon Brown.

5, 6 Bausteinsicht und Laufzeitsicht

Diese soll die Funktionalität des Systems auf unterschiedlichen Flughöhen zeigen. Ideal für das ist das C4-Modell.

Grob ist das C4-Modell so aufgebaut:

- Level 1 - Context: Zeigt das System im Verhältnis zu seinen Nutzern und externen Systemen
- Level 2 - Containers: Beschreibt die Hauptbausteine des Systems, wie Anwendungen und Datenbanken, sowie deren Interaktionen

- Level 3 - Components: Detailliert die intern in einem Container vorhandenen Bausteine und deren Beziehungen.
- Level 4 - Code: Geht auf die Implementierungsdetails der Komponenten ein und zeigt, wie der Code strukturiert ist (z.B. Klassendiagramm).

7 Verteilungssicht

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

8 Querschnittliche Konzepte

- 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?
- Teststrategie: Wie wird das System und seine Komponenten getestet?

- Wichtige Schnittstellen: Interne Schnittstellen zwischen Komponenten

9 Entwurfsentscheidungen

- 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?

10 Qualitätsanforderung

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

4.2 Modularisierung und Schichtenarchitektur

4.2.1 Modularisierung: Konzepte und Vorgehen

Begriff: Modul

Ein Modul ist ein in sich abgeschlossener Teil des gesamten Programm-Codes, bestehend aus einer Folge von Verarbeitungsschritten und Datenstrukturen.

Kopplung und Kohäsion



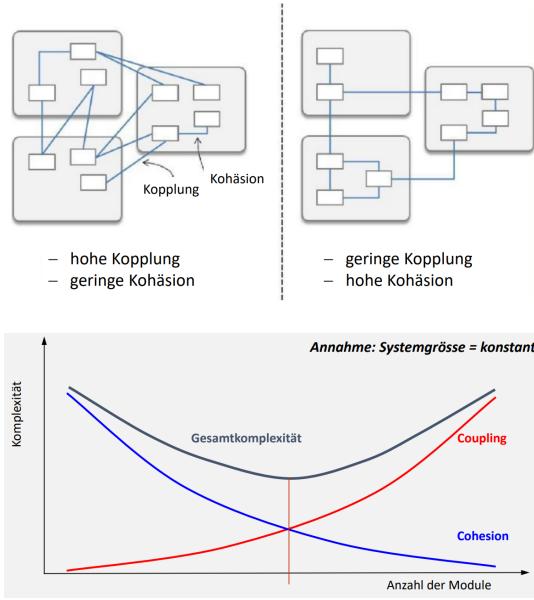
hohe Kopplung

"Module":
– Frachtraum
– Führerkabine



geringe Kopplung

Die Kopplung zwischen Modulen sollte man möglichst minimal halten. Denn so werden sie von einander unabhängig, was die Gesamtqualität des Codes verbessert.



Arten der Kohäsion

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

Arten der Kopplung

- **Laufzeitumgebung, Ausführungszeit:** 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).

Wichtige Kriterien des modularen Entwurfs

- **Zerlegbarkeit (Top-Down):** Teilprobleme sind unabhängig voneinander entwerfbar.
- **Kombinierbarkeit (Bottom-Up):** Module sind unabhängig voneinander (wieder-)verwendbar.
- **Verständlichkeit:** Module sind unabhängig voneinander zu verstehen.
- **Stetigkeit:** Kleine Änderungen der Spezifikation führen nur zu kleinen Änderungen im Code.

Zerlegbarkeit (Top-Down)

Zerlege ein Softwareproblem in eine Anzahl weniger komplexe Teilprobleme und verknüpfe diese so, dass die Teile möglichst unabhängig voneinander bearbeitet werden können.

Die Zerlegung wird häufig rekursiv angewendet: Teilprobleme können so komplex sein, dass sich eine weitere Zerlegung aufdrängt.

Kombinierbarkeit (Bottom-Up)

Strebe möglichst frei kombinierbare Software-Elemente an, die sich auch in einem anderen Umfeld wieder einsetzen lassen.

Kombinierbarkeit und Zerlegbarkeit sind von einander unabhängige Eigenschaften.

Verständlichkeit

Der Quellcode eines Moduls soll auch verstehtbar sein, ohne dass man die anderen Module des Systems kennt.

Softwareunterhalt setzt voraus, dass die Teile eines Systems unabhängig von einander zu verstehen und zu warten sind.

Stetigkeit

Von einer kleinen Änderung der Anforderungen soll auch nur ein kleiner Teil der Module betroffen sein.

Es ist oft unvermeidlich, dass sich im Laufe eines Projektes die Anforderungen ändern. Stetigkeit bedeutet, dass dies nicht die ganze Systemstruktur beeinflusst, sondern sich lediglich auf einzelne Module auswirkt.

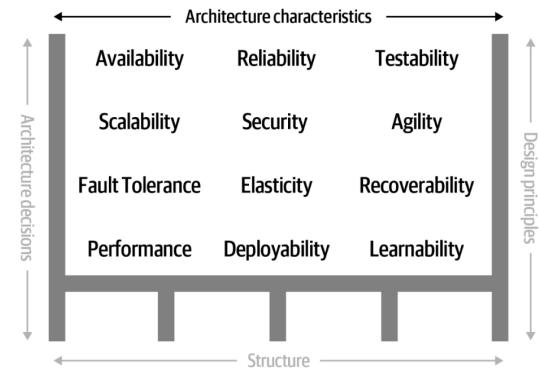
Prinzipien des modularen Entwurfs

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

4.2.2 Schichtenarchitektur

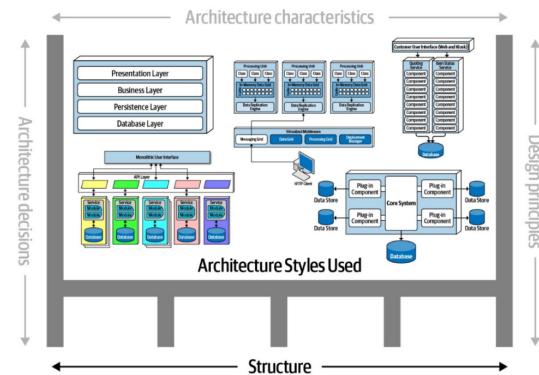
Architektonische Eigenschaften

Architektonische Eigenschaften sind nicht-funktionale Eigenschaften eines Systems, welche zur ordentlichen Funktionsweise notwendig sind.



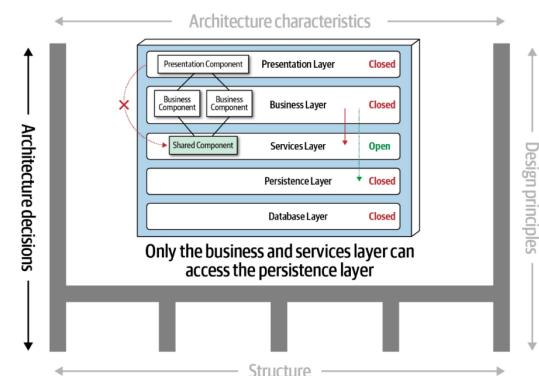
Struktur

Das sind die verwendeten Architekturstile.



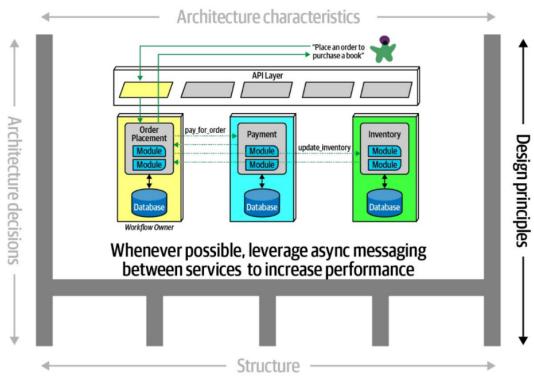
Architektonische Entscheidungen

Regeln, welche Entwickler beim Entwurf der Komponenten befolgen müssen.



Entwurfsprinzipien

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



Was sind Schichten?

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

Schichtenarchitektur

Komponenten werden oft einzelnen Schichten zugeordnet. Eine Schichtenarchitektur ist die Basis für komplexere Architekturen. Dies erlaubt es auch einzelne Schichten zu publizieren und nicht immer die ganze Applikation.

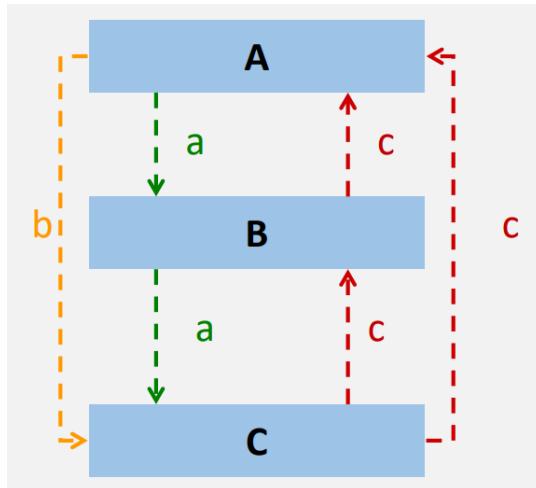
Typische Schichten

- **Presentation Layer:** Darstellung und Benutzerinteraktion mit der Geschäftslogik einer App.
- **API Layer (Services):** Bereitstellung des Zugriffs auf die Geschäftslogik
- **Business Layer:** Geschäftslogik (Funktionalität und Datenstrukturen)
- **Service Layer:** Hilfsfunktionen für Komponenten einer darüberliegenden Schicht.
- **Persistence Layer:** Abstraktion des Datenzugriffs (z.B. möchten wir Kundendaten

oder Kontoinformationen laden und nicht einfach Textfiles).

- **Database Layer:** Zugriff auf den Storage (z.B. Definition des Schemas bei relationalen Datenbanken).

Schichtenbeziehungen: Zulässigkeit



- a: ok.
- b: gefährlich, falls nicht vorgesehen, z.B. Presentation Layer direkt auf API anstelle über Business Logic.
- c: nicht zulässig, da keine zyklischen Abhängigkeiten erlaubt zwischen Schichten.

Offene Schichtenarchitektur

Offene Schichtenarchitektur bedeutet: Offene Schichten können von direkt darüberliegenden Schicht übersprungen werden. In einer offenen Schichtenarchitektur sollte man horizontale Abhängigkeiten vermeiden. Man hätte einen Performance-Gewinn, wenn eine Schicht sonst nur Anfragen weiterreichen würde.

Schichten vs. Tier

Eine Schicht ist eine logische Separierung. Ein Tier ist zusätzlich eine physische Separierung oft kombiniert mit Verteilung (heutzutage oft als Service).

5 Entwicklung mit Container und Docker

Was ist Docker?

Container sind leichtgewichtige virtuelle Umgebungen. Sie sind zwar vollständig isoliert, aber werden trotzdem direkt auf dem Host ausgeführt (schlanker und schneller als VMs).

Die Motivation hinter Docker ist es, eine lauffähige Applikation inklusiver der Laufzeitumgebung in einer standardisierten Form verteilen zu können. Und das alles ohne komplexe Installation von Abhängigkeiten.

Wichtige Konzepte von Docker sind: Container, Images, Volumes, Netzwerke, Registry und Repositories.

Docker ist klar in der Linux-Welt zuhause, kann aber auch auf Windows und Mac genutzt werden.

Funktionsweise von Docker

Docker verwendet einige Techniken aus dem Linux-Kernel (z.B. cgroups und namespaces) um Prozesse und Speicher virtuell vollständig zu trennen (Siehe Unterlagen ITEO). Diese Prozesse laufen isoliert auf demselben OS-Kern, nutzen dasselbe Memory (aber isoliert), können auf ein virtuelles Dateisystem zugreifen und auch (kontrolliert) auf Netzwerkdienste.

Docker läuft seit einiger Zeit auf der Basis von WSL auf Windows Geräten (Windows Subsystem Linux).

Wichtige Docker Begriffe

- **Container:** Ist eine laufende Instanz einer Anwendung in Docker. Ein Container basiert immer auf einem Image.
- **Image:** Enthält die im Docker-Container ausführbare Anwendung in Form eines virtuellen, geschichteten Dateisystems. Images sind read-only und werden in Repositories auf Registries abgelegt. Ein Repository heißt dabei auch ein Image. In einem Repository kann man aber mehrere Versionen eines Images mit Tags definieren (`scope/imagename:tag`).
- **Tags:** Sind Versionslabels für Images

- **Volume:** Virtuelles Dateisystem, das für die persistente Speicherung in Containern genutzt werden kann.

- **Registry:** Ein Zugangspunkt (Server) für eine Anzahl von Repositories für die Verteilung von Docker Images.

- **Repository:** Identifikation einer Gruppe von Images in verschiedenen Versionen (Tags) in einer Registry.

Wichtige Konzepte von Docker

- Docker ist ein typisches CLI-Tool
- Alles in Docker (Container, Images, Volumes) bekommt eine eindeutige ID, welche auf einem hash-Wert basiert.
- Beispiel: Wird ein Container z.B. mit `docker run -d <name>` gestartet, liefert Docker dessen vollständige ID: `dd0b28e28fcffbb5d07567...506a`
- Diese ID ist auch bei einem `docker ps` sichtbar. Dann aber bereits in gekürzter Form: `dd0b28e28fcf`
- Um Docker-Objekte zu identifizieren, müssen jeweils nur so viel von der ID angegeben werden, dass es eindeutig wird. Oft reichen die ersten zwei Ziffern. z.B. `docker kill dd` (für das Beispiel oben).

5.0.1 Einsatzszenarien von Docker für die Entwicklung

- Spontanes, einfaches ausprobieren von Anwendungen, ohne aufwändige Installation.
- Nutzung als Buildumgebung für unterschiedliche, fremde oder auf dem Host gar nicht installierte Plattform.
- Deployment und Betrieb (eigener) produktiver Anwendungen.

- Schnelle und flexible Testumgebung. Integration in eigene Testfälle möglich (Testcontainers).

5.0.2 Erstellen eigener Docker Images

Images werden in Layern erstellt, wobei jeder Layer eine Anzahl Dateien ergänzt oder Befehle ausführt (die Dateien verändern). Images basieren typisch auf existierenden Images.

Die einzelnen Schritte werden über Befehle in einem **Dockerfile** beschrieben, das ist quasi der Quellcode für die Image-Erstellung. Images werden somit reproduzierbar erstellt, vergleichbar mit einem Buildprozess.

Images werden typisch in einer Registry abgelegt, wo sie zur Verwendung zur Verfügung gestellt werden (Deployment). Die Dockerfiles (und alle weiteren Quellartefakte) stellt man typisch unter Versionskontrolle (VCS).

Beispiel eines einfachen Dockerfiles:

```
1 FROM nginx:latest
2 COPY index.html /usr/share/nginx/html
3 EXPOSE 80
4 CMD ["nginx", "-g", "daemon off;"]
```

Der Befehl um das Docker Image zu bauen ist folgender:

```
1 docker build -t "name:tag" <
    path_to_dockerfile>
```

Der Befehl um einen Container damit zu starten ist:

```
1 docker run -d -p "8081:80" name:tag
```

Dies startet einen Container im **detached**-Modus. Also werden wir nicht gleich in der Shell mit dem Container verbunden sondern er startet im Hintergrund. Außerdem machen wir hier ein Port-Mapping (8081:80), welches die Kommunikation von unserem lokalen Port 8081 auf den Containerport 80 weiterleitet. So können wir im Browser <http://localhost:8081> aufrufen und kommen auf die Website des **nginx** im Container.

Auch können wir anstelle des **docker run**-Befehls ein Docker Compose erstellen, dies ist jedoch nicht Teil dieses Moduls.

5.0.3 Integration Tests mit Containern

Dies wird in einem späteren Kapitel genauer behandelt.

5.0.4 Tool-Tipps für Docker

- **ctop**: Shell-Tool für die permanente Anzeige aller Container: <https://github.com/bcicen/ctop>
- **dive**: Shell-Tool für die Analyse von Images, praktisch zur Fehlersuche: <https://github.com/wagoodman/dive>

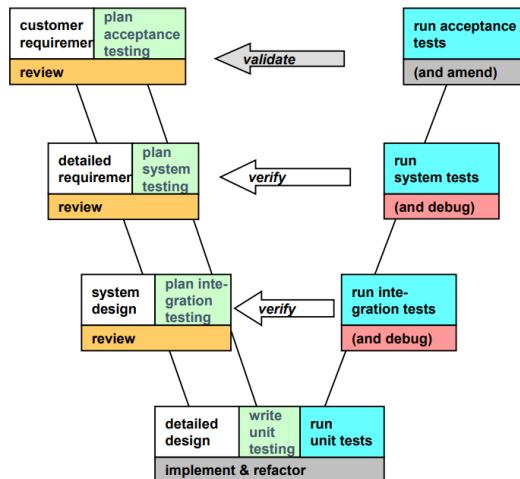
6 SW06 - Integraion- und Systemtests, Automatisiertes Testing und Test Doubles

6.1 Integrations- und Systemtests

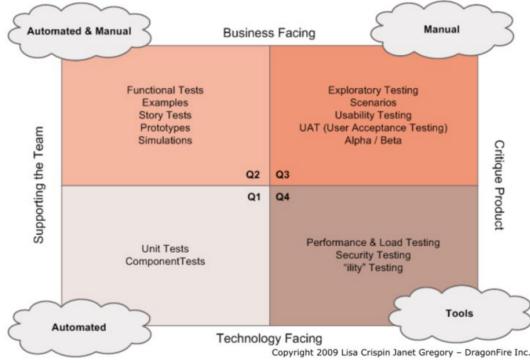
Das Testen muss systematisch ablaufen. Alles andere ist Zeitverschwendungen. Nur dokumentierte oder automatisierte Tests lassen sich wiederholen. Nur so ist die Codefunktionalität messbar.

Klassisches Testen mit dem V-Modell

Anforderungen und Spezifikationen sind Grundlage für weitere Arbeiten. Validieren (haben wir das Richtige entwickelt?) und verifizieren (haben wir es richtig gemacht?).



Agiles Testen



6.1.1 Teststrategie

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

Festlegung der Teststrategie

Eigenschaften einer soliden Teststrategie:

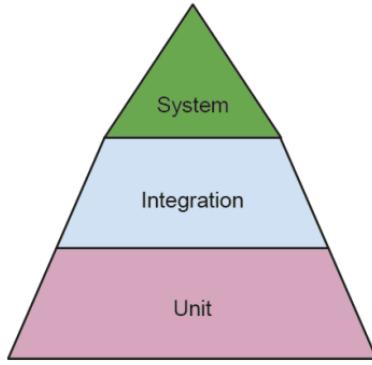
- Fachgerecht:** Passende Testziele, Testmethode, Testarten.
- Risikoorientiert:** Nicht alle Systeme bzw. Systemkomponenten sind gleich kritisch. Abdeckung entsprechend anpassen.
- Wirtschaftlich:** In der Regel beschränktes Testbudget

Qualitätskriterien des Produkts



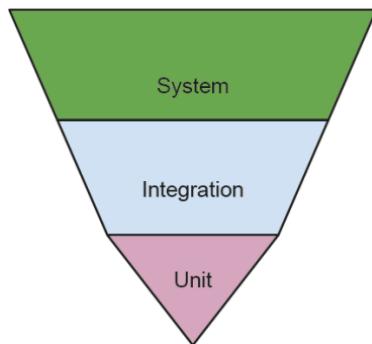
Hierarchieebenen

Die klassische agile Testpyramide:



Idee: Integrations- und Systemtests sind aufwändig, da schwer automatisierbar. Unitests stellen die Basis der Test-Strategie dar.

Alternative Testpyramide:



Idee: Mittels Tools und Virtualisierung sind Integrations- und Systemtests gut automatisierbar (Testcontainer). Unitests nur bei besonders kritischen Funktionalitäten einsetzen.

Teststrategie: Vorgehen

- Pro Hierarchieebene ermitteln, ob getestet werden soll. Kriterium ist immer Aufwand (Entwickler und Tester) gegenüber Ertrag (Reduktion des Fehlerrisikos). Wichtig: Auf Systemebene wird immer getestet.
- Wie soll getestet werden: Einzelne Module (Units) und/oder mehrere Module im Verbund (Integrationstests)? Unitests oft aufwändiger zu erstellen, aber selektiver (zielgerichteter, schnellere Ausführung, schnelleres Fehlerfinden).

- Welche Qualitätskriterien sollen getestet werden? Immer Funktionalität. Soll zusätzlich noch Benutzbarkeit, Zuverlässigkeit, Effizienz, Wartbarkeit oder Übertragbarkeit getestet werden?
- Soll automatisiert werden? Wie oft wird der Test ausgeführt?

6.1.2 Integrationstests

Integrationstests testen folgendes:

- **Schnittstellen (direkte Abhängigkeiten)**: Objektkompatibilität, Aufrufsequenzen, Inputvalidierung
- **Datenabhängigkeiten (indirekte Abhängigkeiten)**: Pro Komponente Datenabhängigkeiten ermitteln und testen. Gemeinsam genutzte Ressourcen (z.B. Dateien, Shared-Memory, Datenbanktabellen) besonders gründlich testen.
- **Abdeckung des Call-Graphs**: Bei Komponenten, die von verschiedenen Aufrufern genutzt werden, auch alle Aufrufvarianten testen.

Hilfsmittel für Integrationstests: Äquivalenzklassenanalyse

Man versucht äquivalente Inputwerte zusammenzufassen, da i.d.R. nicht alle Kombinationen von Methodenparametern getestet werden können. Zum Beispiel fasst man alle positiven Werte zusammen, alle maximalen Werte, usw.

Beispiel: `sendMessage(String message)`

Hier können nun folgende Äquivalenzgruppen gemacht werden:

- `null`
- `“”`
- `“x”`
- `“Dies ist ein Fehler”`
- `“lange meldung ...”` (max. Anzahl von Zeichen).

Integrationstestfälle entwerfen: Step-By-Step

1. **Wechselwirkung analysieren:** Welche Operationen von B ruft A auf? Diese Aufrufe sind zu testen.
2. **Parametersätze der Aufrufe von B ermitteln:** ggf. mittels Äquivalenzklassenanalyse
3. **Testfallinputs ermitteln:** Welche Aufrufsequenzen von A sind notwendig, um im Aufruf von B die in Schritt 2 ermittelten Parametersätze auszulösen
4. **Soll-Ist-Vergleich:** Wie kann die Sollreaktion von B ermittelt werden? Direkt aus dem Output von A? Oder muss der Übertragungsweg mitgeschnitten werden?

Hilfsmittel für Tests: Stellvertreter

Um ein Testobjekt zu testen, müssten alle Komponenten, die das Testobjekt benötigt, in der Testumgebung installiert und im Testlauf mit verwendet werden können.

Es gibt aber die Möglichkeit fehlende Komponenten durch Stellvertreter (z.B. Mock-Objekte oder Simulatoren) zu ersetzen.

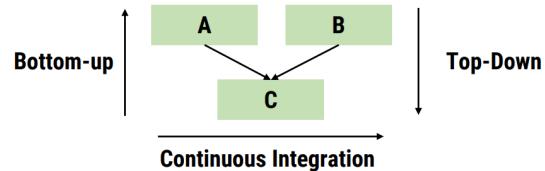
Der Einsatz von Stellvertretern erlaubt auch selektivere Tests zu erstellen.

Bei inkrementeller Entwicklung ist es eher die Regel als die Ausnahme, dass nicht alle benötigten Komponenten fertig bzw. vorhanden sind.

Exkurs: Integrationsstrategien

- **Continuous Integration:** Bei iterativer Entwicklung neu dazu gekommenes laufend integrieren und testen. Dies bedingt agiles Entwicklungsvorgehen.
- **Bottom-Up:** Kleinere Teilsysteme lassen sich bottom-up integrieren.
- **Top-Down:** Bei aufwändigen Kontrollstrukturen mit Hilfe von Stellvertretern top-down vorgehen und dann die richtigen Komponenten integrieren.

- **Big-Bang:** Was für den weiteren Testablauf benötigt wird in einem Aufwisch zusammenführen. Wird i.d.R nicht gelingen.



6.1.3 Systemtest

Systemtests prüfen die gesamte Wirkungskette im Softwareprodukt, also Aspekte, die mit Unitests und Integrationstests nicht abgedeckt werden.

Potenziell lieferbare Softwareprodukte müssen:

- ausserhalb der Entwicklungsumgebung lauffähig sein
- über eine Bedienschnittstelle verfügen
- mit anderen Applikationen und Systemen interagieren

Für Systemtests sind Testfälle notwendig, welche in einer Testumgebung ablaufen, welche der späteren Einsatzumgebung möglichst nahe kommt.

Herleitung der Systemtestfälle

Systemtestfälle können hergeleitet werden aus:

- Akzeptanzkriterien (Scrum)
- Anforderungen (funktional / nicht-funktional)
- Use-Case-Beschreibung

Hinweis: Nicht-funktionale Anforderungen sind auch wichtig zu testen: Last-, Performance-, Stress-, Security-, Robustness-Tests.

Test-First-Ansatz: Formulieren der Systemtests fördert das Verständnis der Anforderungen und bringt Unklarheiten und Inkonistenzen frühzeitig zu Tage.

Regressionstests

Bereits realisierte und getestete Features müssen nach jeder Änderung / Erweiterung der Software erneut getestet werden. Somit häufen sich die Tests in einem Software-Projekt über Zeit an.

Dokumentation der Systemtests

Systemtests werden als Regressionstest auch in weiteren Entwicklungsschritten immer wieder gebraucht und genutzt. Damit Systemtests wiederholbar sind, müssen deren Spezifikation und Ausführung nachvollziehbar dokumentiert werden. Wichtige Bestandteile der Beschreibung eines Testfalls (Testdrehbuch):

- die Vorbedingungen für die Testausführung
- die Handlungen und Eingaben für die Durchführung des Tests
- die erwarteten Ergebnisse und Nachbedingungen

Tests von Benutzerschnittstellen

Auch Benutzerschnittstellen müssen getestet werden. Dies kann auch mit Tools wie z.B. Selenium oder Cypress automatisiert werden.

Entkopplung der Testumgebung

Beispiel: Timeouts

Die Tests werden so gestaltet, dass sie unabhängig von der spezifischen Testumgebung ausgeführt werden können. Dies bedeutet, dass Variablen wie Zeitlimits (Timeouts) konfiguriert und angepasst werden können, ohne die Testlogik zu beeinflussen.

Entkopplung der Testschnittstelle

Beispiel: Verschiedene Oberflächen

Die Testfälle sind so strukturiert, dass sie unabhängig von der Art der Benutzerschnittstelle (GUI, API, etc.) sind. Dies ermöglicht es, die Tests wiederzuverwenden, auch wenn sich die Oberfläche ändert.

6.1.4 Testing in Scrum

Ein Sprintziel muss es sein, die Fertigstellung von Features mit Hilfe von Akzeptanzkriterien zu prüfen.

Konsequenz davon ist, dass im Taskboard zu den in diesem Sprint geplanten User-Stories auch die erforderlichen Ingrationstests, Systemtests und Abnahmetests eingeplant werden müssen.

Natürlich müssen auch die jedes Mal zunehmenden Regressionstests eingeplant werden.

Testaufgaben im Scrum Team

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

6.2 Automatisiertes Testing

Bei vielen Entwicklern ist das Testing unbeliebt. Warum?

- Oft aus Übermut. Man denkt man macht schon keine Fehler.
- Man möchte programmieren, nicht testen.
- Aus Zeitgründen.

Das Testen gilt oft als nicht wichtig bei Entwicklern oder dem Management. Tests aber gewährleisten, dass (speziell auch wichtige) Software möglichst fehlerfrei arbeitet. z.B. medizinische Geräte, Verkehrsmittel, Atomkraftwerke, etc.

Motivation für Testen

Anstelle zu testen um Fehler zu finden, sollte man kontinuierlich während der Implementation testen, um die Gewissheit zu haben, dass es funktioniert.

Am besten findet man Fehler bevor man sie gemacht hat und korrigiert sie bevor man sie implementiert hat. Wie macht man das? - mit Test First.

Test First Methodik

Ganz einfacher Ansatz: Vor der Implementati-on immer zuerst die Testfälle schreiben. Dadurch folgen viele positive Effekte:

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

6.2.1 Unit Tests

Unit Tests werden häufig mit Komponenten-, Modul- und Entwicklertests gleich gesetzt. Sie sind funktionale Tests von einzelnen, in sich abgeschlos-senen Units (typisch Klasse, aber auch Komponen-te oder Modul). Ziele von guten Unit Tests:

- Schnell und einfach ausführbar, selbstvali-dierend (mit `assert`-Methoden) und auto-matisiert.
- Möglichst ohne Abhängigkeiten zu anderen Klassen, Komponenten oder Modulen (lose Kopplung).
- Werden während der Entwicklung geschrie-ben und ausgeführt.

Es gibt gute Unterstützung für Unit Tests durch Frameworks wie (für Java) mit z.B. JUnit, TestNG, etc.

Pros und Cons

- Positiv:
 - Neue oder veränderte Komponenten können sehr schnell getestet werden (re-gressiv).
 - Testen ist vollständig in die Implemen-tationsphase integriert.
 - Test First Ansatz möglich.
 - Automatisiertes, übersichtliches Feed-back / Reporting.
 - Messung von Codeabdeckung kann in-tegriert werden.
- Negativ:
 - Für GUI(-Komponenten) etwas auf-wändiger.
 - Qualität und Nachvollziehbarkeit der Testfälle muss im Auge behalten wer-den: Qualität vor Quantität!
 - In manchen Architekturen / Umgebun-gen schwierig umsetzbar.

6.2.2 Integrationstests mit JUnit

JUnit (und andere Frameworks) können zur Auto-matisierung (fast) aller Testarten verwendet wer-den! Für Integrationstests existiert JUnit (und Apache Maven) eine eigene Namenskonvention:

- Klassenname `XyzIT` für Integrationstests.
- Werden auch unter `src/test/java` abge-legt.

Die Unterscheidung ergibt sich nur durch den Zeitpunkt der Ausführung, und deren Laufzeitabhängigkeiten. Integrationstests können mit Apache Maven mit der eigenen Stage `integration-test` bzw. `verify` ausgeführt wer-den. Getrennte Plugins `surefire` und `failsafe` weisen die Testresultate auch getrennt (Unit und Integration) aus.

Abgrenzung: Unit vs. Integration Tests

Wird oft individuell festgelegt und kontrovers diskutiert. Unit Tests sind wirklich Unit Tests, wenn sie (unter anderem) auf einem beliebigen System und jederzeit lauffähig sind und (bei Java) auch auf unterschiedlichen Betriebssystemen laufen.

Konsequenz: Testfälle welche z.B. mit dem Dateisystem interagieren, sind in strenger Sichtweise bereits Integrationstests. Testfälle die z.B. Sockets verwenden (auch wenn nur auf localhost sind bereits Integrationstests).

Unit Tests sollten somit nie aufgrund von Fremdeinflüssen fehlschlagen!

6.2.3 Messung der Code Coverage

Code Coverage ist eine Metrik, welche zur Laufzeit misst, welche Quellcodezeilen ausgeführt wurden. Diese Messung erfolgt typisch während der Ausführung der Testfälle.

Somit kann eine Aussage gemacht werden, wie umfassend der Code tatsächlich genutzt bzw. getestet wurde.

Ausserdem ist eine umfangreiche, statistische Aufbereitung der Daten möglich:

- Nach Testfall, Komponente, Package, Teilsystem, etc.
- Auf Buildserver z.B. auch historisiert, d.h. zeitliche Veränderung der Werte wird sichtbar gemacht.

Eine hohe Coverage ist kein Beweis für gute Testfälle oder gar Fehlerfreiheit des Codes!

Was kann man messen?

- **Statement Coverage (Line Coverage):** Misst ob (und wie häufig) eine Codezeile durchlaufen wurde. Problem hierbei ist: Handelt es sich bei der Zeile z.B. um einen logischen Vergleich/Ausdruck, ist ein einmaliger Durchlauf nicht repräsentativ.
 - **Branch Coverage:** Prüft, dass alle Zweige einer Anweisung ausgeführt werden.
- **Decision Coverage:** Bei Fallunterscheidungen (z.B. in einem if-Statement) wird geprüft, dass alle Teilausdrücke in der Bedingung auf **true** und **false** aufgelöst wurden (strenger als Branch Coverage).
 - **Path Coverage:** Bei der Path Coverage wird gemessen, ob alle möglichen Kombinationen von Programmablaufpfaden durchlaufen wurden. Problem: Die Anzahl der Möglichkeiten steigt exponentiell mit der Anzahl Entscheidungen. Dies ist in der Praxis nicht durchführbar.
 - **Function Coverage:** Misst auf der Basis der Funktionen ob sie aufgerufen wurden.
 - **Race Coverage:** Konzentriert sich auf Codestellen die parallel ablaufen.

Diese Messwerte sind unterschiedlich Aufwändig und Aussagekräftig.

Coverage - Technische Umsetzung

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

Wer misst und wann?

Gemessen wird die Abdeckung bei der Ausführung des Codes durch den (modifizierten) Code selber. Das Coverage-Werkzeug instrumentiert den Code.

Ein populäres Coverage Werkzeug für Java ist JaCoCo. Bestandteil von EclEmma, welches zu Eclipse gehört. Es ist ein Buildtool, die IDE oder der Buildserver werten die Resultate nur aus und stellen diese dann entsprechend dar.

6.2.4 Dependency Injection

Oft stellen Entwickler fest, dass sich selbst einfache Klassen oder Komponenten schlecht testen lassen. Das führt sehr schnell zum Verzicht auf Unit-Tests oder es werden wieder vermehrt Integrationstests implementiert.

Bei näherer Betrachtung sind oft zu viele bzw. die zu stark gekoppelten Abhängigkeiten die Ursache. Deren negative Auswirkungen fallen bei der ersten Anwendung (und das sind die Testfälle) sehr schnell auf.

Die eigentliche Ursache ist somit schlicht schlechtes Design. Darum: Lässt sich eine Softwareeinheit nur schlecht (kompliziert und/oder aufwändig) testen, sollte man das Design immer kritisch hinterfragen.

Lösung: Dependency Injection (DI)

Um Abhängigkeiten aus dem Weg zu räumen um den Code testen zu können, verwenden wir Dependency Injection. Ein einfaches Beispiel ist z.B. ein überladener Konstruktor, um die Implementation der Abhängigkeit austauschen zu können für den Test. Dazu muss man im Konstruktor den konkreten Typ durch ein Interface ersetzen.

Vorteile beim Einsatz von Dependency Injection

Man ersetzt den konkreten Typ durch ein Interface, womit die Kopplung stark abnimmt. Dadurch können auch verschiedene, alternative Implementierungen genutzt werden. Es resultiert eine bessere **Separation of Concerns (SoC)**.

Das Beste: Die Testbarkeit wird dadurch massiv vereinfacht. Man kann während der Tests eine alternative Implementation als Platzhalter (**Test**

Double) einfügen. Integrationstests werden somit wieder zu Unit Tests. Es resulieren schnellere und selektivere Tests.

6.2.5 Effektives Testen mit Test Doubles

Nachteile von Fake-Implementierungen beim Testen

So elegant die Idee ist, so hat sie auch grosse Nachteile: Je besser und umfangreicher man testet, desto grösser wird die Anzahl von unterschiedlichen Fake-Implementierungen. Unterhalt dieser Fake-Implementierungen erhöht den aufwand. Auch leidet die Übersichtlichkeit.

Die Lösung: Man kann während der Tests eine dynamisch zur Laufzeit erstellte Implementation als Platzhalter (Test Double) einfügen. So haben wir eben keine persistenten Fake-Implementierungen.

6.2.6 (Integrations-)Testen mit Containern

Selbst wenn Integrations-Tests eigentlich automatisiert sind, stellen sie uns vor grosse Probleme: Die nötigen "Umsysteme" müssen meist in mühsamer Arbeit manuell installiert und konfiguriert werden.

- z.B. Datenbankserver, Applikationsserver, Webserver, etc.
- viel Handarbeit, fragil, grosse Fehleranfälligkeit
- Testdatenmanagement: Bei manchen Projekten ein sehr wichtiges (und häufig unterschätztes) Thema!
- Dokumentationsaufwand, Versionierung.

Eine gute Lösung dafür ist Docker und Test-containers.

Integrations-Tests mit Docker-Containern

Die Grundidee: Wir bauen für die Tests individuelle Container, welche schnell und einfach konfiguriert, hochgefahren und danach ohne Spuren

wieder abgeräumt werden können. Noch dazu: Das geht alles schnell (vgl. virtuelle Maschine).

Testcontainers ist ein JUnit ergänzendes Framework, welches:

- Eine Java-API zu vielen Docker-Aktionen anbietet.
- Vorbereitete Images (als Klassen) zur Verfügung stellt.
- Das Hoch- und Runterfahren weitgehend automatisiert.
- Sich automatisch um das Portmapping kümmert.

Für maximale Flexibilität können eigene Images sogar innerhalb eines Textfalles (ad-hoc) mit einer sehr eleganten Fluent-API erstellt und konfiguriert werden.

Herausforderungen von Testcontainern

Das Konzept ist faszinierend und hat eine grosse Mächtigkeit. Es stellt aber auch grosse Ansprüche an die Infrastruktur:

- Images müssen gespeichert werden, bzw. sollten gezielt verwaltet (und auch gelöscht) werden.
- Eigene Registries und Repositories werden unverzichtbar.
- Was vorher individuell auf einem Rechner installiert wurde, zentralisiert sich jetzt z.B. auf einer Buildinfrastruktur

Docker-in-Docker (dind) ist möglich, muss aber explizit berücksichtigt und vorgesehen werden.

Benötigt je nach Build(-server)umgebung noch zusätzliche Konfigurationen.

6.3 Testing: Test Doubles

Ein Test Double ist ein Platzhalter für eine echte, produktive Implementation während der Tests (also Mock-Implementationen). Häufig spricht man

unpräzis nur von Mocks und Mocking. Der korrekte Oberbegriff ist **Test Double**, davon gibt es dann verschiedene, interessante Spezialisierungen.

Warum Test Doubles?

- Test Doubles dienen hauptsächlich dazu den Aufwand für Integrationstests zu reduzieren, indem man stattdessen mehr Testfälle als einfache Unit Tests realisieren kann.
- Man will so viel wie möglich mit Unit Tests prüfen, weil dies die erste Teststufe direkt bei den Entwicklern ist. Sie ist schnell, man kann sie häufig ausführen, sie sind überall lauffähig und vollständig automatisiert.
- Test Doubles können aber auch innerhalb von Integrationstests sehr nützlich sein. Gezielte Isolation der Tests von einzelnen Integrationen.

Anforderungen für Testen mit Test Doubles

Damit das Testen mit Test Doubles gelingt, muss ein gutes Design vorliegen, das dies erlaubt. Gutes Testen und gutes Design unterstützen sich gegenseitig.

Der Einsatz von Interfaces lohnt sich fast immer. Eine Schnittstelle lässt verschiedene Implementierungen zu.

Die Wahl der gewünschten Implementationen muss zur (Test-)Laufzeit beeinflusst werden können. Per **Dependency Injection** manuell oder mit einem Framework.

Übersicht der Test-Doubles

Es gibt einige unterschiedliche Test-Doubles: Dummy, Stub, Spy, Mock und Fake.

Test-Doubles: Dummy

Dies sind sehr primitive und häufig leere Ersatz-Implementierungen, die als aktueller Parameter an Methoden übergeben wird. Dies wird angewendet, wenn der Parameter zwar notwendig ist, dessen Nutzung und Implementation für den Test aber irrelevant.

Dummy dient zur meist funktionslosen Entkopplung der beim Test unerwünschten Abhängigkeiten.

Beispiel: Einem Objekt muss z.B. ein Logger übergeben werden, der soll aber einfach nichts machen, weil das ins File loggen ist nicht das eigentliche Testziel.

Test-Doubles: Stub

Einfache Implementation, welche mit möglichst geringem Aufwand sinnvolle, vordefinierte Werte (z.B. Konstanten) zurückliefert.

Dies erlaubt ein sogenanntes **State**-Testing. Der State wird durch Daten repräsentiert und ist bei Stubs in der Regel konstant.

Für die unterschiedlichen Testziele werden ggf. auch mehrere unterschiedliche Stubs (Implementierungen) erstellt.

Beispiel: Klasse für Authentifikation, welche beliebige Benutzer / Passwörter akzeptiert (`login = true`).

Test-Doubles: Spy

Alternative Implementation, welche dynamische Werte zurückliefern kann. Gleichzeitig merkt sich der Spy exakt die Aufrufe der Methoden (Anzahl, Häufigkeit, Parameter, Zeitpunkt, Exceptions, etc.).

Nach der Interaktion können die aufgezeichneten Ereignisse für die Verifikation des Testfalls genutzt werden.

Dies erlaubt ein so genanntes **Behaviour**-Testing.

Beispiel: Wurde auf dem im Testkandidaten registrierten `ActionListener` die Methode `actionPerformed(...)` auch tatsächlich aufgerufen?

Test-Doubles: Mock

Dies ist eine Spezialisierung des Spy, welche dynamische Werte zurückliefern kann und die korrekte Interaktion selber (das ist die Abgrenzung zum Spy) verifizieren kann.

Mocks werden typisch mit Hilfe von speziellen Mock-Frameworks zur Laufzeit für jeden Testfall als individuelle Mock-Objekte erstellt. Das Verhalten wird dynamisch für jeden einzelnen Testfall und somit programmatisch konfiguriert.

Ein Mock ist dem Spy sehr ähnlich. Einziger Unterschied ist der Ort der Verifikation, Mocks

sind dadurch spezifischer.

Test-Doubles: Fake

Alternative Implementation, welche eine Komponente mit vernünftigem Aufwand vollständig ersetzen kann.

Ermöglicht die vollständige Entkopplung von einer Abhängigkeit. Trade-off: Aufwand dessen Implementation muss in einem vernünftigen Verhältnis zum Nutzen sein.

Beispiel: Abhängigkeit von Webservices wird durch eine lokale (Fake-)Implementation ersetzt:

- Kommunikation fällt weg -> schneller
- Implementation ist trotzdem vorhanden (wenn nicht zu komplex), idealerweise sogar wiederverwendet.

6.3.1 Empfehlungen

Wann setzt man was ein?

- **Dummy und Stub:** Einfache Ersatzimplementierungen um eine bessere Testisolation zu erreichen
- **Spy und Mock:** Universalwaffen für Behaviour-Testing mit Hilfe von Mocking-Frameworks.
- **Fake:** Eher aufwändige Implementation, zur vollständigen Entkopplung vom Original. Aufwand muss sich lohnen!

Sonstiges

- Mocking-Frameworks sind kein goldener Hammer. Es gibt Klassen, die sind zu aufwändig für Mocking. Es gibt auch Klassen die zu einfach sind für Mocking.
- Die Verständlichkeit des Testcodes steht an hoher (erster) Stelle. Wenn ein Testfall durch Mocking so kompliziert wird, dass man ihn nicht mehr versteht, oder sich nicht mehr getraut ihn anzufassen, hat man verloren.

7 SW08 - Fehlertoleranz, Resilienz und Entwurfsmuster

Information: In der SW07 hat kein Unterricht stattgefunden (Osterunterbruch). Deswegen gibt es hier einen Sprung von SW06 auf SW08.

7.1 Fehlertoleranz und Resilienz

Definitionen

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

Fehlertoleranz vs. Resilienz

- **Fehler:** Ursache einer Funktionsstörung.
- **Fehlertoleranz:** System kann trotz Vorkommnissen von Fehlern seine Dienste anbieten.
- **Resilienz:** Widerstandsfähigkeit gegenüber schwerwiegenden Fehlerereignissen. Diese Ereignisse müssen jedoch 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 und operiert exakt wie vor dem Ausfall (Wartbarkeit).

Consensus-Protokolle

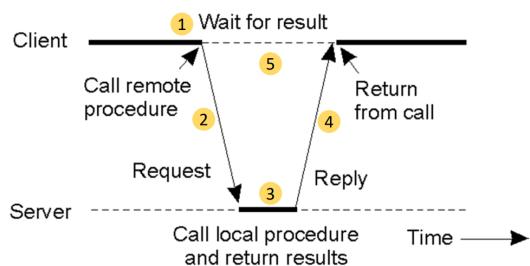
Consensus (dt. Konsens): Bei unterschiedlichen Ausgaben redundanter Systeme muss festgelegt werden, welche Ausgabe verwendet wird.

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

Fehlerarten bei verteilten Systemen

- Zielsystem nicht erreichbar: Daten / Messages können nicht verschickt werden.
- Auslassungsfehler: Falsche Reihenfolge der Daten / Messages. Von TCP zuverlässig verhindert.
- Abrupter Verbindungsabbruch durch Netzwerkausfall: Daten gehen verloren (z.B. Messages)
- Abrupter Verbindungsabbruch durch Systemausfall: Daten gehen verloren (z.B. Messages) oder inkonsistente Daten.

Potenzielle Fehler bei einem asynchronen Aufruf (RPC)



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

7.1.1 Fehlertolerante Systeme

Fehlersituation: Server nicht erreichbar (ohne Redundanz)

Die Gegenstelle ist nicht erreichbar VOR Absenden einer Message.



Erkennung: Aufbau der Verbindung schlägt fehl.

Massnahmen:

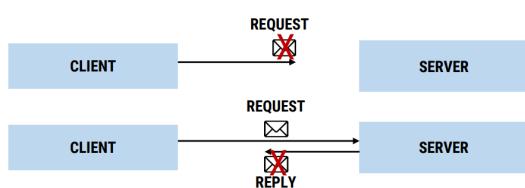
- Kein Betrieb möglich (z.B. Terminalserver). Also müssen User informiert werden.
- Reduzierter Betrieb, falls Server nur Zusatzfunktionalität anbietet.
- Verwendung eines lokalen Caches (ZwischenSpeicher).

Verwendung eines lokalen Caches

- **Beim Lesen:** Beziehe Information aus vorangehender Kommunikation.
- **Beim Schreiben:** Führe SchreibOperationen lokal durch, sende an Server, sobald verfügbar.

7.1.2 Auslieferungsgarantien

Fehlersituation: Verlorene Messages (Request oder Reply)



Erkennung:

- Client startet Timer bei Absenden eines Requests.

- Ist nach Ablauf des Timers noch keine Antwort eingetroffen, gilt die Message als verloren.
- Unterscheidung eines verlorenen Requests von einer verlorenen Reply?

Massnahmen:

- Benutzer informieren / Fragen z.B. bei interaktiven Verbindungen.
- Falls sinnvoll, Request nochmals senden. Duplikatscheck i.d.R. notwendig.

Auslieferungsgarantien

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

At-least-once und Idempotenz

Idempotente Funktionen (math.) sind Funktionen, welche auf sich selbst angewandt das identische Resultat ergeben:

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

Beispiel für eine idempotente Funktion ist die Rückgabe des absoluten Werts: $f(x) = |x|$

Beispiel für eine nicht-idempotente Funktion ist die Rückgabe der Negation: $f(x) = -x$

Idempotente Anfragen

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

Beispiele:

Idempotent:

- Read-Requests: z.B. Abfrage nach einem Fahrplan. Falls Lese-Zugriffe protokolliert werden (z.B. bei Banken) gäbe es einen Nebeneffekt und somit wäre dies nicht idempotent.
- Ändere Adresse von Ort A zu Ort B (ohne weitere Nebeneffekte).

Nicht idempotent:

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

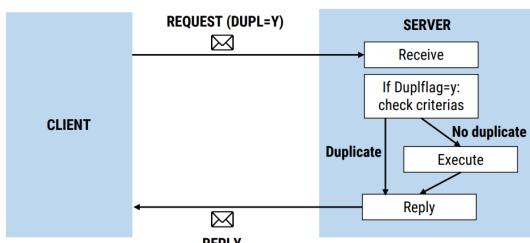
At-least-once und Duplikaterkennung

Die Duplikaterkennung erfolgt entweder durch Heuristik (Methode, um trotz unvollständigen Wissens praktikable Ergebnisse zu erzielen), falls eine gewisse Fehlerwahrscheinlichkeit tolerierbar ist oder per Sequenznummer, falls alle Duplikate erkannt werden sollen.

Erkennen von Duplikaten mittels Heuristik

Mit gewisser Wahrscheinlichkeit durch Heuristiken:

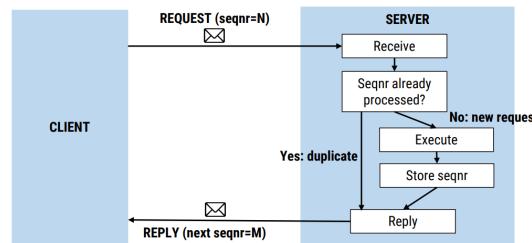
- Duplikatsflag: Client gibt an, dass ein Request wiederholt wird (Optimierung).
- Verwendung diverser Kriterien (Kundennummer, Betrag, Ort, usw.).



Exakte Erkennung von Duplikaten mittels Sequenznummern

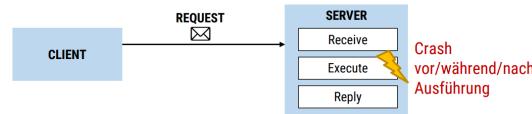
- Jeder Request erhält eine Sequenznummer.

- Server muss sich zusätzlich zur Ausführung die Sequenznummer (seqnr) merken.
- I.d.R. grosse Nummer, erhöht also auch Grösse der Message.
- Mögliches Vorgehen: Server wird bei jeder Antwort die nächste Sequenznummer vorgeben (Request-Response).



7.1.3 Resilienz durch Wiederherstellbarkeit

Fehlersituation: Server-Crash während Requestverarbeitung



Erkennung:

- Client: Keine Unterscheidung gegenüber verlorener Message möglich.
- Server: Kann über Aktionen ein Log führen und dieses Restart abarbeiten (zusätzliche Kosten durch Diskzugriff, erfordert i.d.R. eine Datenbank)

Massnahme:

- Falls vor Ausführung: Client kann Message erneut senden
- Falls während Ausführung: Konsistenz wieder herstellen.
- Falls nach Ausführung: Reply senden. ACHTUNG: Client könnte erneute Anfrage gesendet haben (Duplikat).

Fehlersituation: Client-Crash während Warten auf Antwort



Entweder ist hier die Verbindung unterbrochen oder der Client läuft nicht (identischer Effekt).

Erkennung:

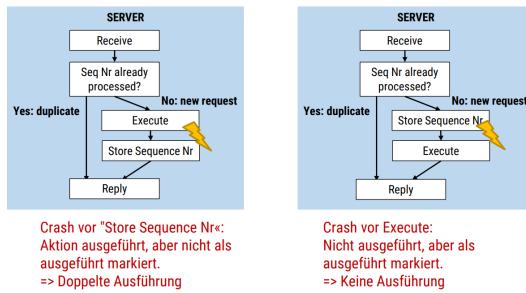
- Server: Keine Verbindung zu Client möglich.

Massnahme:

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

Server-Crash: Fehlende Konsistenz bei Duplikatserkennung

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



Transaktion zur Lösung des Konsistenzproblems

Transaktion:

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

Ablauf:

- Transaktion starten
- Mehrere zusammengehörige Operationen innerhalb der Transaktion durchführen (Daten abfragen, einfügen, modifizieren, löschen).
- Transaktion entweder erfolgreich abschließen (commit) oder abbrechen (rollback). Entweder wurde alles erfolgreich ausgeführt oder es wird nichts ausgeführt.

Transaktionen zur Lösung des Konsistenzproblems (Beispiel)

Geldtransfer von 100CHF von Konto A zu Konto B.

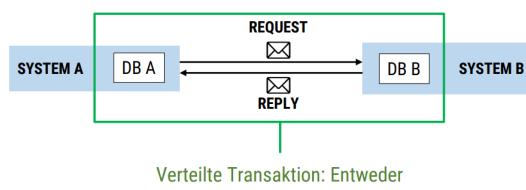
1. Transaktion T starten
2. Account A um 100 CHF reduzieren
3. Account B um 100 CHF erhöhen
4. commit von T

Falls eine der Aktionen vor dem Commit (Schritt 4) fehlschlägt, wird alles rückgängig gemacht und abgebrochen.

7.1.4 Verteilte Transaktionen

Ziel: Kombination von Transaktionen über zwei oder mehr Systeme hinweg. Alle Transaktionen werden entweder ausgeführt oder nicht.

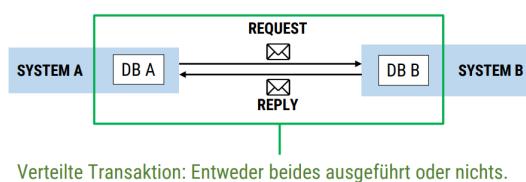
Voraussetzung: Jedes an der verteilten Transaktion teilnehmende System verfügt über einen Transaktions-Mechanismus, z.B. eine Datenbank (DB).



Kommunikation mit verteilten Transaktionen

Vorteil: Einsatz von Sequenznummern usw. ist unnötig

Nachteil: Höherer Ressourcenbedarf und Administrationsaufwand.



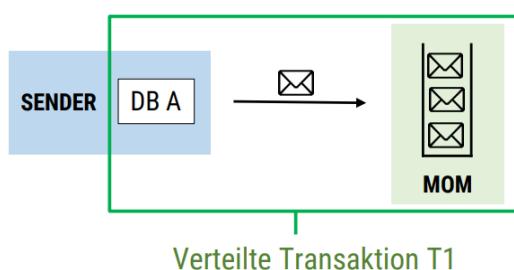
Verteilte Transaktion: Entweder beides ausgeführt oder nichts.

Achtung: Man muss transaktional programmieren (kein autocommit!).

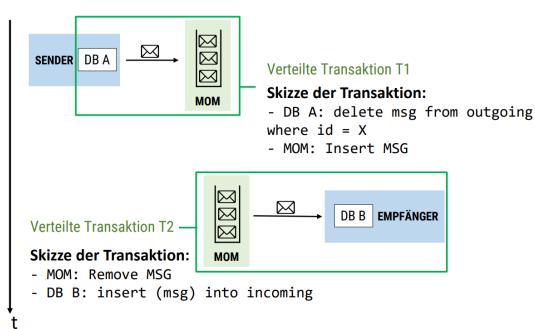
Verteilte Transaktionen mit persistenter Kommunikation

Hier wird wieder eine Message-orientierte Middleware (MOM) verwendet um diese verteilte Transaktionen zu unterstützen.

Das Ziel ist es eine zeitliche Entkopplung mit exactly-once Auslieferungsgarantie zu erreichen.



Ein Beispiel:



Funktionsweise verteilter Transaktionen

- Two-Phase-Commit (2PC): Typischer Algorithmus für verteilte Transaktionen.
- Koordiniert wird 2PC von einem Teilnehmer der verteilten Transaktion, z.B. der ersten Transaktion, welche ein Commit ausführt.
- **Erste Phase:** Koordinator fragt alle beteiligten Systeme, ob Sie die Transaktion erfolgreich abschliessen können (YES) oder nicht (NO).
- **Zweite Phase:** Koordinator trifft Entscheidung: Entweder alle commiten (nur YES erhalten), oder alle brechen die Transaktion ab (mindestens ein NO erhalten).

Two-Phase Commit im Detail

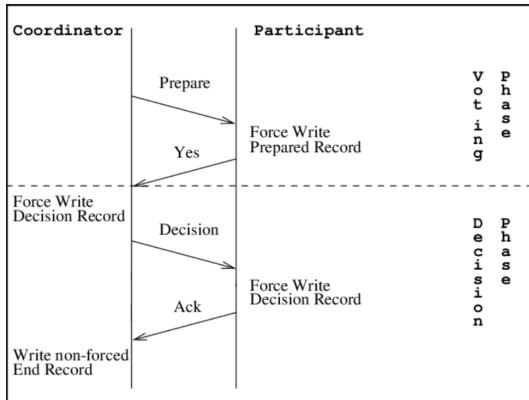
Voting Phase

- Koordinator sendet eine PREPARE-Anfrage an alle Teilnehmer.
- Falls ein Teilnehmer eine PREPARE-Anfrage erhält, antwortet er entweder mit YES, falls er seine Transaktion abschliessen kann und will, ansonsten antwortet er mit NO.

Decision Phase

- Koordinator sammelt alle Antworten der Teilnehmer
 - Falls alle Teilnehmer mit YES geantwortet haben, sendet er eine COMMIT-Anfrage an alle Teilnehmer
 - Falls mindestens ein Teilnehmer mit NO geantwortet hat, sendet er eine ABORT-Anfrage an alle Teilnehmer.
- Jeder Teilnehmer, welcher mit YES geantwortet hat, wartet auf die Anweisung des Koordinators.
 - Falls er eine COMMIT-Anfrage erhält, schliesst er die Transaktion erfolgreich ab.
 - Falls er eine ABORT-Anfrage erhält, macht er die Transaktion rückgängig.

- Falls Koordinator crasht, können andere Teilnehmer angefragt werden.



7.2 Entwurfsmuster (Patterns)

Patterns sind eine Art Schablone für wiederkehrende Entwurfsprobleme.

Wiederverwendung in der SW-Entwicklung

Wiederverwendung von bewährten Entwurfsmustern als Ziel. Verschiedene Arten von Wiederverwendung in der Softwareentwicklung:

- Objekte zur Laufzeit wiederverwenden.
- Wiederverwendung von Quellcode / Klassen.
- Wiederverwendung von einzelnen Komponenten.
- Einsatz von Klassen-Bibliotheken / Frameworks.
- Wiederverwendung von Konzepten, z.B. Entwurfs-, Architektur-, oder Kommunikationsmuster.

Wiederverwendung von Objekten

- Wiederverwendung von Objekten in einer Software während der Laufzeit.
- Beispiele: Threads in einem Executor-Pool, DB-Connection in einem Connection-Pool.

- Effekt: Bessere Performance, höhere Effizienz, geringerer Ressourcenbedarf.
- Herausforderung: Effiziente Verwaltung der Objekte

Wiederverwendung von Quellcode/Klassen

- Wiederverwendung durch
 - Copy & Paste: schlecht (CleanCode: DRY)
 - Vererbung: Häufig schlecht
 - Aggregation und Komposition: Gut (Clean Code: FCoI)
- Effekte:
 - Geringerer Entwicklungsaufwand
 - Geringere Fehlerrate (Klassen sind bereits umfangreich getestet).
- Herausforderungen:
 - Schnittstellen der Klassen eher fremd bestimmt.
 - Auswahl der geeigneten Klassen / Bibliotheken.
 - Lernaufwand, Wartung und Weiterentwicklung.

Wiederverwendung von Komponenten

- Beispiele: Logging-Komponente, Jakarte EE-Beans, Corba-Komponenten, etc.
- Effekte
 - Geringerer Entwicklungsaufwand, weniger Fehler
 - Ganzheitlicherer Ansatz, Blackbox, Abstraktion
- Herausforderungen
 - Anforderungen an die Umgebung / Kontext
 - Eventuell inkompatible Schnittstellen
 - Verwaltung der Komponenten
 - Abhängigkeit vom Lieferanten
 - Wartung und Weiterentwicklung

Herausforderung der (Quellcode-)Wiederverwendung

- Wiederverwendung ist sehr gut, bringt aber auch ein paar Herausforderungen mit sich
 - Unterschiedliche Kontexte / Fachverständnisse.
 - Unterschiedliche Technologien / Lösungsansätze.
 - Einfache Weiterentwicklung und Wartung.
 - Aufwändiges Konfigurationsmanagement.
 - Verschiedene, inkonsistente Designkonzepte.
 - Zusätzliche Abhängigkeit von Dritten.

Wiederverwendung von Konzepten

Die Wiederverwendung von Konzepten ist eine Alternative zur Wiederverwendung von Code.

- Konzepte bleiben relativ konstant und stabil.
- Zusätzlich relativ breit abgestützt und erprobt, weil weitgehend Sprach- und Implementationsunabhängig.
- Die Wiederverwendung von bewährten Entwurfsmustern ist eine sehr elegante, wirkungsvolle, unproblematische und kosten sparende Form von Wiederverwendung

7.2.1 Patterns - Klassifikation

Entwurfsmuster werden primär nach ihrem Zweck klassifiziert. Daraus sind drei Gruppen entstanden:

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

Auch gibt es eine sekundäre Unterteilung in **Klassenmuster** (Legen Beziehungen bereits zum Kompilierzeitpunkt fest) und **Objektmuster** (Beziehungen sind zur Laufzeit dynamisch veränderbar).

Kategorie 1: Erzeugungsmuster

- Abstrahieren die Erzeugung von Objekten
 - Entscheidung welcher (dynamische) Typ verwendet wird.
 - Entscheid über den Zeitpunkt der Erzeugung (z.B. lazy).
 - Entscheid auf welche Art das Objekt konfiguriert wird (Kontext, Initial-Konfiguration, etc.)
- Delegation der Erzeugung an ein anderes Objekt.
- Beispiele sind: Singleton, Factory Method, Builder Pattern, Abstract Factory, Prototype.

Kategorie 2: Strukturmuster

- Fassen Objekte (oder Klassen) zu grösseren oder veränderten Strukturen zusammen.
- oder Erlauben unterschiedlichen Strukturen einander anzupassen und sich miteinander zu verbinden.
- Beispiele sind: Adapter (oder auch Wrapper), Bridge, Decorator, Facade, Flyweight, Composite und Proxy.

Kategorie 3: Verhaltensmuster

- Beschreiben die Interaktion zwischen Objekten.
- Legen die Kontrollflüsse zwischen den Objekten fest.
- Zuständigkeit und/oder Kontrolle delegieren.
- Beispiele sind: Command (Action, Transaction), Observer, Visitor, Interpreter, Iterator, Memento, Strategy, Mediator.

7.2.2 Singleton

Das Singleton-Pattern gewährleistet, dass von einer Klasse genau nur eine einzige Instanz (Objekt) erzeugt wird, und stellt für diese einen Zugriffspunkt zur Verfügung.

Wichtigste Eigenschaften der Implementierung:

- Privates, statisches Attribut für Objektinstanz.
- Öffentliche, statische Methode für Zugriff auf Objekt.
- Privater Konstruktor (verhindert externe Instanziierung).

Singleton hat mittlerweile einen schlechten Ruf, da es zu einer starken Kopplung führt. Ein späterer Austausch ist nur mit grossem Aufwand möglich.

Empfehlung: Sehr zurückhaltend und gezielt einsetzen. Singleton niemals als universellen, globalen Zugriffspunkt verwenden.

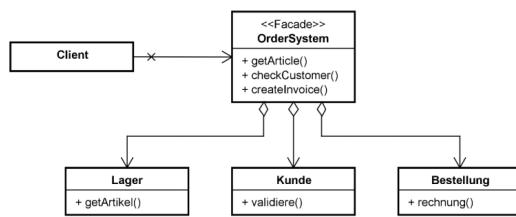
7.2.3 Fassade

Die Fassade stellt eine einheitliche, zusammengefasste Schnittstelle zu einer Menge von Schnittstellen mehrerer Subsysteme zur Verfügung.

Die Fassade

- weiss, welche Subklassen für eine Anfrage zuständig sind und delegiert die Anfragen entsprechend weiter.
- Sorgt für eine konsistente Namensgebung der Methoden.
- Enthält selber keine eigene Funktionalität.

Subsystemklassen Beispiel: Lager, Kunde, Bestellung implementieren die eigentliche Funktion. Sie selbst kennen keine Referenz auf die Fassade.



Motivation für Einsatz:

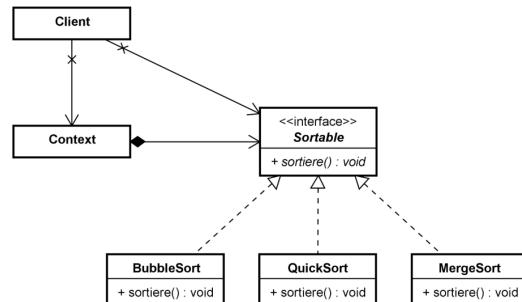
- Vereinfacht die Anwendung mehrerer Subsysteme
- Minimiert die Abhängigkeit zu den Subsystemen (Kopplung minimieren!).
- Einfache Austauschbarkeit eines Subsystems ermöglichen.

Die Gefahr beim Gebrauch einer Fassade ist, dass es zu einem reinen Durchlauferhitzer wird.

Empfehlung: Mit einer Fassade lässt sich sehr gut und einfach entkoppeln. Darauf achten, dass die Fassade nicht plötzlich wesentliche Funktionalität enthält, sie delegiert ausschliesslich.

7.2.4 Strategy-Pattern

Das Strategy-Pattern ermöglicht es, eine Familie von Algorithmen zu definieren, sie austauschbar zu machen und die Algorithmen unabhängig vom Client, der sie verwendet, zu variieren.



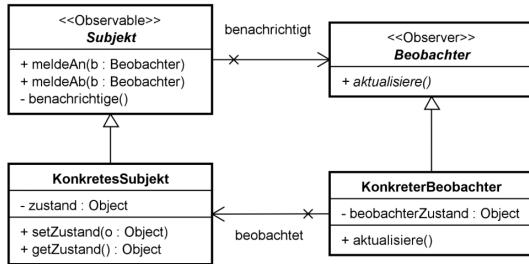
Motivation für Einsatz:

- Anbieten von unterschiedlichen Varianten / Implementationen von Algorithmen (z.B. Zeit vs Speicher, Aufwand).
- Eng verwandte Klassen, die sich nur im Verhalten unterscheiden, zusammenfassen.
- Wenn der Bedarf nach unterschiedlichem Verhalten viele Bedingungsanweisungen zur Folge hätte.

Empfehlung: Ein Pattern, das man leicht unterschätzt, um das sich auch bei sehr kleinen Methoden schon lohnen kann. Außerdem lassen sich damit z.B. grosse und hässliche **switch**-Statements wunderbar eliminieren.

7.2.5 Observer-Pattern

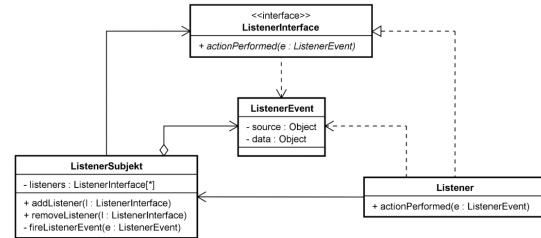
Das Observer-Pattern definiert eine One-To-Many Abhängigkeit zwischen Objekten, sodass, wenn ein Objekt seinen Zustand ändert, alle abhängigen Objekte automatisch benachrichtigt werden.



Motivation für Einsatz:

- Wenn nur eine lose Kopplung der Zuhörer bestehen soll/darf.
- Wenn die Anzahl der vorhandenen Zuhörer nicht interessiert.
- Zur Kommunikation entgegen der Abhängigkeitsrichtung.
- Auch zur Auflösung von zyklischen Referenzen.

Sehr typisch für MVC: Änderungen des Modells müssen an die verschiedenen Views propagiert werden. In Java macht man dies mit Event/Listener-Modell eleganter:



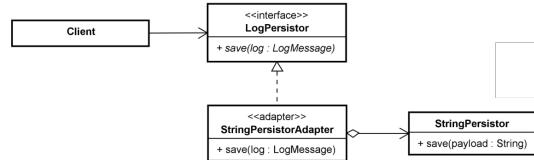
Die Namensgebung bei Event/Listener ist wie folgt:

- Event: Eigentliches Subjekt
- Eventquelle: Verwaltet die Observer
 - `public addXxxListener(...)`
 - `public removeXxxListener(...)`
 - `private fireXxxEvent(...)`
- Listener: Beobachter (`Xxx[Event|Performed](...)`)

Wobei Xxx der Name des Events ist.

7.2.6 Adapter

Das Adapter Pattern passt die Schnittstelle einer vorhandenen Klasse, andie von den Nutzenden erwartete (Ziel-)Schnittstelle an. Es ist deshalb auch bekannt als das Wrapper-Pattern.



Motivation für Einsatz:

- Einfachere Wiederverwendung von existierenden Klassen oder Komponenten, deren Schnittstelle aber unpassend ist.
- Eine möglichst allgemeine Schnittstelle zu implementieren, und diese dann prinzipiell durch Adapter anzupassen.

8 SW09 - Continuous Integration (CI)

8.1 Continuous Integration

Hauptziele von Continuous Integration

- Immer ein lauffähiges Produkt (Buildresultat) zu haben. Es kann somit auch kontinuierlich getestet werden.
- Bei Fehlern jeder Art möglichst schnell ein Feedback zu erhalten. Primär durch automatisierte Unit- und Integrationstests. Aber auch durch Compiler, Classpath, statische Codeprüfung etc.
- Im Team parallel entwickeln zu können und dennoch den Überblick nicht zu verlieren, den Integrationsaufwand zu minimieren und über den aktuelle Zustand auf dem Laufenden zu sein.
- CI gehört zu moderner, zeitgemässer und agiler Software-Entwicklung dazu.

8.1.1 Die 10 Praktiken der CI

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

1 - Versionskontrollsyste

Hier geht es darum, sämtliche Quellartefakte welche für den vollständigen Build einer Software benötigt werden unter Versionskontrolle zu stellen. **Alles was für den Build benötigt wird, nicht aber was erneut gebaut (build) werden kann.**

Dabei sollte man die Fähigkeiten eines VCS nutzen:

1. Sinnvolle Commit-Kommentare vergeben. Ideal: Mit Hinweis auf **Issue** eines Issue Tracking Systems.
2. Tagging: Markieren von bestimmten Versionen für die einfache, eindeutige Identifikation von Releases.
3. Branches: Zweige für parallele Entwicklung. Ermöglicht z.B. die parallele Weiterentwicklung, einfaches Bugfixing, Feature-Branches, etc.

2 - Automatisierter Buildprozess

Der Build soll auf einer kontrollierten, sauberen Maschine durchgeführt werden. Entwickler-Maschinen sind nie sauber (sonstige Dependencies, andere Versionen, etc.).

Der Build soll auf der Basis der aktuellen Quellen aus dem VCS gemacht werden. Dadurch stellt man z.B. sehr schnell und einfach fest, ob man vergessen hat etwas wesentliches einzuchecken.

Der Buildprozess muss inklusive Ausführung der Testfälle und QS-Metriken passieren, so kann man folgende Fragen beantworten:

- Laufen die Unit-Tests tatsächlich immer und überall?
- Gibt es keine Seiteneffekte durch parallele Codeänderungen?
- Ist die Codequalität gut genug?

3 - Automatisierte Testfälle

Möglichst viel des Codes soll durch automatisierte Testfälle abgedeckt sein. Primär Unit Tests, weil diese einfach und überall lauffähig sind. Sekundär auch durch Integrationstests um z.B. Verbindungen oder Abhängigkeiten (Datenbank, Web-API, etc.) zu testen.

Das Ziel ist es so fehlerhafte oder nicht vollständige Implementationen so schnell wie möglich aufzudecken. Bei Integrationstests werden so häufig auch unerwartete Nebeneffekte bekannt.

Bewährt haben sich auch oft Performance-Tests. Wirkt sich ein neues Feature negativ auf die Performance aus?

Das wichtigste ist, dass die Tests immer laufen und im Fehlerfall so schnell wie möglich die Fehler wieder gefixt werden.

4 - Änderungen auf dem Main-Branch des VCS

Ursprünglich wurde gefordert, dass alle Entwickler auf dem einzigen Hauptzweig (**HEAD**, **trunk**, **master**, **main**, etc.) arbeiten. Die Motivation ist, dass sämtliche Änderungen möglichst schnell (und kontinuierlich) in die einzige Codebasis integriert werden.

Bei grossen Teams (welche entsprechend viele Änderungen produzieren) führt das aber zu sehr vielen merges.

Mit modernen VCS welche billige Branches anbieten, begann man darum leichtfertig für jede Entwickler einen eigenen Branch zu eröffnen, um wieder autonomer arbeiten zu können. Das führt aber wieder zu einem sehr aufwändige "BigBang" wenn diese Branches dann vor dem Release in den Hauptzweig gemerged werden müssen. Man fällt wieder in die **Integrationshölle** zurück.

Es wurden verschiedene Branch-Konzepte entwickelt, die einen vernünftigen Kompromiss zwischen autonomer Arbeit und trotzdem genügend häufiger Integration ermöglichen.

Das populärste und bekannteste Prinzip: **Git-Flow**. Mittlerweile von zahlreichen Tools integriert oder adaptiert, um die Abläufe zu automatisieren und vereinfachen.

5 - Automatischer Build bei Änderungen

Ein Buildserver prüft regelmäßig auf Veränderungen im Versionskontrollsysteem (poll) bzw. wird vom SCM aktiv informiert (push). Stellt er solche fest, werden die Quellen ausgecheckt und ein neuer Build gestartet.

Alle Resultate des Build werden offensiv kommuniziert. Erfolg, Testfälle, Laufzeit, Codechecks, Metriken - einfach alles.

Das Ziel ist es immer einen erfolgreichen Build zu haben. Sollte der Build nicht klappen, muss die erste Priorität sein den Build wieder hinzukriegen.

6 - Buildprozess muss schnell sein

Je schneller die Entwickler ein Feedback bekommen, dass etwas nicht mehr läuft, desto besser.

Natürlich muss ein Kompromiss gefunden werden. Manche Tests benötigen mehr Zeit und werden somit kaum lokal durchgeführt. Umso wichtiger ist es, diese im zentralen Build laufen zu lassen!

Alternative: Zwei (oder mehr) gestaffelte Builds durchführen. z.B. einen continuous Build für Feedback an Entwickler und einen langsamen (weil umfangreicher) **nightly build** über Nacht.

Noch flexibler wird man mit Build-Chains oder Pipelines. So wird der Build gestaffelt durchgeführt und man muss nicht auf den gesamten Build warten um gewisse Anomalien oder Fehler zu finden.

7 - Auf realer Umgebung testen

Die zentralen Build- und Testumgebung sollte möglichst ähnlich zur produktiven Umgebung sein (aus DevOps-Prinzipien). Sonst kann es vorkommen, dass beim Testen zwar alles läuft, auf dem produktiven System aber dies nicht mehr der Fall ist.

Das umfasst z.B. folgende Punkte:

- Hardwareaustattung, Betriebssystem, Laufzeitumgebung (Java, etc.), Netzwerkzugriff
- Datenqualität und Datenmenge: Diese soll etwa dieser auf dem produktiven System entsprechen. z.B. könnte es sonst vorkommen, dass ein Timeout auf dem realen System mit mehr Daten eine Funktionalität blockiert.

- Technologien wie Docker helfen hierzu extrem.

In der Realität bei kleinen Systemen ist dies gut zu erreichen, bei grossen System aber häufig viel zu teuer. Ausserdem wird auch Datenschutz zu einer Herausforderung.

8 - Einfacher Zugriff auf Buildartefakte

Sämtliche Buildresultate sollen jederzeit für eine weitere Nutzung zur Verfügung stehen. So dass einfach und schnell die neusten Versionen z.B. weiterführend getestet werden können.

Wird typisch über Buildserver erreicht, welche die Buildresultate selber archivieren können.

Zusätzlich können die binären Artefakte noch in ein binäres Repository deployed werden.

9 - Offensive und offene Information

Für jede Entwickler ist jederzeit einsehbar welche Änderungen ...

- ... von wem und wann eingechecked wurden.
- ... von welchem Build erstmals erfasst wurden.
- ... zu welchen Ergebnissen geführt haben (Build, Tests).
- ... zu welchem Issue gehören.
- ... welche Massnahmen getroffen wurden.

Diese Transparenz und Nachvollziehbarkeit ist eine offene Information und soll nicht zur Kontrolle des Teams dienen, sondern zur gemeinsamen Unterstützung. Mit dem Ziel: **Collective Code Ownership**.

10 - Automatisches Deployment

Wenn immer möglich sollte das Buildergebnis auch automatisch verteilt und installiert werden (aus DevOps-Prinzipien).

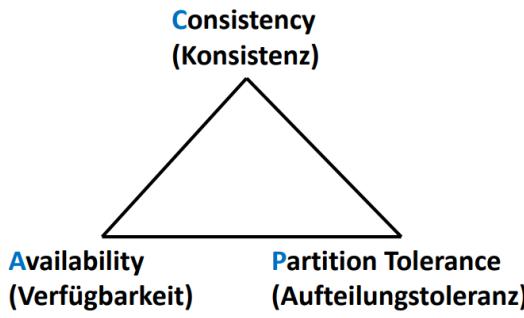
Wenn nicht nach einem Build dann sollte trotzdem zumindest regelmässig das Deployment aktualisiert werden (z.B. täglich, über Nacht). Es macht wenig Sinn alle fünf Minuten eine Server-Applikation zu installieren, wenn ein manueller Test eine Stunde dauert.

9 SW10 - Konsistenz, Replikation, Skalierung und Verteilung

9.1 Konsistenz und Replikation

9.1.1 CAP-Theorem

Brewer's Theorem: Ein verteiltes System kann maximal zwei der folgenden Garantien einhalten:



- **Konsistenz:** Alle beteiligten System haben identische und aktuelle Daten.
- **Verfügbarkeit:** Daten sind für alle Lese-/Schreiboperationen sofort bereit.
- **Aufteilungstoleranz:** System kann trotz Aufteilung in zwei oder mehr bzgl. Kommunikation getrennte Partitionen weiter operieren.

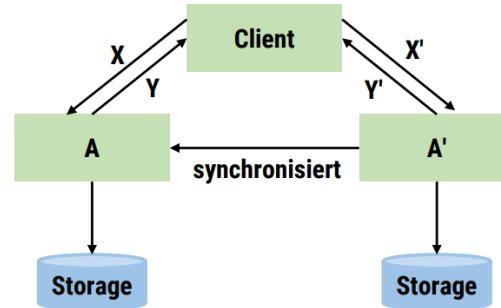
Was bedeutet es, wenn nur zwei der Garantien eingehalten werden können?

- **AP-Systeme:** Operieren trotz ausgefällener Kommunikation weiter. Beispiel: Domain-Name-System.
- **CP-Systeme:** Operieren trotz ausgefällener Kommunikation weiter, aber nur so weit die Konsistenz gewährleistet ist. z.B. Email, Bankautomat.
- **CA-Systeme:** Bieten sowohl Konsistenz als auch Verfügbarkeit dürfen darum nicht partitioniert werden. z.B. typische Web-App.

9.1.2 Konsistenz und Konsistenzgarantien

Konsistenz

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



Liefern A und A' bei identischen Anfragen $X == X'$ die identischen Antworten $Y == Y'$ sind die Daten der beiden Systeme konsistent. Systeme können unterschiedliche Garantien bzgl. der Konsistenz der Daten geben.

Die notwendige Art der Konsistenz wird durch die Anforderungen an die verteilte Applikation bestimmt. Typische Konsistenzgarantien sind:

- **Sequential Consistency:** Bei Sequential Consistency wird sichergestellt, dass alle Prozesse die Änderungen in der gleichen Reihenfolge sehen. Selbst wenn diese Operationen tatsächlich nicht genau in dieser Reihenfolge ausgeführt wurden, müssen die Ergebnisse so erscheinen, als ob sie in dieser Reihenfolge ausgeführt wurden.
- **Eventual Consistency:** Modifikationen werden erst mit einer gewissen Verzögerung sichtbar.

Sequential Consistency Beispiel

1. P1 führt die Operation $X = 1$ aus.
2. P2 liest den Wert von X und erhält 1.
3. P2 führt die Operation $X = 2$ aus.
4. P1 liest den Wert von X und erhält 2.

Beide Prozesse sehen die Reihenfolge als:

$X = 1, X = 2$

Eventual Consistency und Monotonic Reads

Das Gesamtsystem konvergiert gegen die aktuellen Daten. Die Konsistenzanforderung Eventual Consistency erlaubt, dass ein System zu einem Zeitpunkt T noch mit veralteten Daten arbeitet.

Typischerweise in Verbindung mit **Monotonic Reads**: Aufeinander folgende Leseoperationen auf einem Objekt O sehen entweder den gleichen oder einen aktuelleren Wert als den zuletzt gelesenen.

Dies erlaubt eine zeitliche Entkopplung der Systeme. Dies wird eingesetzt wo Daten nicht immer aktuell sein müssen und (falls nur eine Instanz die Hoheit über Modifikationen besitzt ODER nur Schreiboperationen getätigter werden).

Beispiel: System A schreibt welcher User eine gewisse Seite besucht. Und System B zeigt den letzten User an, welcher die Seite besucht hat. Nur wenn System B konsistent den letzten Besucher anzeigt, der die Seite besucht hat, wird die Anforderung an Eventual Consistency mit Monotonic Reads erfüllt.

9.1.3 Replikation

Replika und Replikation

- **Replika:**

- Eine Kopie eines Systems
- Je nach Anwendung muss diese nicht Bit für Bit identisch sein, muss aber die gleichen Informationen enthalten.

- **Replikation:**

- Der Prozess aus einem Original (Primary) eine Kopie (Replikat) herzustellen.
- Hauptanwendungen der Replikation bei verteilten Systemen ist für Ausfallsicherheit (Fällt A aus, kann ein Replikat B übernehmen) und Performance (Lastausgleich auf Replikate).

Fragestellungen zu Replikationen

1. **Wie konsistent müssen Primary und Replikate zu einem bestimmten Zeitpunkt sein:** Die Konsistenz von Primary und Replikation hängt von den Anforderungen der Anwendung ab. Typische Konsistenzgarantien sind Sequential Consistency und Eventual Consistency. Bei Sequential Consistency ist die Reihenfolge der Operationen in allen Systemen gleich. Eventual Consistency bedeutet, dass alle Kopien letztendlich konsistent sein werden, auch wenn es eine Verzögerung gibt.
2. **Wie kann ein Ausfall eines Systems erkannt werden:** Ein Ausfall eines Systems kann durch Heartbeat-Protokolle erkannt werden. Dabei sendet man regelmäßig Heartbeat-Nachrichten an die Systeme. Wenn innerhalb eines bestimmten Zeitraums keine Heartbeat-Nachricht empfangen wird, wird das System als ausgefallen betrachtet. In einem Umfeld mit Auto-Scaling und Load-Balancing könnte man automatisiert neue Instanzen erstellen.

3. **Wie kann der Ausfall eines Systems toleriert werden:** Der Ausfall eines Systems kann durch die Leader-Election-Protokolle toleriert werden. Wenn ein Primary ausfällt, wird ein neuer Primary aus den vorhandenen Replikaten gewählt. Dies kann durch ein Quorum (Mehrheitsentscheid) oder mittels eines zentralen Stores wie ZooKeeper erfolgen, der einen neuen Leader bestimmt.
4. **Wo müssen Primary und Replikate platziert sein:** Die Platzierung von Primary und Replikaten hängt von der Netzwerktopologie und den Performance-Anforderungen ab. Es ist sinnvoll, sie so zu platzieren, dass die Latenzzeiten minimiert und die Ausfallsicherheit maximiert werden. Bei verteilten Systemen sollte man zudem die Netzwerkpartition berücksichtigen und sicherstellen, dass zumindest ein Quorum erreicht werden kann.
5. **Wie wird die Last verteilt:** Die Lastverteilung in einem verteilten System kann durch Load Balancing erreicht werden. Anfragen können auf verschiedene Replikate verteilt werden, um die Performance zu verbessern und die Last gleichmäßig zu verteilen. Die Wahl der Konsistenzgarantien (z.B. eventual consistency) kann ebenfalls die Lastverteilung beeinflussen, indem sie die Anforderungen an die sofortige Synchronisation reduziert.

Klassische Replikation

Ein Scheduler führt in regelmässigen Intervallen (z.B. alle 6h) folgende Anweisungen aus:

1. Erstellung eines Abbilds A des Primarys.
2. Einspielung von Abbild A in alle Systeme, welche als Replikat dienen sollen.

Das Einspielen des Abbilds A in alle Systeme läuft wie folgt ab:

1. Das gespeicherte Abbild A wird von einem zentralen Speicher an alle Replikate verteilt.

2. Das Replikatsysteme aktualisieren ihren Zustand basierend auf diesem Abbild.

Wie konsistent sind diese Replikate?

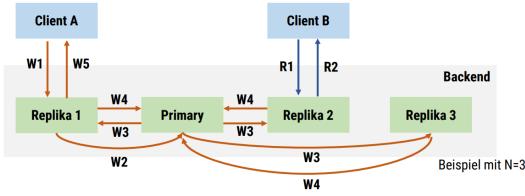
Die Konsistenz der Replikate hängt von der Häufigkeit der Aktualisierung ab. Da die Replikate alle 6 Stunden aktualisiert werden (in diesem Beispiel), können Sie bis zu 6 Stunden hinter dem Primärsystem zurückbleiben. Dies bedeutet, dass die Replikate nicht jederzeit konsistent mit dem Primary sind, sondern nur periodisch.

Was gilt es bei dem Prozess zu beachten?

- **Aktualisungsintervall:** Die Wahl des Intervalls ist entscheidend. Kürzere Intervalle erhöhen die Konsistenz, belasten jedoch das System mehr.
- **Datenintegrität:** Sicherstellen, dass das Abbild korrekt und vollständig erstellt wird, um inkonsistente Zustände zu vermeiden.
- **Ausfallsicherheit:** Planen, wie das System reagiert, wenn der Primary während der Erstellung des Abbilds oder der Verteilung ausfällt.
- **Leistung:** Der Prozess des Erstellens und Verteilens von Abbildern kann ressourcenintensiv sein und die Leistung des Primarysystems beeinträchtigen.
- **Speicherplatz:** Ausreichender Speicherplatz muss vorhanden sein, um die Abbilder zu speichern und verwalten.
- **Konsistenz:** Da es zu Verzögerungen kommen kann, müssen die Anwendungen, die auf die Replikate zugreifen, diese Inkonsistenzen tolerieren können.

Primary-Backup-Protokoll für On-the-Fly Replikas

Bei dieser Methode gibt es einen Primary der alle Schreibanfragen behandelt. So können Replikas On-the-Fly erstellt werden.



- W1: Schreibanfrage
- W2: Weiterleitung an Primary
- W3: Update von Primary an Replikat
- W4: Replikat bestätigt Update
- W5: Bestätige Schreibanfrage

Die Standard-Implementation ist blockierend: Antwort auf Schreibanfrage kehrt erst zurück, sobald Replikas die Daten geschrieben haben.

Wenn das Primary ausfällt, können die Replikate weiter betrieben werden, wobei eines der Replikate als neues Primary fungieren kann, um die Kontinuität zu gewährleisten. Dies erfordert jedoch Mechanismen zur Erkennung des Ausfalls und zur Wahl eines neuen Primary (z.B. Leader-Election).

Da das Primary-Backup-Protokoll blockierend ist, wartet die Antwort auf eine Schreibanfrage, bis alle Replikas das Update bestätigt haben. Dies führt zu einer erhöhten Antwortzeit.

Fallstrick: Filesystem-Cache

Betriebssystemen haben i.d.R. einen Filesystem-Cache. Änderungen werden zuerst in den Cache und erst nach gewisser Zeit auf die Disk geschrieben. Für eine sequenzielle Konsistenz (z.B. Datenbanken) muss Schreiben auf Disk erzwungen werden.

Beispiel: `java.nio.channels.FileChannel` mit dem `void force(boolean metaData)` throws `IOException` { } kann dies übernehmen.

Variationen des Primary-Backup-Protokolls

- Nicht blockierender Schreibzugriff: Ggf. Datenverlust (Daten nicht auf Replikat) nach Crash von Primary oder Primary muss nach Crash wiederhergestellt werden.
- Push- vs. Pull-Prinzip: In festgelegtem Intervall: Keine sequenzielle Konsistenz, aber je nach Daten und Anwendungszweck immer noch **Eventual Consistent**.
- Einzelter vs. verteilte Primarys: Jedes teilnehmende System kann für gewisse Objekte Primary und für andere Objekte Replikat sein. Ist komplexer, aber schneller durch verteilte Schreibzugriffe.

Standard-Replikation bei Datenbank MariaDB

Grober Ablauf:

- Abgeschlossene Transaktionen werden in ein Binary-Log geschrieben.
- Replikas lesen neue Informationen im Binary-Log nach dem Pull-Prinzip und applizieren es lokal.
- Konsistenz: Replikas verfügen nur mit Verzögerung über die gleichen Daten.

9.1.4 Ausfallsicherheit mittels Replikation

Heartbeat-Protokoll

Das Ziel eines solchen Protokolls ist es zu erkennen, ob ein Primary System ausgefallen ist. Das Vorgehen ist wie folgt:

- 1 Primary sendet Heartbeat-MESSAGE in festem Intervall
- N Replika überprüfen, ob diese mindestens ein Heartbeat-Signal innerhalb eines Timeout-Intervalls erhalten haben. Ist dies nicht der Fall, wird der Primary als ausgefallen betrachtet.

Leader-Election mittels Quorum

Das Ziel ist es einen neuen Leader mittels Quorum (Mehrheitsentscheid zu ernennen. Das Vorgehen ist wie folgt:

- 1 (oder mehr) Replika R erkennen Ausfall des Primary.
- Replika R sendet Vote Request an alle anderen Replika.
- Die anderen Replikas senden Vote Grant, falls diese mit Wechsel einverstanden sind (keine anderen Requests; erkennen Primary als Down).
- Replika R wird neues Primary, falls es $\lfloor \frac{N}{2} \rfloor + 1$ Vote Grants erhalten hat.

Wieso braucht es in Quorum?

Um inkonsistente Zustände zu vermeiden darf bei einer Netzwerkpartitionierung nur eine Partition weiter operieren. Ein Quorum ist die Mindestanzahl von Systemen, die erforderlich ist, um Entscheidungen zu treffen und Operationen fortzusetzen.

Ein Quorum ist ab $N = 3$ Systemen möglich, wobei N sinnvollerweise ungerade ist.

Leader-Election mittels externem Store

Ein kleiner aber ausfallsicherer zentraler Store implementiert einen Quorum-Algorithmus. Dieser Store wird von grösseren Systemen verwendet.

Ablauf: N Replika senden Election-Requests an zentralen Store, einer der Replika wird als Primary eingetragen.

Beispiele: ZooKeeper (generischer Store in Java) oder etcd (Kubernetes).

ZooKeeper

ZooKeeper ist Ausfallssicher durch Konsensus-Protokoll. ZooKeeper bietet verteilten Systemen folgende Funktionalität:

- Namensdienste (Auffinden von Systemen).
- Verteilte Synchronisation (z.B. systemübergreifende Locks).
- Verwaltung von Gruppenzugehörigkeit (z.B. für Replikas).

9.2 Skalierung und Verteilung

Topologie verteilter Systeme

Embedded Topologie:

- **Beschreibung:** In dieser Topologie enthalten die Knoten sowohl die Anwendung als auch die Daten.
- **Anwendungsbereich:** Geeignet für Anwendungen, die eine asynchrone Ausführung von Tasks oder Hochleistungs-Computing erfordern.
- **Vorteil:** Da die Daten lokal auf den Knoten gespeichert sind, kann die Kommunikation zwischen den Knoten schneller und effizienter sein.

Client/Server Topologie:

- **Beschreibung:** Diese Topologie besteht aus einem Cluster von Server-Knoten, die Daten speichern und verwalten, während Clients mit diesen Server-Knoten kommunizieren, um auf die Daten zuzugreifen.
- **Anwendungsbereich:** Ideal für Szenarien, in denen skalierbare Server-Knoten erforderlich sind und verschiedene Clients auf die zentralen Daten zugreifen müssen.
- **Vorteil:** Erlaubt eine bessere Skalierbarkeit und zentralisiert Verwaltung der Daten.

Skalierung verteilter Systeme

- **Zustandslose Systeme:** Alle Anfragen sind unabhängig. Mehrere Instanzen einer Serveranwendung sind gestartet. Jede Anfrage wird je nach Auslastung an die Instanzen verteilt. **Einfache Lastverteilung**
- **Zustandsbehaftete Systeme mit temporären Daten:** Mehrere Instanzen einer Serveranwendung sind gestartet. Erste Anfrage einer Session je nach Auslastung an eine Instanz N verteilen. Folgeanfragen jeweils wieder an die gleiche Instanz N leiten

(da man vom vorherigen Request vielleicht das Resultat braucht). **Komplexere Lastverteilung oft mittels Inspektion der Kommunikation**

- **Zustandsbehaftete Systeme mit persistenten Daten:** Daten replizieren (ggf. Partitionieren, d.h. zw. Instanzen aufteilen). Anfragen an diejenigen Instanzen der Serveranwendung verteilen, welche entsprechende Daten vorhält. **Komplexeste Lastverteilung (read vs. write, Konsistenz der Replikas, etc.)**

9.2.1 Lastverteilung mittels Reverse Proxys

Reverse-Proxys sind vorgeschaltete Middleware, welche Anfragen auf verschiedene Instanzen einer Serveranwendung verteilen. Sie bieten:

- Load-Balancing: Last wird auf mehrere Instanzen verteilt, sodass keine Instanz überlastet ist, solange noch andere Instanzen Kapazität haben.
- Weitere Funktionalität wie z.B. Sicherheit (Verschlüsselung, Monitoring, Metriken, Kompression, etc.)

Beispiel Technologien: HAProxy, Webserver wie Apache / Nginx, Traefik.

Lastverteilungsmethoden

- **Round-Robin:** Erstelle eine Liste mit allen Instanzen. Die Liste wird abgearbeitet und jede Instanz in dieser Liste erhält eine Anfrage. Hat die letzte Instanz eine Anfrage erhalten, beginnt der Proxy von vorne. Dies ist gut bei homogener Last der Anfragen und homogener Systemausstattung.
- **Anzahl bestehender Verbindungen:** Leite Anfrage zur Instanz mit der geringsten Anzahl Verbindungen weiter. Dies ist gut für langdauernde TCP-Verbindungen.
- **Hash:** Anwendung einer Hash-Funktion auf IP-Adresse des Clients um die Zielinstanz zu

bestimmen. Alle Anfragen einer IP-Adresse werden an die identische Instanz weitergeleitet. Dies ist eine einfache Möglichkeit Requests an einen gleichen Server zu leiten (benötigt jedoch stabile Client-IPs).

Ausfallsicherheit auf Seite Serverinstanz

Load-Balancer testen mittels Health-Checks, ob eine Instanz funktioniert. Es gibt keine Weiterleitung von Anfragen auf eine Instanz die nicht antwortet.

Typische Health-Checks sind:

- Steht TCP-Verbindung? (Transport-Layer).
- Beliebiger HTTP-Request (Application-Layer).
- Ausführen eines Agents (Hilfsprogramm) auf Serverinstanz.

Ausfallsicherheit auf Seite Proxy

Ein einzelner Reverse-Proxy wäre ein Single-Point of Failure. Deshalb braucht es mindestens zwei Instanzen für eine Ausfallsicherheit.

Dies kann implementiert werden durch eine Kombination mit Hochverfügbarkeitslösungen und Floating-IPs (IP, welche auf verschiedene Ziele geroutet werden kann). Oder auch in dem man bei Ausfall eines Proxys die IP Adresse wechselt.

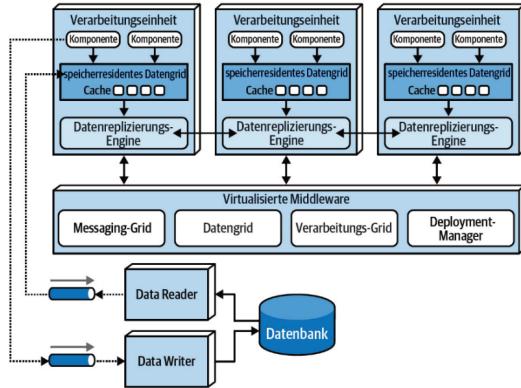
9.2.2 In-Memory Datagrids

Was können In-Memory Datagrids?

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

Einige Implementationen sind z.B. HazelCast, Apache Ignite, Oracle Coherence.

Architektur auf grosser Flughöhe



- Jede Verarbeitungseinheit enthält mehrere Komponenten, die Anwendungslogik ausführen.
- **Speicherresidentes Datengrid (Cache):** Daten werden im Hauptspeicher gehalten, um schnellen Zugriff und Verarbeitung zu ermöglichen.
- **Datenreplizierungs-Engine:** Stellt sicher, dass Daten über verschiedene Knoten hinweg synchronisiert und repliziert werden, um Konsistenz und Ausfallsicherheit zu gewährleisten.
- **Messaging-Grid:** Unterstützt die Kommunikation zwischen den Verarbeitungseinheiten.
- **Datengrid:** Das eigentliche In-Memory Data Grid, das die Daten im Hauptspeicher verteilt und verwaltet.
- **Verarbeitungs-Grid:** Koordiniert die Ausführung der Anwendungslogik über die verschiedenen Knoten.
- **Deployment-Manager:** Verwalten und Bereitstellen der Komponenten und Verarbeitungseinheiten.
- **Data Reader und Data Writer:** Komponenten, die Daten zwischen der persistenten Datenbank und dem In-Memory Data Grid synchronisieren.
- Die Datenbank dient als persistente Speicherung für Daten, die dauerhaft gesichert werden müssen.

Wie funktioniert Hazelcast

1. Hinzufügen eines neuen Knotens

- Der neue Knoten startet und sendet eine Verbindungsanfrage an die bestehenden Knoten im Cluster.
- Die bestehenden Knoten akzeptieren die Anfrage und fügen den neuen Knoten zur Cluster-Membership-Liste hinzu.
- Der neue Knoten erhält eine Kopie der aktuellen Cluster-Membership-Liste.
- Die Daten im In-Memory Data Grid werden neu verteilt, um die Last auf den neuen Knoten zu verteilen (Rebalancing).
- Der neue Knoten beginnt, Datenfragmente von anderen Knoten zu empfangen und zu speichern.
- Der neue Knoten ist nun vollständig in den Cluster integriert und kann Anfragen verarbeiten.

2. Ausfall eines Knotens nach Hinzufügen eines neuen Knotens

- Der Cluster erkennt, dass ein Knoten ausgefallen ist, da Heartbeat-Nachrichten ausbleiben.
- Der Cluster aktualisiert die Cluster-Membership-Liste und entfernt den ausgefallenen Knoten.
- Die Daten, die auf dem ausgefallenen Knoten gespeichert waren, werden von den verbleibenden Knoten übernommen (Replikation).
- Der Cluster führt ein erneutes Rebalancing durch, um sicherzustellen, dass die Daten gleichmäßig auf die verbleibenden Knoten verteilt sind.
- Falls ein neuer Knoten verfügbar ist, kann dieser die Daten des ausgefallenen Knotens übernehmen, um die Ausfallsicherheit und Leistung wiederherzustellen.

10 SW11 - Sicherheit in verteilten Systemen und Deployment

10.1 Sicherheit in verteilten Systemen

Cybersecurity vs. Betriebssicherheit

Cyber-Security beschreibt den Schutz kritischer Systemen und sensibler Informationen vor digitalen Angriffen.

Beim Begriff Betriebssicherheit geht es darum, dass Fehlfunktionen nicht auftreten oder keine kritischen Schäden an Mensch und Maschine verursachen.

Sichere Kommunikation zwischen verteilten Systemen

Situation: A und B sollen sicher kommunizieren. Daten gehen über Drittssysteme, eine lange Verbindung oder werden drahtlos übertragen.

Sichere Kommunikation:

1. A muss sicherstellen, dass B das korrekte System ist.
2. B muss sicherstellen, dass A ein berechtigtes System ist.
3. Kein Drittssystem C soll die Kommunikation mithören.
4. Kein Drittssystem C soll die Kommunikation manipulieren.

Dies während der gesamten Dauer der Kommunikation.

Cyber-Security Massnahmen

- **Zugriffsschutz:** Nur berechtigte Benutzer und Systeme können auf unser System regelmäßig zugreifen. Dies wird sichergestellt mittels Authentifizierung und Autorisierung.
- **Manipulationssicherheit:** Gesendete Daten können nicht manipuliert werden. z.B. Check mit Signaturen.
- **Abhörsicherheit:** Keine geschützten Informationen gelangen nach aussen. z.B. mittels Verschlüsselung.

- **Nachvollziehbarkeit:** Wissen darüber, wie und von wem das System verwendet wurde. z.B. mittels Logs / Audit-Trails.

Datenverschlüsselung ist Basis für sichere Datenübertragung. Hier gibt es eine Unterscheidung von symmetrischer und asymmetrischer Verschlüsselung.

Symmetrische Verschlüsselung

Gleicher Schlüssel zum Ver- und Entschlüsseln wird verwendet. Dies ist um einiges effizienter als asymmetrische Verschlüsselung. I.d.R. eingesetzt zum Verschlüsseln von Datenströmen.

Problem hier: Wie können sich zwei Parteien sicher auf einen gemeinsamen Schlüssel einigen, ohne dass ein Angreifer diesen auch mitbekommt?

Asymmetrische Verschlüsselung

Erzeugung eines Schlüsselpaares: Bei asymmetrischer Verschlüsselung gibt es zwei Schlüssel. Einen zum verschlüsseln (public-key) und einen zum entschlüsseln (private-key). Jeder kennt den public-key und kann so verschlüsselt eine Nachricht an diese Instanz senden. Aber nur die Instanz selbst kennt den private-key um die Message auszulesen.

10.1.1 Transport-Layer-Security (TLS)

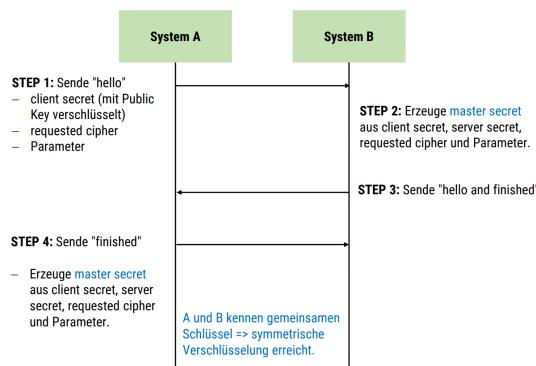
TLS liefert eine transparente Verschlüsselung der Anwendungskommunikation. TLS ist implementiert als Protokoll in der Anwendungsschicht. TLS ist zwischen dem Application- und Transport-Layer.

TLS Versionen:

- 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.

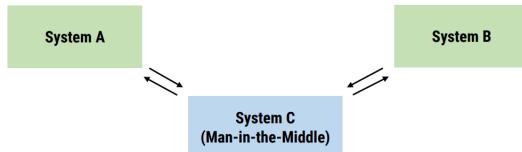
- TLS 1.1, 2006: Nicht mehr in Gebrauch, unsicher.
- TLS 1.2, 2008: In Gebrauch, sicher sofern schwache Verschlüsselungsalgorithmen nicht mehr unterstützt werden.
- TLS 1.3, 2018: Aktuell. Wenn möglich diese Version verwenden (RFC 8446).

Verbindungsauflaufbau (TLS 1.3 Handshake)



TLS gekoppelt mit Zertifikatsprüfung

TLS verschlüsselt in Kombination mit Authentifizierung der Gegenstelle. Authentifizierung mittels X.509 Zertifikaten. Dies ist nötig um “Man-in-the-Middle”-Attacken zu verhindern.



C könnte wie hier gezeigt beim Aufbau der Verbindung A und B eigene Schlüssel senden und Klartextkommunikation lesen. Das heißt C gibt sich gegenüber A als B aus und gegenüber B als A.

X.509 Zertifikate

Diese Zertifikate binden eine Identität mittels digitaler Signatur zu einem öffentlichen Schlüssel. Sie sind zertifiziert durch Zertifizierungsstellen oder selbstzertifiziert. Diese Zertifikate sind unterteilt in zwei Kategorien:

- **Zertifizierungsstelle (CA):** Kann weitere Zertifikate erteilen. Mehrere Hierarchiestufen. Vertrauen in oberste Hierarchie (Root-CA) benötigt.
- **Endstelle:** Identifiziert eine Entität (Domain, Person, Organisation, etc.)

Erstellung eines Certificate Sign Request (CSR)

1. Erstellung Private Key: Erzeugt `server.key` Artefakt (private key).
2. Konfiguration des CSR: Erzeugt `csr.conf` Artefakt (Enthält Angaben über Domain und Organisation, welche von Zertifizierungsstelle überprüft werden).
3. Erstellung CSR: Erzeugt `server.csr`.

Ausstellung Zertifikat durch Zertifizierungsstelle

1. Senden des CSR und weiteren Informationen an Zertifizierungsstelle: Man erhält das `server.crt` der Zertifizierungsstelle (Signierter Public Key).

Zertifikatsaustellung (Selbstzertifiziert)

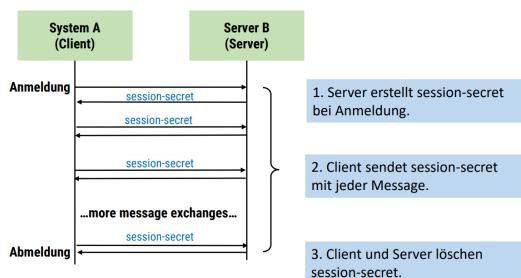
1. Erstellung eigener Zertifizierungsstelle: Erzeugt `rootCA.key` (Private Key der Zertifizierungsstelle) und `rootCA.crt` (Zertifikat der Zertifizierungsstelle).
2. Parametrisierung des Zertifikats: Erzeugt `cert.conf` (Konfiguration des erteilten Zertifikat).
3. Signierung des Zertifikats: Erzeugt `server.crt` (Signierter Public Key).

10.1.2 Sessions

Eine Session ist eine zeitlich beschränkte Zweiwege-Kommunikation (Anmelden bis Abmelden). Typischerweise zwischen Client und Server (Request-Response).

Entkopplung der Session von TCP-Connection

Session Secret sollte nur dem Server und dem einen Client bekannt sein.



Varianten von Session Secrets

Zufällige Zahl ohne Informationsgehalt:

- Zufallsgenerator muss kryptographisch sicher sein.
- Typische Größenordnung: 16 bytes.
- Server speichert Informationen, welche zur Sessions gehören (z.B. Hauptspeicher / Datei / Datenbank / Key-Value Stores).
- Beispiel: Session-Cookie in Web-App.

Verschlüsselte statische Informationen (AccessToken):

- Public / Private-Key Verfahren: Kryptographisch (Signatur) gegenüber Verfälschung gesichert.
- Beispiel: JSONWebToken (JWT). Dies dient als AccessToken (enthält z.B. Verknüpfung mit Benutzerkonto, besteht aus Header / Payload / Signature).

10.1.3 Authentifizierung und Autorisierung

Terminologie

- **Authentisieren:** Eine Partei P (Benutzer oder System) weist sich gegenüber einem System S mittels eines Geheimnisses aus. I.d.R. mittels Kenntnis einer Information, einem Zugang, oder einem Besitz, welcher nur Partei P hat (Passwort, AccessToken, Smartcard, Empfang einer Email, etc.)
- **Authentifizieren:** System S prüft, ob Partei P diejenige ist, welche sie vorgibt zu sein, indem dieses Geheimnis überprüft wird.
- **Autorisierung:** Überprüfen, ob eine authentifizierte Partei berechtigt ist, auf eine Ressource zuzugreifen.

Beispiel:

1. **Authentisierung:** mit Login und Passwort
2. **Authentifizierung:** Abfragen von PW für Login in Datenbank
3. **Autorisierung:** Ist User berechtigt auf die gesuchte Ressource?

In der Regel wird das Passwort einmal beim Login überprüft. Dann wird eine Session erstellt, die temporär für die nahe Zukunft genutzt werden kann.

Passwortprüfung

Passwörter sind persistent als Hashwert gespeichert. Da in jedes System eingebrochen werden kann, sind Klartext Passwörter eine Gefahr für andere Systeme (wenn User z.B. selbe Passwörter auf mehreren Systemen verwenden).

Hier sollten immer vorgefertigte und geprüfte Funktionen und Libraries verwendet werden. Hashfunktionen sind nicht umkehrbar, auch wenn der Vorgang bekannt ist.

Geeignete Hashfunktionen

Es sollen keine generischen Hashfunktionen verwendet werden. Diese sind auf Geschwindigkeit

optimiert und haben eine beschränkte Lebensdauer. Besser ist es eine Passwort-Hashfunktion zu verwenden wie: PBKDF2, Bcrypt, Scrypt. Diese sind langsamer in der Berechnung, was jedoch ein Vorteil ist.

Empfohlen: PBKDF2 in Kombination mit HMAC-SHA256.

Hashfunktionen: Weitere Massnahmen



- **Cost:** Anpassen der Kosten der Hashwertberechnung an aktuelle Hardware. Kosten für ein Login sollte in etwa 100ms sein. Dies ist schnell genug für einen Benutzer, aber teuer für Angriffe.
- **Salt:** Verhindert Brute-Force oder Rainbow-Table Attacken. Pro Passwort wird eine zufällige Zahl oder Zeichenkette, die zusammen mit dem Password gehashed wird gespeichert.
- **Pepper:** Der Pepper ist hier nicht abgebildet. Zusätzlich sind alle Passwörter mit weiterer (für alle Passwörter in einer Applikation identische) Zahl oder Zeichenkette gehashed. Diese Zahl oder Zeichenkette wird nicht in der Datenbank gespeichert.
- **Hash:** Das gehashte Passwort

Autorisierung

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

Typische Verfahren zur Zugriffskontrolle sind:

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

10.2 Deployment

Übersicht

- **Verteilung:** Verteilung der Software über Downloads / Datenträger oder SMS (Software Management System) inkl. Dokumentation
- **Installation:** Kopieren der nötigen Dateien an die vorgesehenen Orte und Registrieren der Anwendung, allenfalls prüfen, ob das Zielsystem für die Anwendung geeignet ist (Hardwareaustattung, OS, etc.).
- **Konfiguration:** Einstellungen der Anwendung auf Benutzer, Netzwerkumgebung, Hardware, etc.
- **Organisation:** Planung, ggf. Produktion, Information (Marketing), Schulung, Support bereitstellen.

Wo findet Deployment statt?

Deployment findet in klassischen Projekten am Ende des Projekts statt. In iterativer und agiler Entwicklung soll das Deployment aber kontinuierlich stattfinden. CI/CD erfordert auch Continuous Deployment.

Continuous Delivery (CD) nutzt DevOps Technologien, wie Infrastructure as Code. Außerdem ist auch Staging immer ein Thema: Deployment auf unterschiedliche Umgebungen (Entwicklung, Test, Integration, Vor-Produktion, Produktion).

Deployment - Umfang

Technische Aspekte

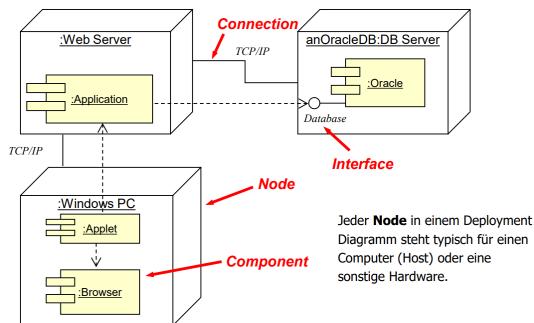
- Deployment Diagramme (Zordnung Komponenten / Hardware)
- Installations- und Deinstallationsprogramme / Skripte
- Konfiguration (Default; Beispiel; etc.)
- Installationsmedium
- Repositories (Ablage der Binaries)

Organisatorische Aspekte

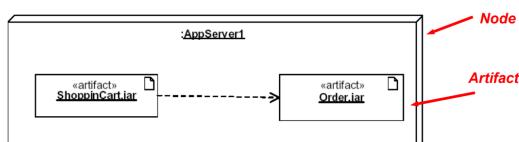
- Konfigurationsmanagement: Welchen Komponenten bilden welche Release (Baseline) oder BOM (bill of material).
- Installations- und Bedienungsanleitungen
- Erwartungsmanagement: Welche Funktionalität ist vorhanden?
- Bereitstellung von Support

10.2.1 Deployment-Diagramme

UML 1.x Deployment Diagramm



UML 2 Deployment Diagramm



- Node: Stellt einen Computer (Host) oder eine sonstige Hardware dar
- Artifact: Stellt ein ausführbares Binary, ein Skript, etc. dar welches durch die Installation explizit einem Node zugeordnet wird.

10.2.2 Deployement - Aspekte

Installation und Deinstallation

Installation (und Update) einer Software soll möglichst vollständig automatisiert sein. Saubere Deinstallation ist ebenfalls wichtig. Dadurch ist dann auch eine vollautomatisierte Verteilung möglich (Software-management). Auch hat die Software so Potenzial für Package-Management-Systeme.

Es gibt sehr unterschiedliche Kundenbedürfnisse / Zielgruppen:

- **Endbenutzer:** Grafische, interaktive GUI-Installation für Endanwender auf dem Desktop.
- **Administrator:** Möglichst Script-basierte, durch Parametrisierung voll automatisierte Installation auf dem Server für den Administrator.
- **Entwickler / Tester:** Spezielle Distributionen, entweder manuell (download) oder über (zentrale) Repositories.

Konfiguration von Anwendungen

Typische Anforderungen

- Datenbankanwendung: Lauffähigkeit auf einer individuellen, bestehenden DBMS-Umgebung.
- Logging / Audit: Einsatz unterschiedlicher Logging- und Überwachungs-Mechanismen/Frameworks.
- Security: Support verschiedener Authentifizierungs- und Autorisierungs-Techniken.

Konfigurationsmanagement

- Mit der Zeit haben verschiedene Kunden unterschiedliche Versionen einer Software. Da muss man einen Überblick haben bez. welche Version läuft bei welchem Kunden? und welcher Kunde hat welche Lizenzen?
- Verschiedene Kunden haben unterschiedliche Produkte und Versionen der Umsysteme (z.B. Datenbank) und der Hardware (z.B. Einfluss auf Performance). Wer hat welche Konfiguration? Mit welchen Komponenten? Welche Kombinationen sind überhaupt lauffähig?

- Ist ein Update von jeder existierenden Konfiguration möglich? Müssen bestimmte Abfolgen eingehalten werden? Wurden die unterschiedlichen Szenarien auch getestet?

Deploymentdokumentation / Manuals

- Release Notes: Neuer Funktionsumfang, Veränderte / Zusätzliche Vorbedingungen, Neue / Veränderte Datenformate oder Protokolle.
- Installationsanleitung: Haben sich HW- oder SW-Voraussetzungen geändert? Varianten unterschiedlicher Konfigurationen berücksichtigen.
- Bedienungsanleitung / User-Manual

10.2.3 Releases und Versionierung

Deployment findet mit einem wohldefinierten Release statt. Dieser braucht eine eindeutige Bezeichnung und Version. Auch sollte man diese im VCS taggen und die technische Version unbedingt von der Marketing Version trennen.

Bewährt hat sich eine dreistellige Version x.y.z. Anhand der Version soll möglichst einfach und klar ersichtlich sein was sich prinzipiell verändert hat.

Semantic Versioning (Repetition, in kurz)

- Major-Release (X.x.x)
- Minor-Release (x.X.x)
- Patch-Release (x.x.X)

Alternative: Time-Based Release Versioning

Man hält sich einen festen Takt an Release-Termen ein. Die Versionsnummer macht deutlich, um welchen Zeitpunkt es sich handelt. Beispiel Ubuntu-Releases: 22.09, 23.04, 23.09, 24.04. Hier hat man sich für das YY.MM-Format entschieden (Jahr.Monat).

Dies ist bei grossen Produkten und/oder Marketing-Versionen sinnvoll, aber weniger bei Libraries/Komponenten.

Versionierung - Release Notes

Die Release-Notes sollen sauber nachgeführt werden, sodass alle Änderungen, Erweiterungen und Korrekturen der Software dokumentiert sind. Dies ergibt dann eine nachvollziehbare Entwicklungsgeschichte.

Release-Notes werden oft manuell nachgeführt, da man qualitative Aussagen tätigen möchte. Oft unterstützt durch Issue-Tracking Systeme.

Für Entwickler dient es als zentrale Informationsquelle um die Möglichkeit bzw. die Notwendigkeit einer Migration auf eine neue Version (und das damit verbundene Risiko) einzuschätzen.

10.2.4 Technisches Deployment (Java)

Java fasst die .class-Dateien und Ressourcen in ein JAR. Für Applikationen für Endbenutzer wird die Applikation oft in binärer Form (setup.exe) mit Anleitung deployed.

Für Server-Applikationen werden klassisch meist separierte JAR-Dateien deployed. Bei Microservices eher Single-JAR deployed.

Für Libraries und Komponenten für Entwickler wird dies oft als binäres Repository deployed (z.B. Maven Repo).

JAR-Dateien

Eine vollständige Applikation kann selber aus 1..n einzelnen JARs bestehen. Sie kann aber auch zusätzlich 0..n Dependency-JARs benötigen.

Bei einem deployment mit mehreren JARs können einzelne augetauscht werden. Ist also modularer. Single-JAR deployments sind zwar einfacher können aber redundante Implementationen oder Dependencies mit sich tragen.

Modularisierung seit Java 9 (Project Jigsaw)

Mit Java wurden die Möglichkeiten zur echten Modularisierung mit Java implementiert. Dabei standen drei Ziele im Vordergrund:

- Reliable Configuration: Den fehleranfälligen Classpath durch den auf Modul-Abhängigkeiten basierenden Modul-Path ablösen.

- Strong Encapsulation: Ein Modul definiert explizit seine öffentliche API. Auf alle restlichen Klassen ist von Ausserhalb kein Zugriff (auch wenn public).
- Scalable Platform: Die Java-Plattform selber wurde modularisiert, so dass für Anwendungen individuell angepasst, schlankere Runtime-Images gebaut werden können.

Umsetzung

- Java-Packages können neu in Modulen zusammengefasst werden. Optionale zusätzliche Strukturebene in der Dateiablage. Es ist eine eindeutige Namensgebung nötig.
- 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 findet beim Start einer Applikation eine Laufzeitüberprüfung statt, ob alle notwendigen Komponenten vorhanden sind. `ClassNotFoundException` Exception sollte somit nicht mehr auftreten.
- Das Ende der JAR-Hell: Es wurden ein neues Format (`jmod`) definiert, der Classpath wird durch den Modul-Path abgelöst.

Verteilung über Binär-Repositories

- Ein Binär-Repository ist eine strukturierte Ablage von Binaries (JAR, WAR, EAR, JMOD, etc.). Primär verwendet für den Entwicklungsprozess (Dependency Management).
- Ursprünglich als Maven-Repository entstanden. Heute meist als spezielle Serversoftware implementiert, erlaubt z.B. effizientere Suche, Management und Analysen.
- Produkte für on-site-Betrieb z.B: JFrog Artifactory, Apache Archiva, Sonatype Nexus.
- Auch als reine Cloud-Dienste: z.B. integriert in VCS wie gitlab.

Deployment bei Open Source Projekten

Deployment erfolgt häufig über zwei verschiedene Distributionen:

- Binär: Enthält binäre Runtime und Dokumentation, direkt einsetzbar.
- Source: Einthält nur den Quellcode und alle notwendigen Buildartefakte.

11 SW12 - Code-Qualität

11.1 S.O.L.I.D.-Prinzipien

Solid fasst 5 wichtige Designprinzipien zusammen:

- Single Responsibility Principle (SRP)
- Open Closed Principle (OCP)
- Liskov Substitution Principle (LSP)
- Interface Segregation Principle (ISP)
- Dependency Inversion Principle (DIP)

Durch Einhaltung der fünf fundamentalen SOLID Prinzipien erreicht man ein qualitativ besseres und schöneres Design. Besseres Design heisst konkret:

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

11.1.1 Single Responsibility Principle (SRP)

Hauptziele:

- Eine Klasse soll nur eine Verantwortlichkeit haben.
- Eine Klasse soll nur einen Grund zur Änderung haben.

Einhaltung von SRP hat eine hohe Kohäsion zur Folge (Es kommt und bleibt zusammen, was zusammen gehört).

Wird SRP verletzt, ergibt sich umgekehrt eine hohe Kopplung. Dies führt zu einer höheren Komplexität und somit schlechtere Wart- und Erweiterbarkeit.

Das SRP liefert im Design typisch viel mehr und auch kleinere Klassen. Dies ist deutlich besser als wenige, grosse Klassen.

Das SRP sagt ausserden, dass es für jede Klasse nur einen Grund für Änderungen geben sollte, sonst ist das SRP schon verletzt.

11.1.2 Open Closed Principle (OCP)

Hauptziele:

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

Offen für Erweiterungen: Design ist für eine einfache und sichere Erweiterbarkeit ausgelegt, beispielsweise durch Einsatz des Strategy Patterns (Gof).

Geschlossen für Änderungen: Design ist so ausgelegt, dass bestehende Methoden und Klassen bei einer Erweiterung möglichst nicht verändert werden müssen.

Die Motivation dahinter ist es das Fehlerrisiko zu minimieren beim Einbau von Erweiterungen.

z.B. Anstelle ein `switch-case`, welches in Zukunft vermutlich erweitert werden würde, soll mit Hilfe des Strategy-Pattern gearbeitet werden.

11.1.3 Liskov Substitution Principle (LSP)

Hauptziele:

- Subtypen sollten sich so verhalten wie ihre Basistypen.
- Einfacher formuliert: In spezialisierten Klassen nur Methoden ergänzen oder für Erweiterung überschreiben, aber nie fundamental verändern!

Beispiel: Wenn eine Klasse `Kreis` eine Spezialisierung von der Klasse `Ellipse` ist. Kann man die Funktionen aus `Ellipse` (`skaliereX()`, `skaliereY()`) nicht mehr verwenden, da für einen Kreis ja $r_x = r_y$ gelten muss.

Somit stimmt in diesem Beispiel die Anforderung: "Die Achsen können unabhängig voneinander skaliert werden" hier nicht mehr. Somit wird das LSP verletzt.

11.1.4 Interface Segregation Principle (ISP)

Hauptziele:

- Interfaces strikt von Details der Implementation trennen.
- Interfaces strikt voneinander trennen (keine Überschneidungen, keine Population von Schnittstellen, keine fat-Schnittstellen).

Clients sollen nicht mit Details belastet werden, die sie nicht benötigen.

Gibt es verschiedenartige Klienten eines Systems, sollte jeder Typ von Klient sein eigenes Interface haben. So werden Abhängigkeiten minimiert und somit auch die Kopplung. Dies hat auch einen positiven Einfluss auf die Sicherheit.

Refactoring für ISP

- Superklasse aus bestehender Klasse extrahieren.
- Interface aus bestehender Klasse extrahieren.
- Bestehende Interfaces auftrennen für schmalere Schnittstellen (Separation of Concerns (SoC)).

11.1.5 Dependency Inversion Principle (DIP)

Hauptziele:

- Änderungen voneinander isolieren.

High-Level Klassen (Hoher Abstraktionsgrad) sollen nicht von Low-Level Klassen (konkrete Implementation) abhängig sein, sondern allenfalls beide von Interfaces. Siehe auch Single Level of Abstraction (SLA).

Eine gute Lösung ist es mittels Dependency Injection mittels Konstruktor und einem Adapter die Abhängigkeit zu entkoppeln.

11.2 C.U.P.I.D.-Eigenschaften

CUPID fasst 5 wichtige Designprinzipien zusammen:

- Composable - lässt sich gut mit anderem nutzen.
- Unix philosophy - kümmert sich genau um eine Sache.
- Predictable - macht das, was man erwartet.
- Idiomatic - Nutzung fühlt sich natürlich an.
- Domain-based - sowohl in Sprache als auch Struktur.

CUPID wurde entwickelt als Ergänzung zu den SOLID Eigenschaften. Motivation für CUPID ist es nicht sture Regeln und Richtlinien zu definieren, sondern positive Eigenschaften, die mehr oder weniger erfüllt werden können.

11.2.1 Composable

- Guter Code hat eine kleine Oberfläche: Von Aussen sind nur jene Teile sichtbar, die für die Entwickler relevant sind.
- Die (Wieder-)Verwendung des Codes soll möglichst einfach sein.
- Eindeutig definierten Schnittstellen helfen beim Verständnis und Verwenden des Codes.
- Intention des Codes soll offensichtlich sein.

11.2.2 Unix Philosophy

- Guter Code sorgt dafür, dass eine Softwareeinheit nur genau eine Aufgabe erledigt.
- Klassisches UNIX-Konzept: Viele, kleine Tools, die sich vielseitig kombinieren lassen.
- Errinnert an SRP (Single Responsibility Principle).

11.2.3 Predictable

- Der Code verhält sich wie erwartet, ohne unliebsame, unerwartete Überraschungen (keine Nebeneffekte).
- Der Code ist technisch beobachtbar (keine Magie). Dies vereinfacht sowohl das Debugging als auch die Fehlersuche.
- Der interne Status einer Softwareeinheit kann von den Ausgaben zwar abgeleitet werden, ist aber nicht veränderbar.

11.2.4 Idiomatic

- Wenn der Code gut ist, dann fühlt sich dessen Nutzung und die Bearbeitung des Codes natürlich (auch leicht verständlich) an.

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

11.2.5 Domain-based

- Für guten Code wird die DomänenSprache und deren Struktur verwendet
- Die Struktur des Codes soll die Lösung wiederspiegeln, und nicht das darunterliegende Framework und/oder eingesetzten Konzepte.
- Anstatt technischer Begriffe und Konzepte (Models, Views, Controllers) sollen Domänenbegriffe (Dokumente, Zahlungen) verwendet werden.

12 SW13 - Komponentenmodelle und Technische Schulden

12.1 Komponentenmodelle

12.1.1 Übersicht Komponentenmodelle

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 die grundlegenden Eigenschaften einer Komponente fest.
- Rahmen für die Komponentenentwicklung oft in Zusammenhang mit einer Laufzeitumgebung.
- Basis für die Interaktion: Nur kompatible Komponenten können miteinander interagieren.
- Komponentenmodelle sind unterschiedlich spezifisch.

Grundlegende Eigenschaften eines Komponentenmodells

- **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? (Austauschbarkeit, Auffindbarkeit).
- **Deployment:** Wie werden Komponenten ausgeliefert? Wie werden Komponenten gebündelt?

12.1.2 Das OSGi-Komponentenmodell

Dies ist ein leichtgewichtiges Modul- und Servicestystem für Java. Es sind Implementationen verschiedener Hersteller verfügbar (Apache Felix, Gemini, Concierge OSGi, Knopfferfish, etc.). Einige bekannte Anwender von OSGi sind NetBeans, Confluence und JIRA, Eclipse, IntelliJ, usw.

OSGi als Komponentenmodell

- **Schnittstellendefinition:** Java-Interfaces
- **Kommunikation (Verteilung):** Method-Calls
- **Komposition (Austauschbarkeit, Auffindbarkeit):** Import/Export von Interfaces, Service Registry, Isolation der nicht-exportierten Packages
- **Deployment:** Einzelne JARs, Bundle-JAR

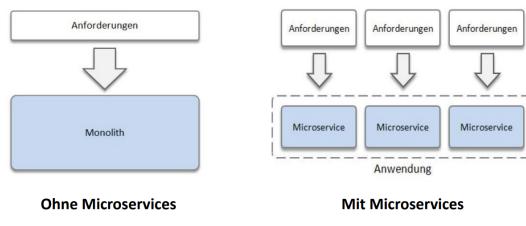
12.1.3 Ein Komponentenmodell für Microservices

Was sind Microservices

Die Microservice Architektur hat das Ziel eine einzige Software als Paket aus einzelnen kleinen Services zu bauen. Diese Services erledigen alle eine eigene Aufgabe und sind unabhängig von den anderen deployable.

Aufteilung nach Business Capabilities

Das Ziel ist es eine bessere Aufteilung auf verschiedene Entwicklungsteams hinzukriegen. Jedes Team kann sich auf eine Fachdomäne konzentrieren und ggf. den Service separat ausliefern.



Eigenschaften von Microservices

- **Aufteilung in Services:** Anwendung in kleine Services unterteilt, welche als Komponenten verwendet werden.
- **Schichten übergreifend:** enthält alle Schichten einer Funktionalität / Domäne (Frontend / Backend / Businesslogik / Persistenz).
- **Unabhängiges Deployment:** Deployment der kleinen Services einzeln.
- **Kommunikation über Netzwerk:** REST, MQ, SOAP, etc.
- **Entkopplung:** Microservice nur über Schnittstellen bekannt, Ausführung als eigener Prozess.
- **Technologische Entkopplung:** Pro Microservice kann Plattform, Datenbank, Programmiersprache individuell festgelegt werden.

Das Komponentenmodell für Microservices

- **Schnittstellendefinition:** OpenAPI
- **Kommunikation (Verteilung):** HTTP REST API, Verteilung über virtuelle und physische Systeme möglich.
- **Komposition (Austauschbarkeit, Auffindbarkeit):** Orchestrierung: Automatische Vergabe von Name/IP und Port an Service-Anbieter, Automatische Übergabe von Name/IP und Port an Konsument z.B. mit Umgebungsvar. Isolation durch unabhängige Prozesse
- **Deployment:** Gebündelt mit HTTP-Server und allen Abhängigkeiten als Container-Image. Abgelegt in einem binär-Repository.

12.2 Technische Schulden

12.2.1 Herausforderungen der statischen Codeanalysen

Beispiele für Java:

- Checkstyle
- PMD
- Spotbugs

Die Herausforderungen sind vielfältig:

- Pflege der Konfiguration (Projekt vs. Zentral).
- Unterschiedliches, heterogenes Reporting, Deoppelbefunde.
- Umgang mit false-positive.
- Integration der Tools in Build und Entwicklungsumgebung.

12.2.2 Konzept der technischen Schulden

Zentrale Idee: Man hinterlegt für jede Art und Kategorie von Befunden (auch aus verschiedenen Tools) einen zeitlichen oder monetären Preis für die Verbesserung.

Damit lässt sich mit einer einfachen Zahl der Zustand einer Software als so genannte technische Schuld eindrücklich kommunizieren. Es werden alle Befunde aufsummiert.

Die technische Schuld ist quasi die Hypothek für die effiziente Weiterentwicklung und Wartung eines Systems.

Kategorisierung der Befunde

Die technischen Schulden lassen sich durch geschickte Kategorisierung auch relativ gut aufschlüsseln. Man kann Befunde mit Einfluss z.B. auf:

- die Wartbarkeit
- die Sicherheit
- die Testbarkeit

identifizieren und gruppieren. Somit lassen sich dann aussagen treffen wie z.B. "Wir haben n Stunden Aufwand, um die Wartbarkeit zu verbessern". Diese Idee hat zur Entwicklung der SQALE Methodik geführt.

12.2.3 SQALE Methodik

SQALE definiert u.a. neun aufeinander aufbauende Kategorien.



Das Kriterium der Testbarkeit ist somit von

fundamentalem Charakter. Und das Ziel der guten Wiederverwendbarkeit ist das höchste Ziel.

12.2.4 SonarQube

SonarQube ist praktisch gleichzeitig mit der SQALE-Methodik entstanden. Einige zentrale Features von SonarQube sind:

- Webapplikation, mit zentralen Regelsets.
- Dezentrale Analysen von Projekten, zentrale Speicherung in DB.
- Vereinfachte SQALE-Methodik, berücksichtigt aber auch dynamische Analysen (z.B. Testausführung, Code-Coverage).
- Kann als hartes Q-Gate genutzt werden (speziell in DevOps-Pipelines sehr interessant).
- SQALE-Methodik ist als kommerzielles Plugin verfügbar.

13 SW14 - Softwarekonfigurationsmanagement

13.1 Softwarekonfigurations management

13.1.1 Konzept und Begriffe

Ziel: über den gesamten Systemlebenszyklus hinweg Änderungen an der Konfiguration kontrolliert durchführen und die Integrität und Rückverfolgbarkeit sicherstellen.

Methodik: Identifizieren der Konfiguration eines Systems auf verschiedenen Ebenen und zu verschiedenen, definierten Zeitpunkten.

Warum Konfigurationsmanagement?

Während Entwicklung und im Betrieb eines Systems sind dessen Bestandteile Änderungen unterworfen. Dies sind u.A:

- Software (möglicherweise mehrere Komponenten)
- die dazugehörigen Einstellungen
- Hardware
- Firmware
- Dokumentation

Nicht alle Versionen aller Bestandteile funktionieren korrekt miteinander. Dies zeigt die Notwendigkeit des Konfigurationsmanagements, sobald ein System aus mehreren Teilen besteht.

Begriffe

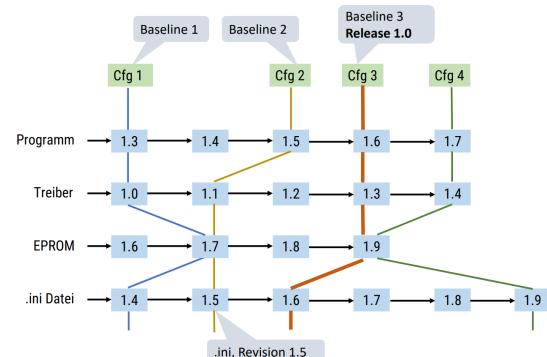
- **Konfiguration:** Sammlung bestimmter Versionen von Hardware-, Firmware-, oder Softwareelementen, welche gemäss einem festgelegten Verfahren kombiniert werden, um einen bestimmten ZWeck zu erreichen.
- **Konfigurationselement:** Aggregation von Software, welche in Rahmen des Konfigurationsmanagements, als einzelne Einheit behandelt wird. Weitere Elemente, welche unter das Softwarekonfigurationsmanagement fallen, sind:

- Pläne

- Spezifikationen
- Bibliotheken
- Daten und Datenverzeichnisse
- Dokumentation für Installation, Wartung und Betrieb

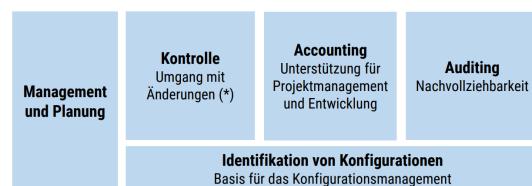
- **Version:** Ein spezifisches, identifizierbares Artefakt auf einem bestimmten Entwicklungsstand.
- **Revision:** Eine neue Version eines Artefaktes mit dem Zweck eine ältere abzulösen.
- **Baseline:** Ein Satz von Revisionen, d.h. ein Snapshot der Konfiguration.
- **Release:** Eine getestete und freigegebene Baseline.

Beispiel: Versionen, Revisionen, Konfigurationen und Releases



13.1.2 Klassisches Konfigurationsmanagement nach IEEE

Es gibt 5 Bereiche für die Organisation des Software-Konfigurationsmanagements



Konfigurationsmanagementplan nach IEEE

1. **Einführung:** Zweck, Gültigkeitsbereich, Begriffsdefinitionen.
2. **SCM Management:** Wer? Zuständigkeiten und verantwortliche Stellen 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. **Wie wird sichergestellt, dass dieser Plan dem Projektablauf entsprechend nachgeführt wird?:**

13.1.3 Modernes Softwarekonfigurationsmanagement

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

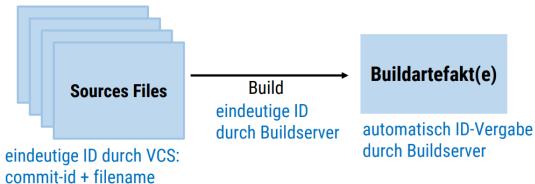
Softwareerstellungsprozess (ab Entwicklung)

Ziel des modernen SW-Konfigurationsmanagements:

- Für jeden Teil der Entwicklungs-Stages (Entwicklung, Test, Staging, Produktion / Release) sind die Input- und Output-Artefakte exakt identifizierbar und reproduzierbar.
- Pipeline ist möglichst vollständig automatisiert (Teil von DevOps).

Build Engineering - Welchen Build verwendet der Kunde?

Ist eine manuelle Versionierung zulässig? Alternativ oder zusätzlich: Automatisiert immutable (nur änderbar mit Signaturen) Build-ID ins Build-Artefakt schreiben.



Es gibt dafür auf verschiedenen VCS auch vordefinierte Variablen die man brauchen kann. z.B. ist es in der `.gitlab-ci.yml` Konfiguration die Variable `$CI_COMMIT_SHA`.

Ausserdem sollte wenn möglich immer eine exakte Containerimage Version bei dem Build-Job gewählt werden. Ein Update des Images könnte den Build brechen.

Environment Configuration

Ziele:

- Pro Komponente nur ein Buildartefakt. Dieses läuft auf allen Umgebungskonfigurationen.
- Umgebungen können zur Laufzeit identifiziert werden.
- Umgebungskonfigurationen sind dokumentiert und identifizierbar.
- Änderungen an Umgebungskonfigurationen werden versioniert.
- Reproduzierbare Entwicklungs-, Test-, QA-, und Produktionsumgebungen.

Generell sind meistens in allen Umgebungen (Dev, TEst, Staging, Prod) der Code und die Komponenten gleich. Was sich i.d.R. ändert sind die Daten und die Konfigurationen.

Daher werden hardcodierte Umgebungsinformationen zum Problem (Adressen, Ports, Dateipfade, etc.). Dies kann man einfach lösen, indem

man z.B. Umgebungsvariablen oder Konfigurationsdateien für die Umgebungen verwendet. Diese kann man z.B. auch in einem Docker Container setzen.

```

1 services:
2   web:
3     image: myserver
4     ports:
5       - "433:5000"
6     environment:
7       LISTEN_PORT: 5000

```

Wir sehen hier, dass die Umgebungsvariable `LISTEN_PORT` in diesem Docker Compose auf den Wert 5000 gesetzt wurde.

Weitere Tools zur Umgebungskonfiguration

Beispiele:

- Docker Swarm / Kubernetes: Transparente Verteilung auf mehrere Systeme. Dazu gehört dann auch Verschlüsselung, Lastverteilung und Redundanz.
- Helm: Package Manager für Kubernetes, kann z.B. Abhängigkeiten automatisch auflösen.

Wichtig ist: Die Konfiguration muss versioniert werden können. Ideal ist Infrastructure as Code (z.B. mit Terraform).

Change Control

Ziel: Kontrollierte und nachvollziehbare Änderungen an allen SW-Konfigurationselementen (Code, Konfigurationen, Doku, etc.)

Damit versucht man zu kontrollieren, wer was und unter welchen Bedingungen ändern darf. z.B. Wer darf den Code ändern? Wer darf Konfigurationen ändern? Wer darf auf welchem Environment deployen?

Change Control Board (CCB)

CBB koordiniert Änderungen pro Release und Umgebung. Verschiedene Zusammenstellungen des CCB sind möglich:

- Kleiner Kreis von Seniors (Achtung: Bottleneck)

- Release Manager pro Release mit Stv.
- Senior pro Komponente mit Stv.
- N Peers.

DBB legt ausserdem Richtlinien fest, ob und wann eine Änderung auf einem bestimmten Release gemacht wird (Gatekeeping), oder ob dies zu riskant ist.

CBB legt auch den Ablauf bei Notfalländerungen und Übersteuerungen bei Nichtverfügbarkeit von kritischen Personen fest.

Auch legt das CBB die Standardkorrekturregeln fest bei der Softwareproduktion mit mehreren Releases.

Gebräuchliche Korrekturregeln

Sollte während der Entwicklung ein nicht kritischer Fehler auftreten, kann man diesen für den nächsten Release ganz einfach beheben und mit diesem Publizieren.

Sollte aber ein kritischer Fehler auftreten, muss dieser Fehler direkt behoben werden. Dann müssen auch die Kunden, die die fehlerhafte Version nutzen, informiert werden.

CBB-Regeln im Gitlab

Im Gitlab gibt es einige Optionen CBB-Regeln umzusetzen (z.B. Approval Settings, Merge request approval configuration, etc.).

Release-Management

Ziel: Schneller, planbarer und wiederholbarer (i.d.R.) automatisierter Prozess, um einen Release zu paketieren, sodass alle Komponenten dieses Pakets eindeutig identifizierbar sind.

Typischerweise wird ein Skript aufgesetzt, welches die Bestandteile eines Releases aus den verschiedenen REPOS in einem Release bündelt.

Deployment

Ziele: Problemlose Einführung eines bestimmten Release auf Produktion (ggf. QA).

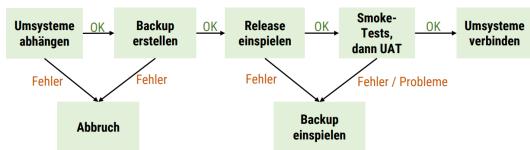
Produktion:

- Eigenes Produktionssystem.
- Freigabe zum automatischen Update durch Clients.
- Freigabe zum Download.

Einspielung eines Release soll funktionieren wie ein Lichtschalter:

- Zu jedem Zeitpunkt müssen betriebsverantwortliche Personen wissen, welcher Release auf der Produktionsumgebung läuft.
- Betriebsverantwortliche Personen müssen die Einführung bei Problemen rückgängig machen können.

Standardablauf für eigene Produktionssysteme



14 SW15 - Koordination verteilter Systeme

14.1 Koordination verteilter Systeme

14.1.1 Physische Zeit

Falls es keine globale Einigung auf die Zeit gibt, kann es durch die lokalen Uhren auf den Systemen bei kleinen Unterschieden schon ein verteiltes System inkonsistent machen.

Notwendigkeit der Uhren-Synchronisierung

Uhren in Computerin zählen mit Hilfe von Schaltungen, welche die Schwingungsfrequenz von Quarzkristallen unter Spannung messen. Auch auf ursprünglich synchronisierten Uhren, läuft die Zeit auseinander, weil Quarzkristalle mit unterschiedlicher Qualität verwendet werden und deshalb mit unterschiedlichen Frequenzen schwingen.

Algorithmus von Cristian

Zeitserver: Maschine mit Zeitzeichenempfänger, mit diesem Server werden alle anderen Maschinen synchronisiert. Dies ist eine Art Zeitzeichensender der am Anfang jeder UTC-Sekunde einen kurzen Impuls sendet (UTC = Universal Coordinated Time).

Der Algorithmus:

1. Client P erfragt die Zeit von Zeit-Server S zum Zeitpunkt t_0
2. Die Anfrage wird von S verarbeitet - dies benötigt eine Zeitspanne I .
3. Die Antwort $C_{UTC}(t_1) + RTT/2$ gesetzt, d.h. die vom Server gemeldete Zeit plus die Rücklaufzeit des Pakets. Die Round Trip Time (RTT) wird dabei berechnet durch $RTT = t_1 - T_0$. Ist die Zeitspanne I bekannt, kann die Berechnung verbessert werden $RTT = t_1 - t_0 - I$.
4. Für genauere Werte wird die Laufzeit öfters gemessen, Messungen ausserhalb eines Bereiches werden verworfen und eine Mittelung der restlichen Werte durchgeführt.

Probleme: Zeit kann nicht rückwärts laufen.
Sollte die Uhr voraus sein, muss man die lokale Zeit verlangsamen, bis die Zeitdifferent ausgeglichen wurde.

Auch braucht die Antwortet des Zeitservers Zeit. Die Laufzeit der Anfrage kann nicht genau bestimmt werden, dies ist abhängig von Netzwerklast. Kompensation durch mehrfache Messung der Dauer der Anfrage und Adaption des vom Zeitervers gelieferten Werts.

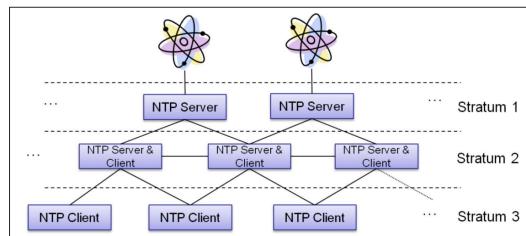
Barkeley-Algorithmus

Keine Maschine hat einen Zeitzeichenempfänger. Der Zeitserver (Zeit-Demon) fragt in regelmässigen Abständen die lokale Zeit von allen teilnehmenden Clients ab. Basierend auf den Antworten berechnet der Zeitserver eine Durchschnittszeit und weist alle Maschinen an, ihre Uhren der neuen Zeit anzupassen.

Network Time Protocol (NTP)

Ziel: Synchronisierung von Rechneruhren im Internet. Der NTP-Demon ist auf fast allen Rechnerplattformen verfügbar, von PCs bis Crays; Unix, Windows, VMs, embedded Systems.

Die erreichbare Genauigkeit beträgt ca. 10 ms in WANs und weniger als 1 ms in LANs.



Prinzipieller Ablauf:

1. Client sendet eine NTP-Message an den Timserver.
2. Server verarbeitet Paket
3. Server sendet Paket zurück
4. Client hat nun vier Zeitstempel (t1-t4) und leitet davon Offset und Delay ab.

Client -> Server ($t_1 \rightarrow t_2$), Server -> Client ($t_3 \rightarrow t_4$).

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

Offset beschreibt die Zeitdifferenz der Rechneruhren (gemittelt). Delay beschreibt die Zeit während das Paket unterwegs war.

14.1.2 Logische Zeit

Eine Übereinstimmung mit der Zeit ausserhalb des Systems ist nicht notwendig (keine physische Zeit nötig). Logische Zeit findet vor allem in Bereichen Anwendung, in denen Kausalität und Verlässlichkeit eine grosse Rolle spielen.

Happened-Before-Relation von Lamport

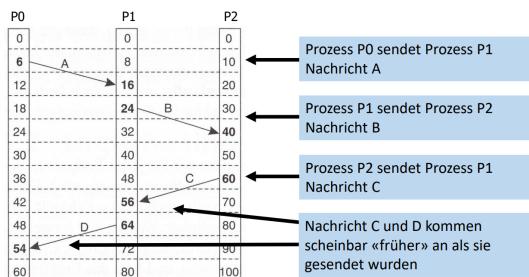
Der Ausdruck $a \rightarrow b$ wird gelesen als "a passiert vor b". Dies bedeutet, dass sich alle Prozesse einig sind, dass zuerst das Ereignis a stattfindet und dann das Ereignis b.

Zwei Ergebnisse $a \neq b$ sind kausal unabhängig, geschrieben als $a \parallel b$, wenn weder $a \rightarrow b$ noch $b \rightarrow a$ gilt.

Happened-Before-Relation ist transitiv: Wenn $a \rightarrow b$ und $b \rightarrow c$ gelten, dann gilt auch $a \rightarrow c$.

14.1.3 Lamport-Zeitstempel

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



Lamport Zeitstempel

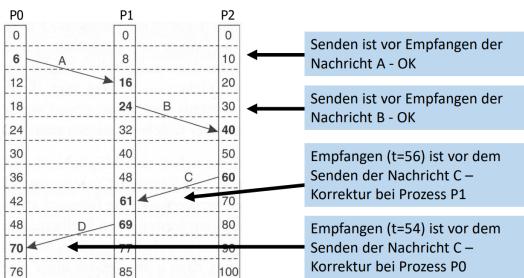
- Ein Prozess sendet eine Nachricht mit Zeitstempel (eigene Zeit) an einen anderen Prozess.

- Einem Ereignis a wird ein Zeitwert $C(a)$ zugeordnet.

- Ein Prozess sendet eine Nachricht mit Zeitstempel a (eigene Zeit) an einen anderen Prozess, welcher die Nachricht zur eigenen Zeit b empfängt, dann müssen $C(a)$ und $C(b)$ so zugewiesen werden, dass $C(a) < C(b)$ ist.

- Die Zeit C muss immer vorwärts laufen.
- Korrekturen können durch Addition von positiven Werten vorgenommen werden.

Die Lösung:



Zusätzliche Forderung:

Zwei Ergebnisse dürfen nie zu genau der selben (logischen) Zeit auftreten. Lösung: Zeitstempel um Prozessnummer ergänzen. Damit kann allen Ereignissen in einem verteilten System eine Zeit zugewiesen werden, die folgenden Bedingungen erfüllt.

- wenn a im selben Prozess vor b auftritt, gilt $C(a) < C(b)$.
- wenn a und b das Senden und Empfangen einer Nachricht darstellen, gilt: $C(a) < C(b)$.
- für alle anderen Ereignisse a und b, gilt $C(a) \neq C(b)$.

Lamport Zeit - Eigenschaften

- Lamports Uhren erfüllen die Uhrenbedingung: $a \rightarrow b \Rightarrow C(a) < C(b)$. Wenn Ereignis a vor Ereignis b stattfindet, dann ist der Zeitstempel von $C(a)$ kleiner als der von $C(b)$.

- Die logischen Lamport-Zeitstempel definieren daher eine partielle Ordnung auf der Menge der Ereignisse, die den kausalen Zusammenhang zwischen Ereignissen erhält.
- Ergänzung zu einer totalen Ordnung ist wieder möglich.

Anhand der Zeitstempel selbst lässt sich nicht immer sicher sagen, ob zwei Ereignisse kausal voneinander abhängen. Hierfür müsste auch die Umkehrung der Uhrenbedingung gelten, aber es gilt lediglich $C(a) < C(b) \Rightarrow a \rightarrow b \vee a||b$

14.1.4 Vektorzeitstempel

Ein Vektorzeitstempel VT(a), der einem Ereignis a zugewiesen wurde, der ein Ereignis a zugewiesen wurde, hat die Eigenschaft, dass Ereignis a dem Ereignis b kausal vorausgeht, wenn $VT(a) < VT(b)$ für ein Ereignis b gilt.

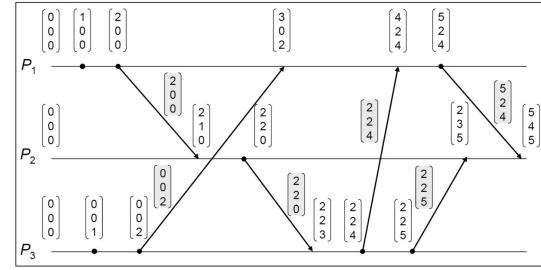
Jeder Prozess P_i besitzt einen Vektor V_i , der für jeden Prozess im System die Anzahl der Ereignisse enthält mit den Eigenschaften:

- $V_i[i]$ ist die Anzahl der Ereignisse, die bisher in P_i aufgetreten sind
- wenn gilt $V_i[j] = k$, erkennt P_i dass in P_j k Ereignisse aufgetreten sind.

Der Vektor V_i wird den gesendeten Nachrichten mitgegeben.

Algorithmus: Vektorzeitstempel

- Jeder Prozess P_i hält einen Vektor V_i bestehend aus n Zählern ($n =$ Anzahl der Prozesse im System).
- Initial ist der Vektorzeitstempel jedes Prozesses der Nullvektor.
- Tritt bei Prozess P_i ein Ereignis auf, so inkrementiert er die i-te Komponente seines Vektors.
- Sendet P_i eine Nachricht, so wird die neue Version von V_i mitgeschickt.
- Empfängt P_i eine Nachricht mit Vektor Zeitstempel VT, so bildet er das komponentenweise Maximum von der neuen Version von V_i und von VT.



Der Vektorzeitstempel in der Nachricht informiert Empfänger über

- die Anzahl Ereignisse die in P_i aufgetreten sind.
- wie viele Ereignisse in anderen Prozessen der Nachricht vorausgegangen sind.
- wie viele vorangegangene Ereignisse möglicherweise kausal abhängig sind.

Kausaler Zusammenhang zwischen zwei Ereignissen

Ereignis A ist eine Ursache von Ereignis B:

- wenn der Zähler für jeden Prozess im Zeitstempel VT(A) kleiner oder gleich dem Zähler im Zeitstempel VT(B) für den entsprechenden Prozess
- und für mindestens einen dieser Zähler kleiner ist.

Beispiele:

- $\begin{pmatrix} 2 \\ 0 \\ 0 \end{pmatrix}$ ist Ursache für $\begin{pmatrix} 4 \\ 0 \\ 1 \end{pmatrix}$
- $\begin{pmatrix} 1 \\ 1 \\ 2 \end{pmatrix}$ ist Ursache für $\begin{pmatrix} 1 \\ 2 \\ 3 \end{pmatrix}$
- $\begin{pmatrix} 2 \\ 1 \\ 3 \end{pmatrix}$ ist keine Ursache für $\begin{pmatrix} 1 \\ 3 \\ 4 \end{pmatrix}$

15 Anhang

15.1 ZeroMQ Code Beispiele

15.1.1 Half-Duplex

Client

```
1 public static void main(String[] args) {
2     try (ZContext context = new ZContext()) {
3         LOG.info("Echo client connecting to " + ADDRESS);
4
5         ZMQ.Socket socket = context.createSocket(SocketType.REQ);
6
7         socket.connect(ADDRESS);
8
9         BufferedReader in = new BufferedReader(new InputStreamReader(System.in));
10
11        while (...) {
12            String input = in.readLine();
13            socket.send(input.getBytes(ZMQ.CHARSET));
14            byte[] reply = socket.recv();
15            LOG.info("Received " + new String(reply, ZMQ.CHARSET));
16        }
17
18    } catch (IOException ex) {
19        LOG.error(ex.getMessage());
20    }
21 }
```

Server

```
1 public static final String ADDRESS = "tcp://localhost:5555";
2
3 public static void main(String[] args) {
4     try (ZContext context = new ZContext()) {
5         ZMQ.Socket socket = context.createSocket(SocketType.REP);
6
7         socket.bind(ADDRESS);
8
9         while (...) {
10            byte[] reply = socket.recv();
11            LOG.info("Received message " + new String(reply, ZMQ.CHARSET));
12            socket.send(reply);
13        }
14    }
15 }
```

15.1.2 Data Distribution

Publisher

```
1 public static final String ADDRESS = "tcp://localhost:5555";
2
3 public static void main(String[] args) {
4     try (ZContext context = new ZContext()) {
5         ZMQ.Socket socket = context.createSocket(SocketType.PUB);
6
7         socket.bind(ADDRESS);
8
9         while (...) {
10             long station = (long) Math.floor(Math.random() * 3.0);
11             long temp = 15 + (long) Math.floor(Math.random() * 10.0);
12             String message = "Temp/" + station + "/" + temp;
13             socket.send(message.getBytes(ZMQ.CHARSET));
14             LOG.info("Published '" + message + "'");
15             Thread.sleep(1000);
16         }
17
18     } catch (InterruptedException e) {
19         LOG.info("interrupted"); // cannot occur
20     }
21 }
```

Subscriber

```
1 public static final String ADDRESS = "tcp://localhost:5555";
2
3 public static void main(String[] args) {
4     String topic = args[0];
5     try (ZContext context = new ZContext()) {
6         ZMQ.Socket socket = context.createSocket(SocketType.SUB);
7
8         socket.connect(ADDRESS);
9         socket.subscribe(topic);
10
11         while (...) {
12             byte[] message = socket.recv();
13             LOG.info("Received: " + new String(message, ZMQ.CHARSET));
14         }
15     }
16 }
```

15.1.3 Aufgabenverteilung

Producer

```
1 public static final String ADDRESS = "tcp://localhost:5555";
2
3 public static void main(String[] args) {
4     try (ZContext context = new ZContext()) {
5         LOG.info("Providing tasks on " + ADDRESS);
6         // Socket to talk to server
7         ZMQ.Socket socket = context.createSocket(SocketType.PUSH);
8
9         socket.bind(ADDRESS);
10        int i = 0;
11
12        while (...) {
13            String workPackage = "Work Package #" + ++i;
14            socket.send(workPackage.getBytes(ZMQ.CHARSET));
15            LOG.info("Send task '" + workPackage + "'");
16            Thread.sleep(1000);
17        }
18    } catch (InterruptedException e) {
19        LOG.info("interrupted"); // cannot occur
20    }
21 }
```

Consumer

```
1 public static final String ADDRESS = "tcp://localhost:5555";
2
3 public static void main(String[] args) {
4     try (ZContext context = new ZContext()) {
5         LOG.info("Listening for tasks on " + ADDRESS);
6         // Socket to talk to server
7         ZMQ.Socket socket = context.createSocket(SocketType.PULL);
8
9         socket.connect(ADDRESS);
10
11        while (...) {
12            byte[] bytes = socket.recv();
13            LOG.info("Received task '" + new String(bytes, ZMQ.CHARSET) + "'");
14            Thread.sleep(3000);
15            LOG.info("Processed task '" + new String(bytes, ZMQ.CHARSET) + "'");
16        }
17    } catch (InterruptedException e) {
18        LOG.error("interrupted"); // cannot occur
19    }
20 }
```