

Verteilte Systeme und Komponenten

Christopher Christensen

18.09.17

Inhaltsverzeichnis

1 Entwicklungsprozesse	4
1.1 Versionskontrollsystem	4
1.1.1 Grundlagen	4
1.1.2 Konzeptionelle Unterschiede	4
2 Komponenten	4
2.1 Begriffe und Architekturen	4
2.1.1 Begriffe	5
2.1.2 Nutzen	5
2.1.3 Entwurf mit Komponenten	6
2.1.4 Komponenten (Java)	8
2.1.5 Deployment (Java)	8
2.1.6 Versionierung (Java)	8
2.2 Schnittstellen	9
2.2.1 Begriff und Konzept	9
2.2.2 Dienstleistungsperspektive	10
2.2.3 Schnittstellen-Spezifikation	10
2.3 Modularisierung	13
2.3.1 Module - Konzept	13
3 Verteilte Systeme	14
3.1 Socket Kommunikation	14
3.1.1 Netzwerk Konzepte	14
3.1.2 Host- und IP-Adressen	15
3.1.3 TCP-Socket	17
3.1.4 TCP Server Socket	19
3.1.5 Netzwerk Interface	21

Abbildungsverzeichnis

1	Software Komponenten in UML	5
2	Nutzen von Komponenten	6
3	Import, Export	7
4	Verdrahtung	7
5	Architektur-Muster	7
6	Rollen der Komponenten	8
7	Verantwortlichkeiten bei DbC	10
8	Operational Interface	11
9	Operational Interface	11
10	Application Programmer Interface	12
11	Java Schnittstellen in UML	12
12	Notationsmöglichkeiten UML-2	13
13	Schichten eines Netzwerks	14
14	Kommunikation	14
15	IP-Adressen finden	16
16	Hostnamen finden	16
17	Lebt der Rechner	17
18	Lokale Adresse des Rechners	17
19	Verbindungsaubau	18
20	Daten senden und empfangen	18
21	Lebenszyklus TCP Server	19
22	Einfacher Zeitserver	20
23	Sequenzdiagramm TCP mit Sockets	20
24	Nicht blockierender Echo Server	21
25	Echo Handler - Erzeugung	21
26	Echo Handler - Ausführung	21
27	NICs im Computer	22
28	Liste aller Netzwerk Adapter und Adressen	23
29	Lokaler Zeitserver	23

1 Entwicklungsprozesse

1.1 Versionskontrollsyste

SCM: System, welches zeitliche Entwicklung von Artefakten festhält und jederzeit Rückgriff auf alte Anderungsstände erlaubt (Revision) → gemeinsame Quellen, automatisiertes Merging bei Konflikten, zentrale/verteilte Datenhaltung

1.1.1 Grundlagen

SCM ≠ Backupsystem, sondern für **Nachvollziehbarkeit von Änderungen**

- **checkout:** lokale Arbeitkopie erstellen
- **update:** Änderungen Dritter aktualisieren
- **log:** Bearbeitungsgeschichte ansehen
- **diff:** Revisionen vergleichen
- **commit:** Artefakte in Repository zurückschreiben

Tagging: markieren eines Revisionsstandes mit Namen (z.B. 1.4.3, 1.5.2beta)

Branching: voneinander getrennte Entwicklungswege erstellen (z.B. für Bugfixing, Prototypen, Tests, Experimente, etc.)

In SCM nur **Quell-Artefakte** abspeichern → **Nie** Artefakte die generiert/erzeugt werden können (.class, ./target/**)

1.1.2 Konzeptionelle Unterschiede

Es gibt verschiedene SCMs, welche sich in folgenden Punkten unterscheiden können:

- Zentrale/verteilte Systeme
- Optimistische/Pessimistische Lockverfahren
- Versionierung (Basis: Datei/Verzeichnisstruktur, Änderung)
- Transaktionsunterstützung (ja/nein)
- Zugriffsprotokolle
- Integration Webserver

2 Komponenten

2.1 Begriffe und Architekturen

Software-Komponente = Software-Element, das zu einem bestimmten Komponentenmodell passt und entsprechend einem Composition Standard ohne Änderungen mit anderen Komponenten verknüpft und ausgeführt werden kann.

2.1.1 Begriffe

Wichtige Eigenschaften von Komponenten:

- eigenständige, ausführbare SW-Einheiten (Laufzeit-Sicht), d.h. Subsysteme, Prozesse, Objekte
- über ihre Schnittstellen austauschbar definiert
- lassen sich unabhängig voneinander entwickeln
- kunden-/anwendungsspezifische, bzw. wiederverwendbare Software sowie COTS
- können installiert / deployed werden



Alternative Darstellung:



Abbildung 1: Software Komponenten in UML

Komponenten sind hierarchisch und werden in **System** und **Subsysteme** eingeteilt.

Komponentenmodelle:

- konkrete Ausprägungen des Paradigmas der komponentenbasierten Entwicklung
- genaue Form, Eigenschaften einer Komponente, Interaction-Standard und Composition Standard festlegen
- kann Implementierungen verschiedener Hersteller besitzen

Folgende Komponentenmodelle sind bekannt: MS .NET, Enterprise Java Beans, DCOM, CORBA Component Model, OSGI Framework

2.1.2 Nutzen

Es gibt verschiedene Nutzen von Komponenten, wie z.B.:

- Packaging (Reuse Benefits)
- Service (Interface Benefits)
- Integrity (Replacement Benefits)

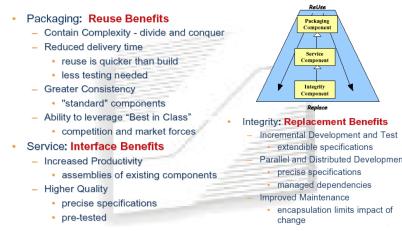


Abbildung 2: Nutzen von Komponenten

2.1.3 Entwurf mit Komponenten

Die Konstruktion von Software aus vorgegebenen Komponenten und Bibliotheken spielt eine immer grösse Rolle, da die Komplexität von Systemen, Protokollen und Anwendungsszenarien anwächst und damit eine Eigenentwicklung aus Gründen der Wirtschaftlichkeit und Sicherheit nicht ratsam ist.

Komponenten zu entwickeln, stellt hierbei noch höhere Anforderungen an unsere softwaretechnischen Fertigkeiten als reguläre Software, da eine ganze Reihe neuer Einflüsse beachtet werden müssen, die bei einmaliger Anwendung nicht auftreten.

Praktische Eigenschaften von Komponenten:

- Wer eine Komponente einsetzen will, braucht nur deren Schnittstelle zu kennen
- Komponenten, die dieselbe Schnittstelle haben, sind gegenseitig austauschbar
- Komponententest ist blackbox Test
- Komponenten lassen sich unabhängig voneinander entwickeln
- Komponenten unterstützen die Wiederverwendbarkeit

Komponenten-Spezifikation:

An der Prüfung wird nach den Komponenten-Spezifikationen gefragt.

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

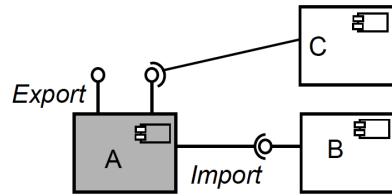


Abbildung 3: Import, Export

Verdrahtung von Komponenten: Wenn der Client keine Person ist, geht dieses Szenario nicht (Klassenspezifikation). Wenn die Abhängigkeiten aussagen, dass Client eine Person ist, dann geht es.

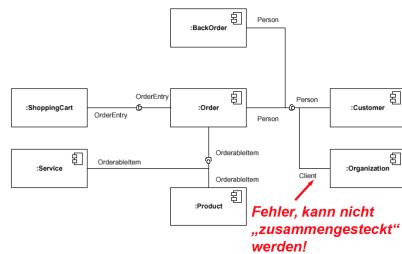


Abbildung 4: Verdrahtung

Architektur-Muster:

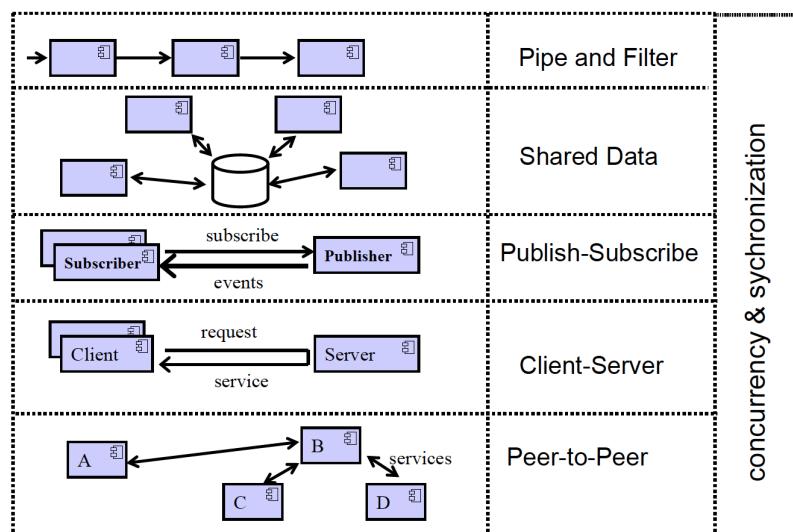


Abbildung 5: Architektur-Muster

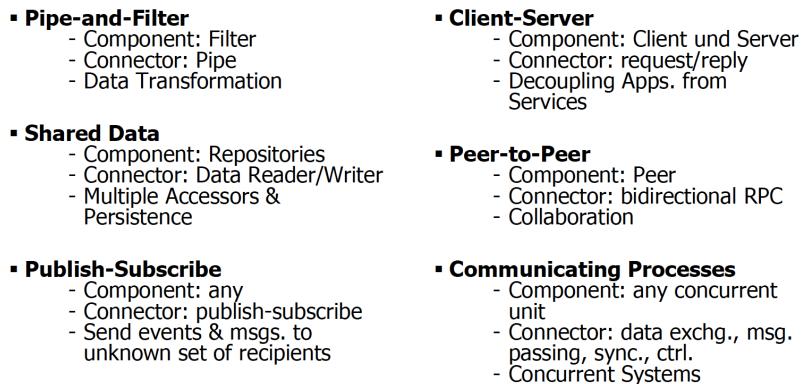


Abbildung 6: Rollen der Komponenten

2.1.4 Komponenten (Java)

Folgende Faustregeln gelten für die Komponenten in Java:

- Komponenten als Klassen implementiert
- Komponenten können sich an Java Beans Spezifikation (JBS) halten
- JBS: Default-Konstruktor, Setter/Getter, PropertyChange, Vetoable, Serializable, Introspection
- Java EE enthält weitergehende Komponentenmodelle
- <http://www.oracle.com/technetwork/java/javase/documentation/spec-136004.html>

Sollen Komponenten austauschbar sein verwendet man Schnittstellen (über Java-Interface definiert/dokumentiert).

Komponenten implementieren als Klassen die Schnittstellen (dadurch mehrere Implementationen möglich). Die Austauschbarkeit ist über die Referenz auf die Schnittstelle gegeben (GoF). → JDBC API

2.1.5 Deployment (Java)

Das Deployment findet meist in einem JAR statt. Kompilierte Klassen und Interfaces werden als class-Dateien in einem ZIP-Format gespeicher (META-INF/manifest.mf-Datei enthält Meta Info).

Schnittstelle (API) und Implementation werden häufig **getrennt deployed**, damit verschiedene Implementations leichter ausgetauscht werden können.

2.1.6 Versionierung (Java)

Für die Versionierung in Java SE gilt:

- Komponenten unbedingt versionieren (eindeutige Identifizierung)

- Für Java gibt es keine bzw. nur optionale und nicht verbindliche technische Mittel zur Umsetzung
- dreistelliges Versionsschema mit semantischer Bedeutung empfohlen: Major.Minor.Bugfix

Major: nicht rückwärtskompatible Veränderung

Minor: rückwärtskompatible Erweiterung

Bugfix: reine Implementationsänderung / Korrektur

Es gibt drei Varianten zur Versionierung.

- **keine Versionierung:** sehr schlecht!
- **Versionierung im Dateinamen:** JAR-Datei einfach anhand Namens identifiziert (eindeutige Referenz in Klassenpfad möglich). Jedoch einfach Manipulierbar und Anpassung von Klassenpfaden bei Bugfix notwendig
- **Versionierung im manifest.mf:** Manipulation nur durch Änderung in JAR-Datei, Spezifikationskonform, Dateiname ohne Versionsnummer möglich (konstante Klassenpfade). Aber die Version ist nur durch Blick ins JAR

2.2 Schnittstellen

Schnittstellen verbinden entweder, Komponenten untereinander (Programmschnittstellen) oder Komponenten mit dem Benutzer (Benutzerschnittstellen).

2.2.1 Begriff und Konzept

Idee der Schnittstelle:

- Software verständlicher machen
- Abhängigkeiten reduzieren
- Wiederverwendung erleichtern

Die wichtigsten Aspekte für Architekten stecken in den Schnittstellen und Beziehungen zwischen den Komponenten.

Das hat mehrere Gründe:

- Beziehungen ermöglichen die Funktion des Systems
- Keine Komponente kann gewünschte Funktionalität allein bieten
- Komponenten (Teilsysteme) kommunizieren über Schnittstellen
- Spezialisten konzentrieren sich auf ihre (lokalen) Probleme

- Über Schnittstellen findet auch Kommunikation mit Aussenwelt

Kriterien für gute Schnittstellen:

- Minimal (wenig Methoden, Parameter, keine globalen Daten)
- Verständlich
- Gut dokumentiert

2.2.2 Dienstleistungsperspektive

Zusammenspiel der Komponenten wird durch einen Vertrag erreicht (Design by Contract, **DbC**):

- Precondition: Zusicherung, die Aufrufer einzuhalten hat
- Postconditions: Nachbedingungen, die Aufgerufene garantiert
- Invarianten: Bedingung, die Instanzen einer Klasse ab Erzeugung erfüllen müssen

Vertrag kann sich auf Variablen, Parameter, Objektzustände beziehen.

	Nutzer	Anbieter
Precondition:	Nutzer muss sicher stellen, dass Vorbedingungen vor Ausführung einer Methode gelten	<i>Defensives Programmieren: Doppelcheck mit assert</i>
Postcondition:	<i>Defensives Programmieren: Doppelcheck mit assert</i>	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 <i>Defensives Programmieren: Doppelcheck mit assert</i>

Abbildung 7: Verantwortlichkeiten bei DbC

2.2.3 Schnittstellen-Spezifikation

Dokumentation von Schnittstellen:

- Was für Benutzung der Komponente wichtig ist
- Was Programmierer verstehen/beachten muss

Jede Programmschnittstelle definiert Menge von Methoden mit folgenden Eigenschaften:

- Syntax (Rückgabewerte, Argumente, etc.)
- Semantik (Was bewirkt Methode)
- Protokoll

- Nichtfunktionale Eigenschaften (Performance, etc.)

Es gibt verschiedene Interface Types:

- Operational Interface
- Signal Interface
- Stream Interface

Operational Interface: Control- and Data-Flow. Komponente offeriert Services, die durch Interface-Operations geholt werden

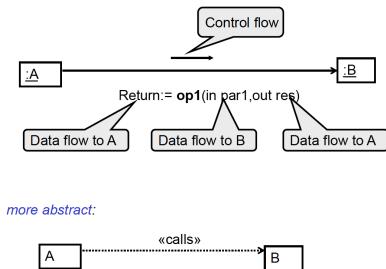


Abbildung 8: Operational Interface

If a client uses one operation, it is likely that it will use some of the other operations too. Sets of such operations belong together and have related effects on the state of the component object

Operational Interface Specification: name, input/output parameters (Info zur/von Komponente), change of state, input/output/state relationships

changeCustomerName (in aCustomer: Customer, in newName: String)	<small>The operation name and parameters</small>
Precondition: aCustomer is a valid Customer <small>The condition under which the operation guarantees that the postcondition will be true.</small>	
Postcondition: the name is changed <small>Specifies what the effect of the operation will be, provided the precondition is true.</small>	

Abbildung 9: Operational Interface

Signal Interface: Signale die vom Komponent gesendet/empfangen werden können → name, out-/ingoing signals

Stream Interface: Collections von Data-Streams, die vom Komponent gebraucht/erstellt werden → name, produced/consumed streams (outgoing/ingoing), quality of service

Schnittstellen in Java:

- Mit Java-Interfaces deklariert
- Zu class-Dateien kompiliert
- Können somit in JAR-Dateien verpackt werden
- Mit JavaDoc dokumentieren
- Schnittstellen an Systemgrenze und zwischen Subsystemen sind architekturentwicklungsrelevant und werden in SysSpec beschrieben
- API: öffentliche Schnittstellen

API (Application Programmer Interface): Spezifiziert Operationen einer Softwarekomponente. Zweck ist Menge an Funktionen unabhängig von Implementierung zu definieren, damit Implementierung variieren kann, ohne die Benutzer der Softwarekomponente zu beeinträchtigen.



Abbildung 10: Application Programmer Interface

Schnittstellen sind Spezifikationen des externen Verhaltens von Klassen (oder anderer Elemente) und beinhalten eine Menge von Signaturen für Operationen sowie Attributen, die Klassen (o.a.), die diese Schnittstelle bereitstellen wollen, implementieren müssen.

Schnittstellen sind mit dem Schlüsselwort «*interface*» gekennzeichnet.



Abb. 2.2-17: *String* realisiert die Schnittstelle *Sortierbar*

Gewöhnliche Klassen, die eine Schnittstelle implementieren wollen, müssen alle in der zugehörigen Schnittstellenklasse definierten Operationen implementieren. Die in der Schnittstelle definierten Operationen sind de facto abstrakte Operationen.

Abbildung 11: Java Schnittstellen in UML

Bereitstellung und Anforderung einer Schnittstelle können über eine Abhängigkeitsbeziehung miteinander verbunden werden.

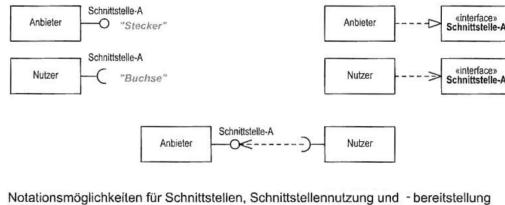


Abbildung 12: Notationsmöglichkeiten UML-2

2.3 Modularisierung

2.3.1 Module - Konzept

Modul: In sich abgeschlossener Teil des gesamten Codes, bestehend aus einer Folge von Verarbeitungsschritten und Datenstrukturen.

Kopplung: Unabhängigkeit der Module → Minimiere Kopplung

Kohäsion: Ausmass der Kommunikation innerhalb eines Moduls → Maximiere Kohäsion

Arten von Modulen:

- Bibliotheken
(Datum-, Trigonometrie-Modul)
- abstrakte Datentypen
(komplexe Zahlen, etc.)
- Modellierung physischer Systeme
(Sensorsystem-Modul, etc.)
- Modellierung logisch-konzeptionelle Systeme
(Grafikmodul, Datenbankmodul)

Kriterien des modularen Entwurfs:

- Zerlegbarkeit
- Kombinierbarkeit
- Verständlichkeit
- Stetigkeit

3 Verteilte Systeme

3.1 Socket Kommunikation

In diesem Kapitel werden Netzwerkkonzepte zur Java-Netzwerkprogrammierung erklärt.

3.1.1 Netzwerk Konzepte

Hier ist eine erste Begriffsliste:

- **Host:** ein am Netzwerk angeschlossener Rechner
- jeder Host bekommt eine **IP-Adresse**
- Versionen von IP-Adressen: **IPv4** (32bit), **IPv6** (128bit)
- Statt IP-Nummern werden **Hostnamen** verwendet
- **Domain Name Service (DNS):** übernimmt Zuordnung zw. Name und IP-Adresse

Die Internet-Kommunikation benutzt das Vier-Schichten-Modell: **Applikationsschicht**, **Transportschicht**, **Internetschicht**, **Netzwerkschicht**.

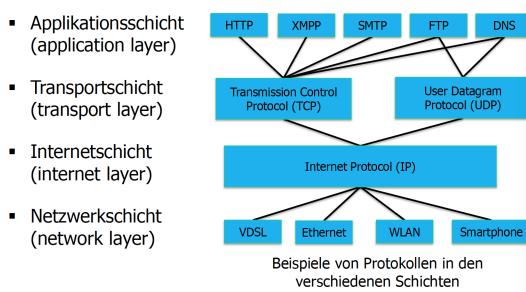


Abbildung 13: Schichten eines Netzwerks

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

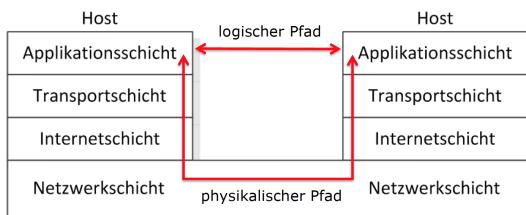


Abbildung 14: Kommunikation

Die **IP-Adresse** wird in der Internetschicht benötigt. Kein Host im Internet hat die gleiche IP-Adresse.

Die **Port-Nummer** wird in der Transportschicht benötigt, welche die empfangenen Daten an die korrekte Applikation weiterleitet.

Eine IP-Adresse zusammen mit der Port-Nummer spezifiziert einen Kommunikationsendpunkt (**Socket**).

Die beiden wichtigsten Protokolle der Transportschicht sind:

- **Transmission Control Protocol (TCP).**

Es ist ein zuverlässiges, verbindungsorientiertes, Bytestrom Protokoll. Seine Hauptaufgabe ist die Bereitstellung eines sicheren Transports von Daten durch das Netzwerk.

- **User Datagram Protocol (UDP).** Es ist ein unzuverlässiges, verbindungsloses Protokoll. Unzuverlässig heisst nicht, dass die Daten evtl. fehlerhaft beim Ziel-Host ankommen, sondern das Protokoll garantiert nicht, dass die Daten auch tatsächlich beim Ziel-Host ankommen.

Es gibt zwei Arten von Teilnehmern in einem Netzwerk:

- **Server (dt. Diener)**

Er ist ein Dienstleister, stellt in einem Computersystem Daten oder Ressourcen zur Verfügung. Das Computersystem kann dabei aus einem einzelnen Computer oder einem Netzwerk mehrerer Computer bestehen. Der Begriff ist mehrdeutig:

Server-Programm: Ein Computerprogramm, das einen Dienst (z.B. Netzwerkprotokoll) implementiert.

Server-Computer: Der Computer auf dem ein oder mehrere Server-Programme laufen.

- **Client (dt. Kunde)**

Er ist ein Dienstnehmer, der in einem Computersystem Dienste von Servern benutzt.

3.1.2 Host- und IP-Adressen

Der Datenaustausch im Internet geschieht durch kleine IP-Pakete. Der Empfänger der Pakete wird durch eine Kennung, die numerische IP-Adresse, identifiziert. Diese Zahl ist schwer zu behalten, weshalb oft der Hostname verwendet wird. DNS übernimmt die Konvertierung von Hostnamen in IP-Adressen. Auch in Java kann zu einem Hostnamen die IP-Adresse erfragt werden und auch umgekehrt zu der IP-Adresse der Hostnamen.

Java sieht das Internet durch sein API. Mit dem **import** von **java.net.*** werden Interfaces und Klassen zu Netzwerkprogrammierung verfügbar. → **InetAddress**, **Socket**, **ServerSocket**, **SocketAddress**

Klasse **InetAddress**:

- **static InetAdress getByName(String host)**
Liefert die IP-Adresse eines Hosts anhand des Namens. Der Hostname kann als Maschinenname angegeben sein oder als numerische Repräsentation der IP-Adresse.
- **String getHostName()**
Liefert den Hostnamen.
- **String getHostAddress()**
Liefert die IP-Adresse als String im Format %d.%d.%d.%d.
- **String getCanonicalHostName()**
Liefert den vollständig qualifizierten Domänennamen (FQDN), sofern das möglich ist.

IP-Adressen finden

```
public class DemoInetAddress {
    private static final Logger LOG =
        LogManager.getLogger(DemoInetAddress.class);

    public static void main(final String[] args) {
        try {
            final InetAddress inet =
                InetAddress.getByName("www.alumnihslu.ch");
            LOG.info(inet.getCanonicalHostName());
            LOG.info(inet.getHostAddress());
            LOG.info(inet.getHostName());
            LOG.info(inet.toString());
        } catch (UnknownHostException ex) {
            LOG.debug(ex.getMessage());
        }
    }
}
```

gee.hslu.ch
147.88.201.128
www.alumnihslu.ch
www.alumnihslu.ch/147.88.201.128

Abbildung 15: IP-Adressen finden

Hostnamen finden

```
public class DemoInetName {
    private static final org.apache.logging.log4j.Logger LOG =
        LogManager.getLogger(DemoInetName.class);

    public static void main(final String[] args) {
        try {
            final InetAddress inet =
                InetAddress.getByName("147.88.201.156");
            LOG.info(inet.getCanonicalHostName());
            LOG.info(inet.getHostAddress());
            LOG.info(inet.getHostName());
            LOG.info(inet.toString());
        } catch (UnknownHostException ex) {
            LOG.debug(ex.getMessage());
        }
    }
}
```

www.hochschule-luzern.ch
147.88.201.156
www.hochschule-luzern.ch
www.hochschule-luzern.ch/147.88.201.156

Abbildung 16: Hostnamen finden

Das InetAddress-Objekt kann durch Versenden von Test-Paketen überprüfen, ob der Host im Internet "lebt". Dazu dient die Methode **isReachable(int msec)**.

Das Argument ist eine Anzahl von Millisekunden, die man dem Prüfvorgang geben will.

```
...
InetAddress.getByName("193.99.144.71").isReachable(2000);    true
InetAddress.getByName("100.100.100.100").isReachable(2000); false
...
```

Abbildung 17: Lebt der Rechner

Um Probleme mit der Namensauflösung auszuschliessen, sollte die IP-Adresse verwendet werden.

Klasse **InetAddress** besitzt die Methode **getLocalHost**, um den Hostnamen und die IP-Adressen des eigenen Rechner zu ermitteln.

Die Methode **isSiteLocalAddress** liefert **true**, wenn das InetAddress-Objekt für das eigene Netzwerk steht.

```
...
inet.getLocalHost();          z.B. dell/192.168.2.138
InetAddress.getByName("192.168.1.34").isSiteLocalAddress(); true
...
```

Abbildung 18: Lokale Adresse des Rechners

Läuft ein Java-Programm und wird die IP-Adresse des Rechners geändert, registriert Java diese Änderung nicht. Das liegt daran, dass alle IP-Adressen und zugehörigen Host-Adressen in einem internen Cache gehalten werden und nicht direkt beim Betriebssystem angefragt werden.

3.1.3 TCP-Socket

Socket (Kommunikationsendpunkt in TCP/IP-Netzwerk) dient zur Abstraktion.

Um Daten verbindungsorientiert zu versenden, sind folgende Aktionen nötig:

- Socket erzeugen
- Socket an einen lokalen Port binden (serverseitig)
- Verbindung mit Zieladresse herstellen
- Daten über Socket lesen/schreiben
- Socket schliessen

Verbindungsaufbau

```

public class PortSearch {
    public static void main(final String[] args) {
        final String host = "localhost";
        final ExecutorService executor =
            Executors.newCachedThreadPool();
        for (int numb = 1; numb <= 65535; numb++) {
            final int port = numb;
            executor.execute(() -> {
                try {
                    final Socket theSocket = new Socket(host, port);
                    LOG.info("Port " + port + " from Host " + host +
                            " knows TCP.");
                } catch (Exception ex) {
                    LOG.debug("Exception : " + ex.getMessage());
                }
            });
        }
    }
}

```

Programm wartet bis eine Verbindung hergestellt ist oder das Timeout eine Exception wirft.

Wieso wird der Verbindungsauflauf in einen Thread ausgelagert?

Abbildung 19: Verbindungsauflauf

Die Portsuche erfolgt folgendermassen:

- Programm nutzt Socket-Konstruktor zur Überprüfung, ob Portnummern frei oder benutzt sind
- Socket-Konstruktor versucht Verbindung mit Zieladresse herzustellen, definiert durch Argumente **host** und **port**
- IP-Adresse hier **localhost**
- Kann verbunden werden, ist Port benutzt, ansonsten wirft Socket-Konstruktor **IOException**

```

public class WhoIsThis {
    private static final Logger LOG = LogManager.get
        Ein- und Ausgabe-
        Streams werden mit
        dem Socket verknüpft.
    public static void main(final String[] args) {
        final String host = "whois.nic.ch";
        try (Socket client = new Socket(host, 43)) {
            final PrintWriter outStream = new PrintWriter(
                client.getOutputStream());
            final BufferedReader inStream = new BufferedReader(
                new InputStreamReader(client.getInputStream()));
            outStream.println("hslu.ch");
            outStream.flush();
            String line;
            while ((line = inStream.readLine()) != null) {
                LOG.info(line);
            }
        } catch (Exception e) {
            LOG.debug(e.getMessage());
        }
    }
}

```

Ein- und Ausgabe-Streams werden mit dem Socket verknüpft.

Daten werden dem Server gesendet.

Daten werden vom Server gelesen.

Abbildung 20: Daten senden und empfangen

Das Programm (Abbildung 20) nutzt Dienst **whois** um Infos über eine bestimmte Internet-Adresse zu bekommen. Der Socket-Konstruktor stellt Verbindung mit dem Hostnamen **whois.nic.ch** und dem Port 43 her.

Mit Hilfe der Socket-Methoden **getOutputStream** und **getInputStream** wird **PrintWriter** und **BufferedReader** mit Socket verknüpft. Dem Dienst **whois** wird die gewünschte Internet-Adresse plus Zeilenumbruch gesendet. Der Dienst **whois** sendet die Informationen der Internet-Adresse als Text zurück. → Socket und Streams werden geschlossen!

3.1.4 TCP Server Socket

Server bauen keine eigene Verbindungen auf, sondern horchen an ihrem zugewiesenen Port auf Anfragen. → **ServerSocket**

Konstruktor bekommt als Argument Portnummer, zu der sich Clients verbinden.

Wichtig:

- Portnummer darf nicht vergeben sein
- bei Unix können nur **root**-Besitzer Portnummer unter 1024 nutzen
- bei Windows können alle Portnummer nutzen

Lebenszyklus TCP Server

1. Server-Socket erzeugen
2. Mit **accept** auf Verbindung warten
3. Ein-/Ausabestrom mit erhaltenen Socket verknüpfen
4. Daten lesen/schreiben, entsprechend vereinbarten Protokoll
5. Stream (Client) und Socket schliessen
6. Bei Schritt 2 weitermachen oder Server-Socket schliessen

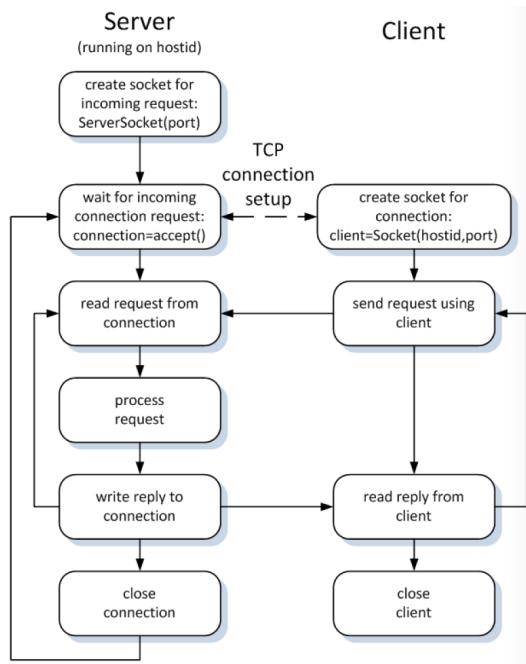


Abbildung 21: Lebenszyklus TCP Server

Zeitserver (Abbildung 22)

Programm erstellt Server Socket und bindet an Port 13 (Daytime Dienst). Daytime ist einfaches ASCII-basiertes Netzwerkprotokoll.

Mit **accept** wird auf neue Verbindung gewartet (Programm blockiert). Wird Verbindung akzeptiert, gibt **accept** ein Socket-Objekt zurück.

DataOutputStream wird mit Socket-Objekt verknüpft. Date-Objekt wird erzeugt und aktuelle Datum und Zeit werden über Ausgabestrom an Client gesendet. Socket und damit auch Streams werden geschlossen.

```
public static void main(final String[] args) {
    try {
        final ServerSocket listen = new ServerSocket(13);
        Wieso genau Port 13?
        Warten auf Client
        while (true) {
            try (final Socket client = listen.accept()) {
                final DataOutputStream dout =
                    new DataOutputStream(client.getOutputStream());
                final Date date = new Date();
                dout.write((date.toString()).getBytes());
            }
        } catch (IOException ex) {
            LOG.debug(ex.getMessage());
        }
    }
}
```

Zeit wird dem Client gesendet.

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

Abbildung 22: Einfacher Zeitserver

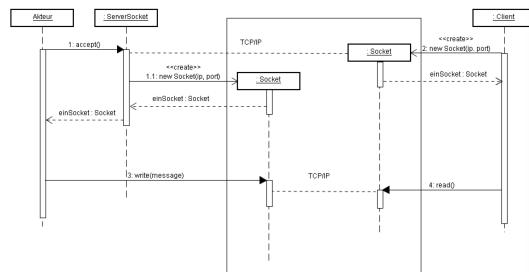


Abbildung 23: Sequenzdiagramm TCP mit Sockets

Nur **accept** (blockierend) nimmt wartende Verbindung an (nur eine). Kommunikation läuft über von **accept** temporär zurückgegebenen Socket. Damit Server schnell wieder verfügbar, muss Programm so schnell wie möglich zu **accept** zurückkehren. → Eigentliche Dienst und Kommunikation mit Client daher in Thread auslagern!

```

public class EchoServer {
    private static final Logger LOG = LogManager.getLogger(EchoServer.class);

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

Das Programm kann sofort auf den nächsten Verbindungsauflauf warten.

```

Executor für das parallele Handling.

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

Der Echo Handler wird in einem eigenen Thread ausgeführt.

Abbildung 24: Nicht blockierender Echo Server

Echo Server erstellt Echo Handler Objekt und übergibt Socket zur Client-Verbindung. Sobald Executor den Echo Handler mit Thread gestartet hat, läuft Handler unabhängig von anderen Threads. Client bestimmt Ende des Echo Handlings.

```

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

    public EchoHandler(final Socket client) {
        this.client = client;
    }
}

Übergabe des Client Socket an den Echo Handler

```

Abbildung 25: Echo Handler - Erzeugung

```

@Override
public void run() {
    LOG.info("Connection to " + client);
    try (OutputStream out = client.getOutputStream();
         InputStream in = client.getInputStream()) {
        int data = in.read();
        while (data != -1) {
            out.write(data);
            System.out.print((char) data);
            data = in.read();
        }
        out.flush();
    } catch (IOException ex) {
        LOG.debug(ex.getMessage());
    }
}

Das Schliessen des Input-/OutputStreams schliesst auch den Socket.

```

Wartet auf Daten vom Client.

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

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

Abbildung 26: Echo Handler - Ausführung

3.1.5 Netzwerk Interface

Netzwerk Interface = Verbindung zwischen Computer und privat/öffentlichen Netzwerk. → **Netzwerk Adapter NIC** (network interface card), nicht

zwingend physisch (auch Software)

Rechner besitzen mehr als ein Netzwerk Adapter: LAN, WLAN, virtuelles Netzwerk.

Server muss wissen, auf welchem Netzwerk Adapter er auf Verbindungen warten soll. Beim Erstellen des Server Sockets oder Datagramm Sockets muss der Netzwerk Adapter angegeben werden.

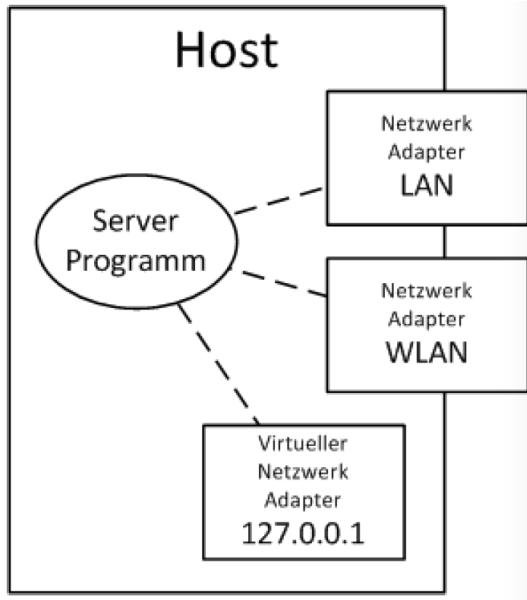


Abbildung 27: NICs im Computer

Die Klasse **NetworkInterface** repräsentiert eine Netzwerkschnittstelle und mit ihr einen **Namen** und eine Reihe von **IP-Adressen**. Um diese Information vom Host auszulesen stehen die folgenden Methoden zur Verfügung:

- **getInetAddresses**, die eine Enumeration von **InetAddress**-Objekten zurückgibt
- **getInterfaceAddresses**, die eine Liste von **InterfaceAddress**-Objekten zurückgibt

```

public static void main(final String[] args) throws SocketException {
    Enumeration<NetworkInterface> nets =
        NetworkInterface.getNetworkInterfaces();
    for (NetworkInterface netint : Collections.list(nets)) {
        displayInterfaceInformation(netint);
    }
}

static void displayInterfaceInformation(NetworkInterface netint)
    throws SocketException {
    LOG.info("Display name: " + netint.getDisplayName());
    LOG.info("Name: " + netint.getName());
    Enumeration<InetAddress> inetAddresses = netint.getInetAddresses();
    Collections.list(inetAddresses).stream().forEach((inetAddress) -> {
        LOG.info("InetAddress: " + inetAddress);
    });
    LOG.info(" ");
}

```

Abbildung 28: Liste aller Netzwerk Adapter und Adressen

Um einen Server Socket einem bestimmten Netzwerk Adapter zuzuordnen, stehen überladene Konstruktoren zur Verfügung. Den Konstruktoren des Server Sockets wird neben der Portnummer auch die IP-Adresse des Netzwerk Adapter übergeben. Zudem muss dem Server Socket noch die Grösse (backlog) der Warteschlange für anstehende Clients mitgegeben werden.

```
ServerSocket(final int port, final int backlog, final InetAddress bindAddr) throws
```

```

public static void main(final String[] args) {
    try {
        final InetAddress inet = InetAddress.getByName("localhost");
        final ServerSocket listen = new ServerSocket(13, 10, inet);
        while (true) {
            try (final Socket client = listen.accept()) {
                final DataOutputStream dout =
                    new DataOutputStream(client.getOutputStream());
                final Date date = new Date();
                dout.write((date.toString()).getBytes());
            }
        } catch (IOException ex) {
            LOG.debug(ex.getMessage());
        }
    }
}

```

IP-Adresse mit «localhost» erstellen.
 Server Socket mit Port, IP-Adresse und Backlog (10 Clients) erstellen.
 Java Code wie beim SimpleDayTimeServer

Abbildung 29: Lokaler Zeitserver

Das Programm **LocalDayTimeServer** erstellt einen Server Socket, bindet diesen an den Port 13 (Daytime Dienst) und verknüpft den Server Socket mit dem virtuellen Loopback Interface **localhost** mit der IP-Adresse **127.0.0.1** (IPv6 **::1**).

Dieser virtuelle Netzwerk Adapter hat jedoch keinerlei Verbindung zu real vorhandenen Netzwerkschnittstellen. Damit ist der Daytime Server nicht von aussern ansprechbar. → Loopback Interface fürs Testen von Servern.