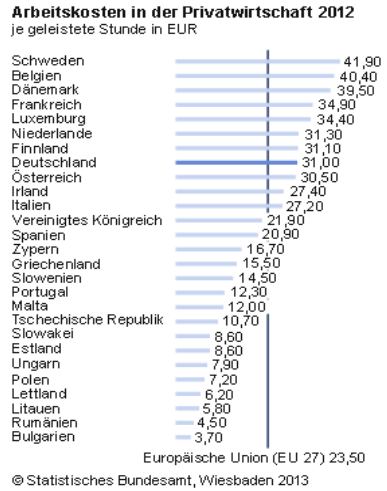
22. Kommunikation mit Iteratoren, Kanälen und Konnektoren

Prof. Dr. Uwe Aßmann
Lehrstuhl Softwaretechnologie
Fakultät für Informatik
Technische Universität Dresden
Version 13-1.3, 20.04.12

1) Channels
   1) Entwurfsmuster Iterator (Stream)
   2) Entwurfsmuster Sink
   3) Channel
2) I/O und Persistente Datenhaltung
3) Ereigniskanäle
4) Konnektoren

Zu den Kosten der Arbeit

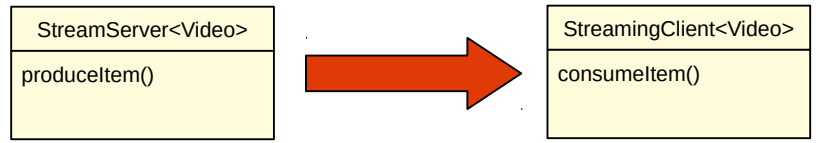


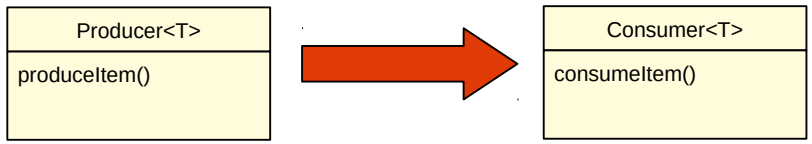
http://www.heise.de/resale/imgs/17/1/0/0/1/3/4/1/ArbeitskostenEULaenderStart2012-9bb2e8b041f1342e.png

22.1 Entwurfsmuster Channel

Wie organisiere ich die Kommunikation zweier Aktoren?

- Wie repräsentiert man potentiell unendliche Collections?

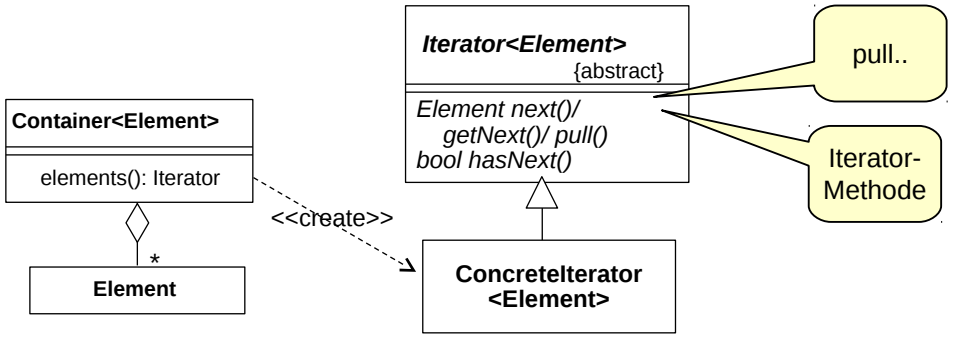




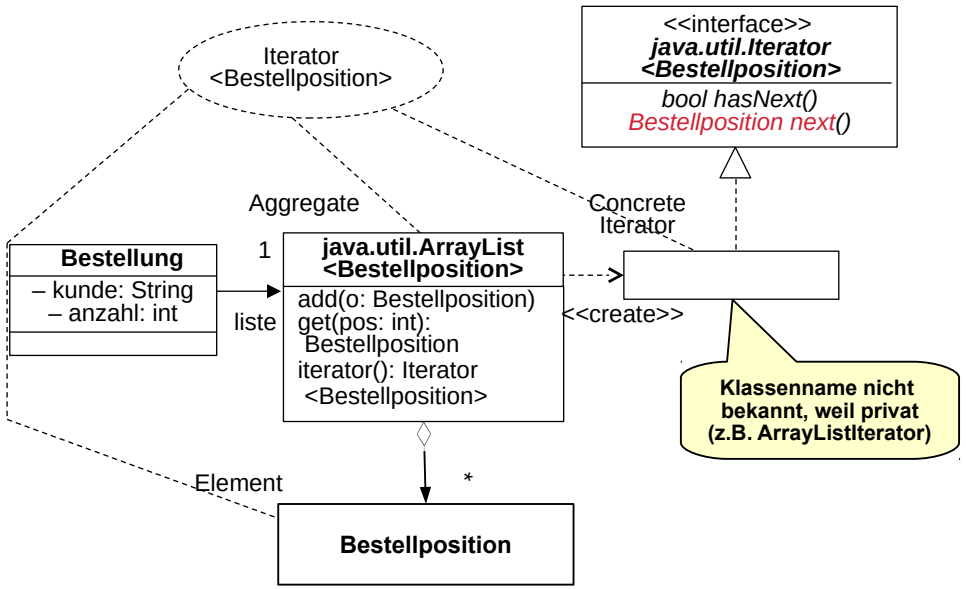
22.1.1 Entwurfsmuster Iterator (Eingabestrom, input stream)

Entwurfsmuster Iterator (Input Stream) (Implementierungsmuster)

- Ein **Eingabestrom (input stream)** ist eine potentiell unendliche Folge von Objekten (zeitliche Anordnung einer pot. unendlichen Folge)
- Entwurfsmuster **Iterator** (auch: **input stream**, Cursor, Enumeration, pull-socket)
  - Sequentielles, polymorphes Durchlaufen der Elemente eines strukturieren Objekts oder einer Collection (Navigation). Typisierung des Inhalts (der Elemente)
  - Aufzählen der in einem "Behälter" befindlichen Elemente durch *Herausziehen (pull)* mit Hilfe einer *Iteratormethode (next, getNext, pull)*
  - Keine Aussage über die Reihenfolge im Container möglich
  - Merken des Zustandes der Navigation



Iterator-Beispiel in der JDK (ArrayList)



Iterator-Implementierungsmuster

- Verwendungsbeispiel:

```
T thing;
List<T> list;
..
Iterator<T> i = list.iterator();
while (i.hasNext()) {
      doSomeThing(i.next());
}
```

- Einsatzzwecke:
  - Iteratoren und Iteratormethoden erlauben, die Elemente einer komplexten Datenstruktur nach außen hin bekannt zu geben, aber ihre Struktur zu verbergen (Geheimnisprinzip)
  - Sie erlauben **bedarfsgesteuerte Berechnungen (processing on demand, lazy processing),** weil sie nicht alle Objekte der komplexen Datenstruktur herausgeben, sondern nur die, die wirklich benötigt werden
  - Iteratoren können "unendliche" Datenstrukturen repräsentieren, z.B. unendliche Mengen wie die natürlichen Zahlen, oder "unendliche" Graphen wie die Karte der Welt in einem Navigator

Anwendungsbeispiel mit Iteratoren


```
import java.util.Iterator;
...
class Bestellung {

  private String kunde;
  private List<Bestellposition> liste;

  ...
  public int auftragssumme() {
    Iterator<Bestellposition> i = liste.iterator();
    int s = 0;
    while (i.hasNext())
      s += i.next().positionspreis();
    return s;
  }
  ...
}
```

Online: Bestellung2.java

Iterator-Implementierungsmuster in modernen Sprachen

- In vielen Programmiersprachen (Sather, Scala) stehen **Iteratormethoden (stream methods)** als spezielle Prozeduren zur Verfügung, die die Unterobjekte eines Objekts liefern können
  - Die yield-Anweisung gibt aus der Prozedur die Elemente zurück
  - Iterator-Prozedur kann mehrfach aufgerufen werden
  - Beim letzten Mal liefert sie null

```
class bigObject {
  private List subObjects;
  public iterator Object deliverThem() {
    while (i in subObjects) {
      yield i;
       // Dieser Punkt im Ablauf wird sich als Zustand gemerkt
       // Beim nächsten Aufruf wird hier fortgesetzt
    }
  }
}
.. BigObject bo = new BigObject(); ...
..  a = bo.deliverThem();
    b = bo.deliverThem();..
```


For-Schleifen auf Iterable-Prädikatschnittstellen

- Erbt eine Klasse von `Iterable`, kann sie in einer *vereinfachten for-Schleife* benutzt werden
- Typisches Implementierungsmuster

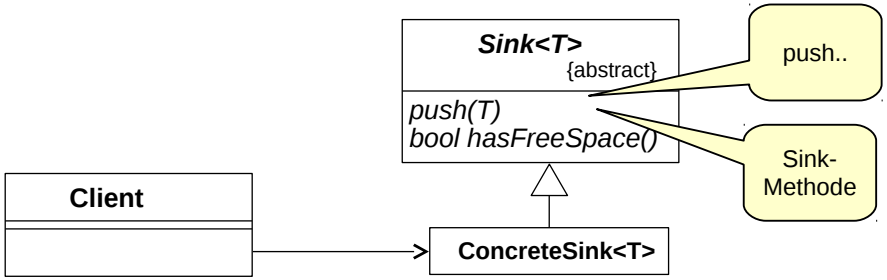
```
class BillItem extends Iterable {
  int price; }
class Bill {
  int sum = 0;
  private List billItems;
  public void sumUp() {
  for (BillItem item: billItems) {
    sum += item.price;
  }
  return sum;
  }
}
```

```
class BillItem { int price; }
class Bill {
  int sum = 0;
  private List billItems;
  public void sumUp() {
  for (Iterator i = billItems.iterator();
       i.hasNext(); ) {
    item = i.next();
    sum += item.price;
  }
  return sum;
  }
}
```

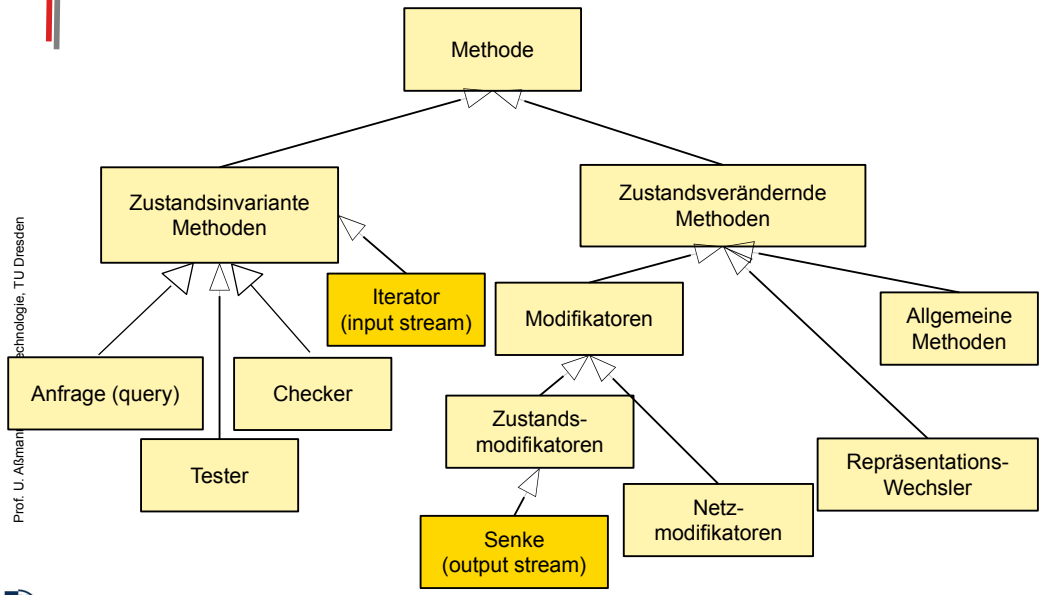

22.1.2 Senken (Sinks)

Entwurfsmuster Senke (Implementierungsmuster)

- Name: **Senke** (auch: Ablage, sink, output stream, belt, push-socket)
- Problem: Ablage eines beliebig großen Datenstromes.
  - push
  - ggf. mit Abfrage, ob noch freier Platz in der Ablage vorhanden
- Lösung:



Erweiterung: Begriffshierarchie der Methodenarten

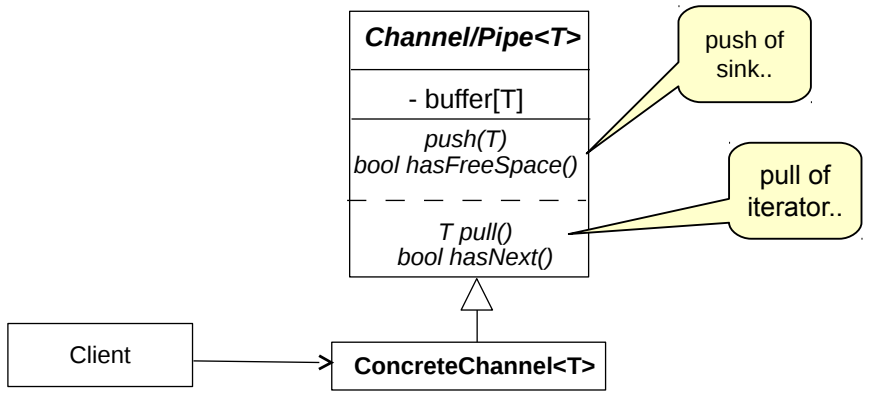


22.1.3 Channels (Pipes)

- Die Kombination aus Senken und Iteratoren, ggf. mit beliebig großem Datenspeicher

Entwurfsmuster Channel und Pipe (Implementierungsmuster)

- Name: **Channel** (Kanal, duplex-stream). Wir sprechen von einer **Pipe (Puffer, buffer)**, wenn die Kapazität des Kanals endlich ist, d.h. `hasFreeSpace()` irgendwann false liefert
- Zweck: asynchrone Kommunikation zwischen zwei Komponenten durch Pufferung eines beliebig großen Datenstromes mit Hilfe eines Puffers buffer
  - Kombination von push- und pull-Methoden (Iterator- und Ablagemethoden)
  - ggf. mit Abfrage, ob noch freier Platz in der Ablage vorhanden
  - ggf. mit Abfrage, ob noch Daten im Kanal vorhanden

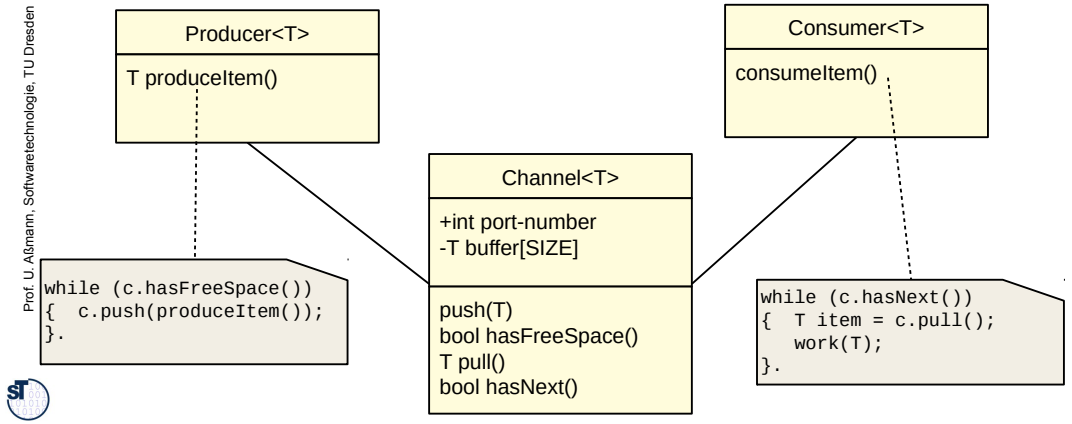


Channels in anderen Programmiersprachen

- Channels (pipes) kommen in vielen Sprachen als Konstrukte vor
  - Shell-Skripte (Operator für pipes: “|”)
  - Communicating Sequential Processes (CSP, Hoare):
    - Operator für pull: “?”
    - Operator für push: “!”
  - C++: Eingabe- und Ausgabestream stdin, stdout, stderr
    - Operatoren “<<” (read) und “>>” (write)
  - Architectural Description Languages (ADL, Kurs CBSE)
- Sie sind ein elementares Muster für die Kommunikation von parallelen Prozessen (producer-consumer-Muster)

Wie organisiere ich die Kommunikation zweier Aktoren?

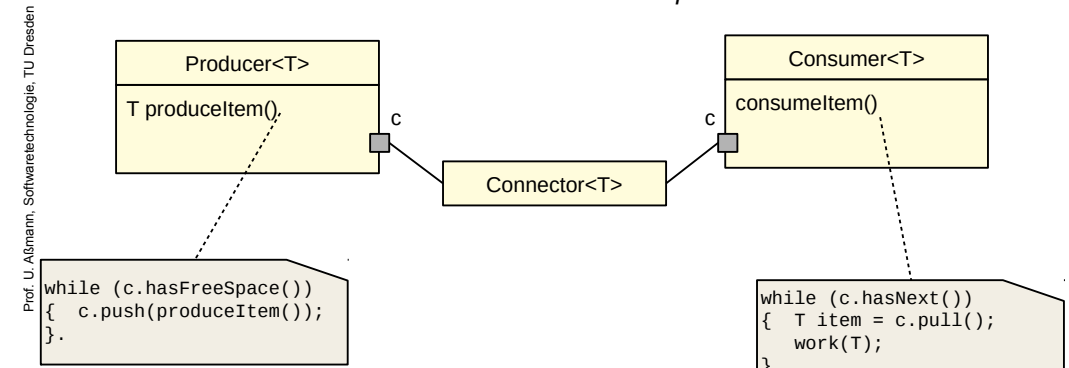
- Einsatzzweck: Wie repräsentiert man potentiell unendliche Mengen?
- Bsp.: Pipes mit ihren Endpunkten (Sockets) organisieren den Verkehr auf den Internet; sie bilden Kanäle zur Kommunikation zwischen Prozessen (Producer-Consumer-Muster)
- Einsatzzweck: Ein ***Aktor*** ist ein parallel arbeitendes Objekt. Zwei Aktoren können mit Hilfe eines Kanals kommunizieren und lose gekoppelt arbeiten



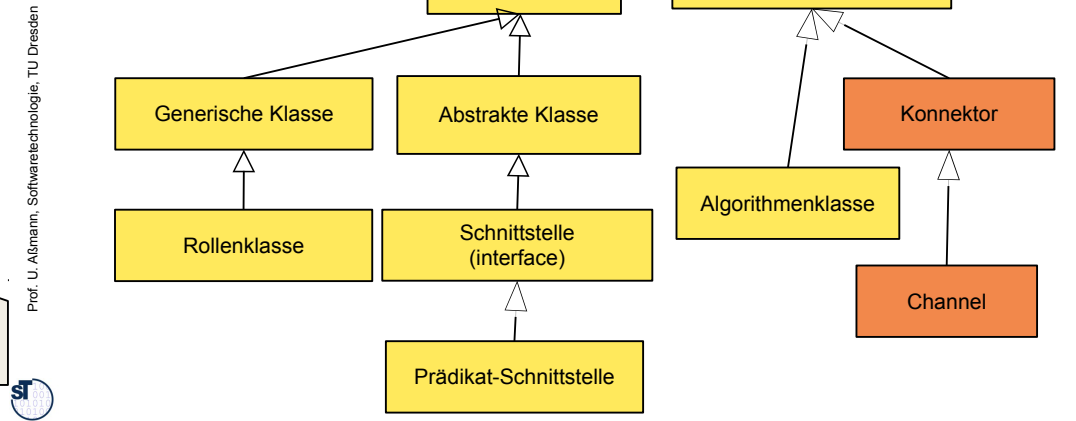
Konnektoren

Def.: Ein **Konnektor** ist eine technische Klasse, die zur Kommunikation von Anwendungsklassen dient.

- Kanäle bilden spezielle Konnektoren
- UML Notation: Andoncken eines Konnektors an *ports*



Begriffshierarchie von Klassen (Erweiterung)

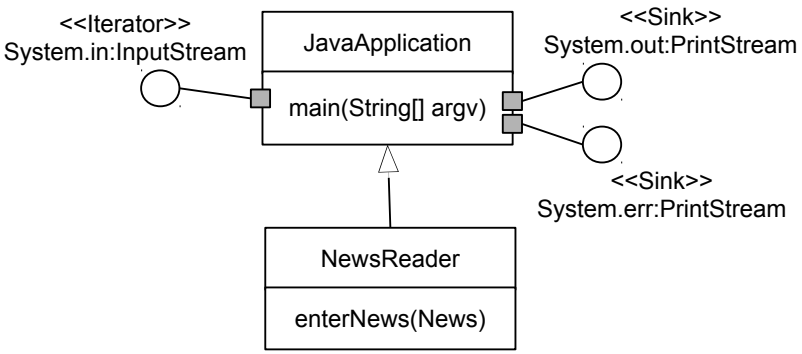


22.2 Input/Output und persistente Datenhaltung

- Das JDK nutzt Iteratoren/Streams an verschiedenen Stellen

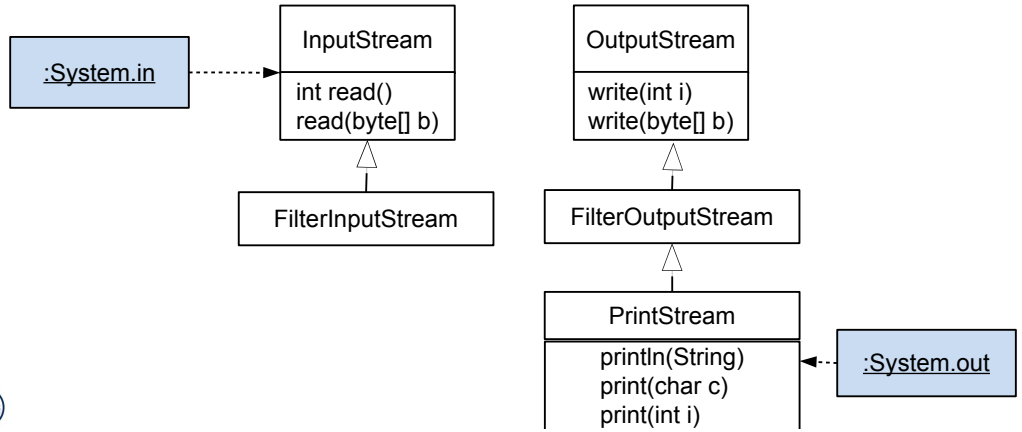
Java-Anwendungen mit ihren Standard-Ein/Ausgabe-Strömen

- Ein Programm in Java hat 3 Standard-Ströme
  - Entwurfsidee stammt aus dem UNIX/Linux-Sytem
- Notation: UML-Komponenten



22.2.1 Ein- und Ausgabe in Java

- Die Klasse **`java.io.InputStream`** stellt einen Iterator/Stream in unserem Sinne dar. Sie enthält Methoden, um Werte einzulesen
- **`java.io.OutputStream`** stellt eine Senke dar. Sie enthält Methoden, um Werte auszugeben
- Die statischen Objekte `in, out, err` bilden die Sinks und Streams in und aus einem Programm, d.h. die Schnittstellen zum Betriebssystem



22.2.2 Temporäre und persistente Daten

- Daten sind
  - *temporär*, wenn sie mit Beendigung des Programms verloren gehen, das sie verwaltet;
  - *persistent*, wenn sie über die Beendigung des verwaltenden Programms hinaus erhalten bleiben.
- Objektorientierte Programme benötigen Mechanismen zur Realisierung der *Persistenz von Objekten*.
  - Einsatz eines Datenbank-Systems
    - Objektorientiertes Datenbank-System
    - Relationales Datenbank-System
      Java: Java Data Base Connectivity (JDBC)
    - Zugriffsschicht auf Datenhaltung
      Java: Java Data Objects (JDO)
  - Speicherung von Objektstrukturen in Dateien mit Senken und Iteratoren
    - Objekt-Serialisierung *(Object Serialization)*
    - Die Dateien werden als Channels benutzt:
    - Zuerst schreiben in eine Sink
    - Dann lesen mit Iterator

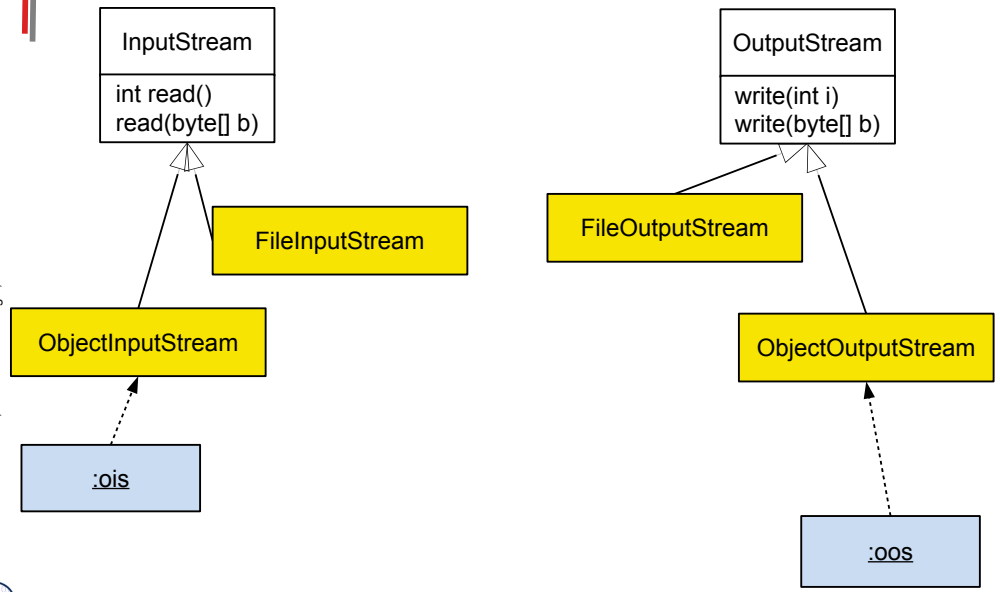
Objekt-Serialisierung in Java, eine einfache Form von persistenten Objekten

- Die Klasse **`java.io.ObjectOutputStream`** und stellt eine Sink dar
  - Methoden, um ein Geflecht von Objekten linear darzustellen (zu *serialisieren*) bzw. aus dieser Darstellung zu rekonstruieren.
    - Ein OutputStream entspricht dem Entwurfsmuster Sink
    - Ein InputStream entspricht dem Entwurfsmuster Iterator
- Eine Klasse, die Serialisierung zulassen will, muß die (leere!) Prädikat-Schnittstelle **`java.io.Serializable`** implementieren.

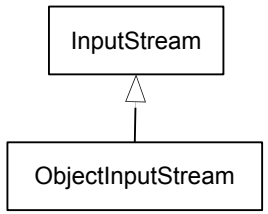
```
class ObjectOutputStream {
  public ObjectOutputStream (OutputStream out)
     throws IOException;
  // push Method
  public void writeObject (Object obj)
     throws IOException;
}
```


Objekt-Serialisierung: Abspeichern


```
import java.io.*;

class XClass implements Serializable {
  private int x;
  public XClass (int x) {
    this.x = x;
  }
}

...
XClass xobj;
...
FileOutputStream fos = new FileOutputStream("Xfile.dat");
ObjectOutputStream oos = new ObjectOutputStream(fos);

// internally realized as push for all child objects
oos.writeObject(xobj); // push
...
```

- **`Die Klasse java.io.ObjectInputStream`** stellt einen Iterator/Stream in unserem Sinne dar
  - Methoden, um ein Geflecht von Objekten linear darzustellen (zu *serialisieren*) bzw. aus dieser Darstellung zu rekonstruieren (zu deseralisieren)
  - Ein OutputStream entspricht dem Entwurfsmuster Sink
  - Ein InputStream entspricht dem Entwurfsmuster Iterator



Objekt-Serialisierung: Einlesen


```
import java.io.*;

class XClass implements Serializable {
  private int x;
  public XClass (int x) {
    this.x = x;
  }
}

...
XClass xobj;
...
FileInputStream fis = new FileInputStream("Xfile.dat");
ObjectInputStream ois = new ObjectInputStream(fis);
// internally realised as pull
xobj = (XClass) ois.readObject(); // pull
```

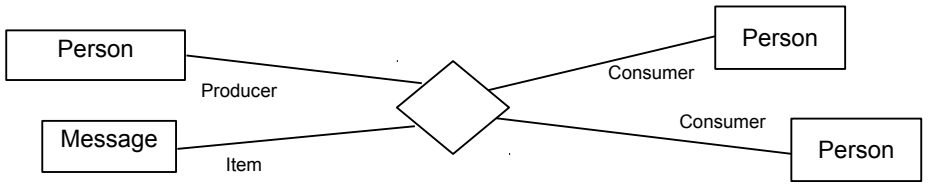

22.2.3 Ereignisse und Kanäle

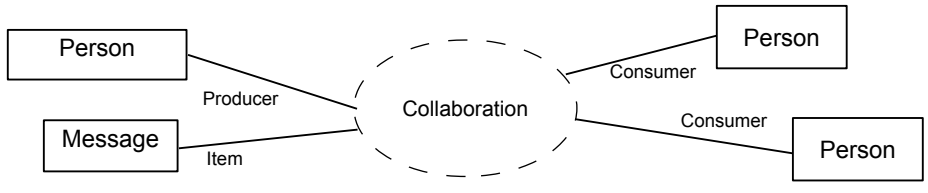
- Kanäle eignen sich hervorragend zur Kommunikation mit der Außenwelt, da sie die Außenwelt und die Innenwelt eines Softwaresystems entkoppeln
- Ereignisse können in der Außenwelt asynchron stattfinden und auf einem Kanal in die Anwendung transportiert werden
  - Dann ist der Typ der Daten ein Ereignis-Objekt
  - In Java wird ein externes oder internes Ereignis immer durch ein Objekt repräsentiert

22.3 Assoziationen, Konnektoren, Kanäle und Teams

Kollaborationen kapseln das Verhalten von Netzen

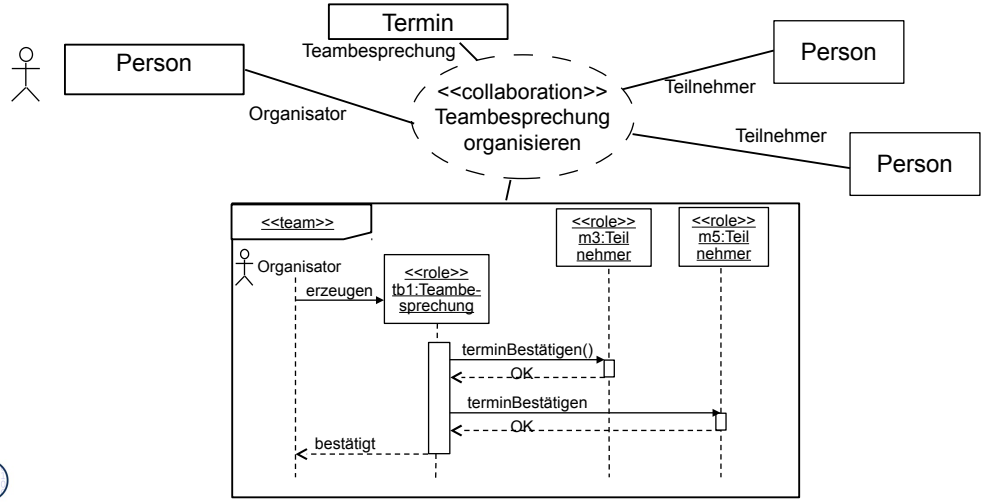
- Statisch fixe Netze werden in UML durch n-stellige Assoziationen oder, wenn es um die Kommunikation der Objekte geht, durch Kollaborationen dargestellt.
- Def.: Eine **Kollaboration (collaboration)** realisiert die Kommunikation eines fixen Netzes mit einem festen anwendungsspezifischen Protokoll





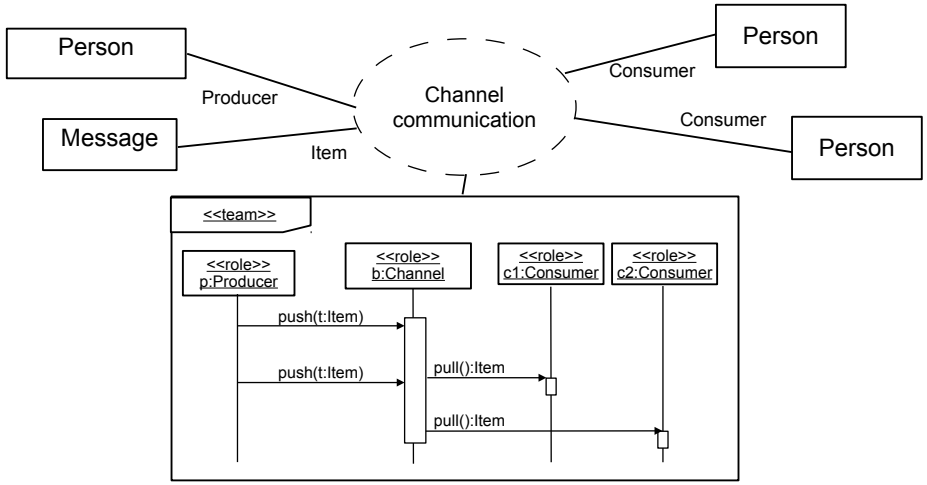
Kollaboration kapseln Interaktionsprotokolle

- **Kollaborationen** beschreiben die anwendungsspezifische Interaktion, Nebenläufigkeit und Kommunikation eines Teams von Beteiligten
- Def.: Ein **Team** realisiert eine Kollaboration durch eine feste Menge von Rollenobjekten, koordiniert durch ein Hauptobjekt. Es wird oft mit einem Sequenzdiagramm als Verhalten unterlegt
  - Die einzelnen Lebenslinien geben das Verhalten einer Rolle der Kollaboration an
- Die Kollaboration beschreibt also ein Szenario querschneidend durch die Lebenszyklen mehrerer Objekte

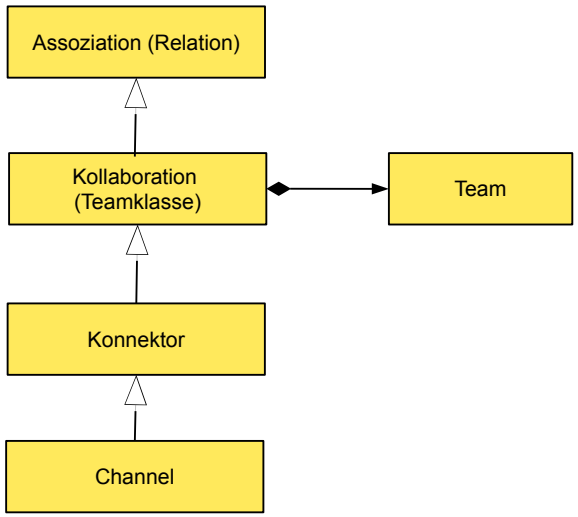


Kanäle sind technische Kollaborationen

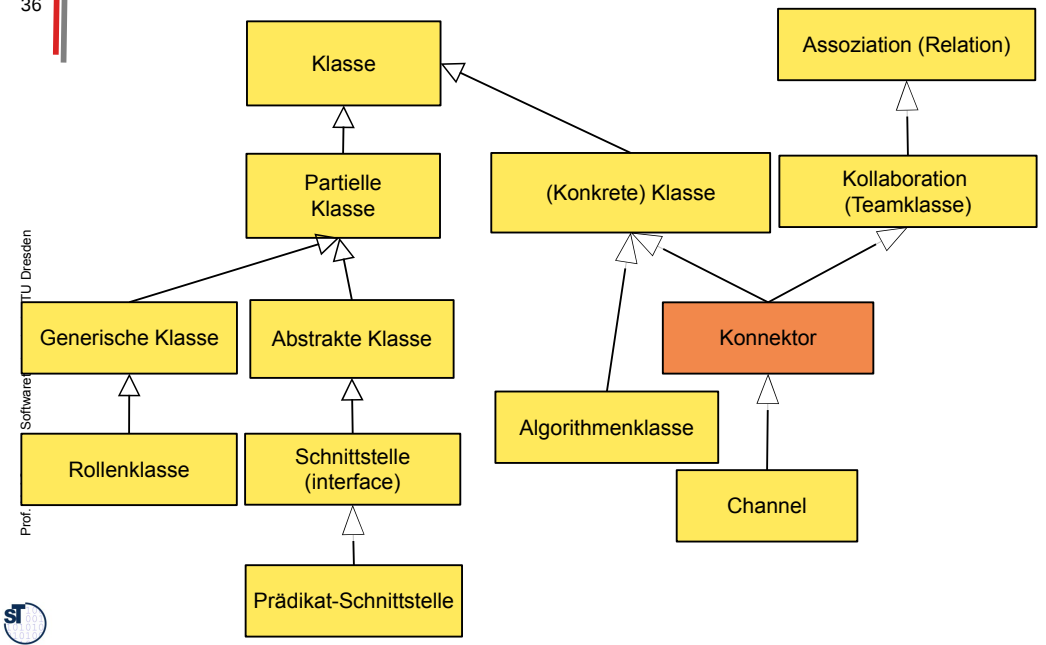
- Ein **Konnektor** bildet eine standardisierte technische Kollaboration.
  - Sie beschreiben die standardisierte Interaktion, Nebenläufigkeit und Kommunikation eines Teams von Beteiligten
  - Konnektoren sind besser wiederverwendbar als allgemeine Kollaborationen
- Bsp: Ein Kanal ist ein einfacher Konnektor



Relationale Klassen (Konnektoren)



Begriffshierarchie von Klassen (Erweiterung)



The End

- Einige Folien sind stammen aus den Vorlesungsfolien zur Vorlesung Softwaretechnologie von © Prof. H. Hussmann, 2002. Used by permission.