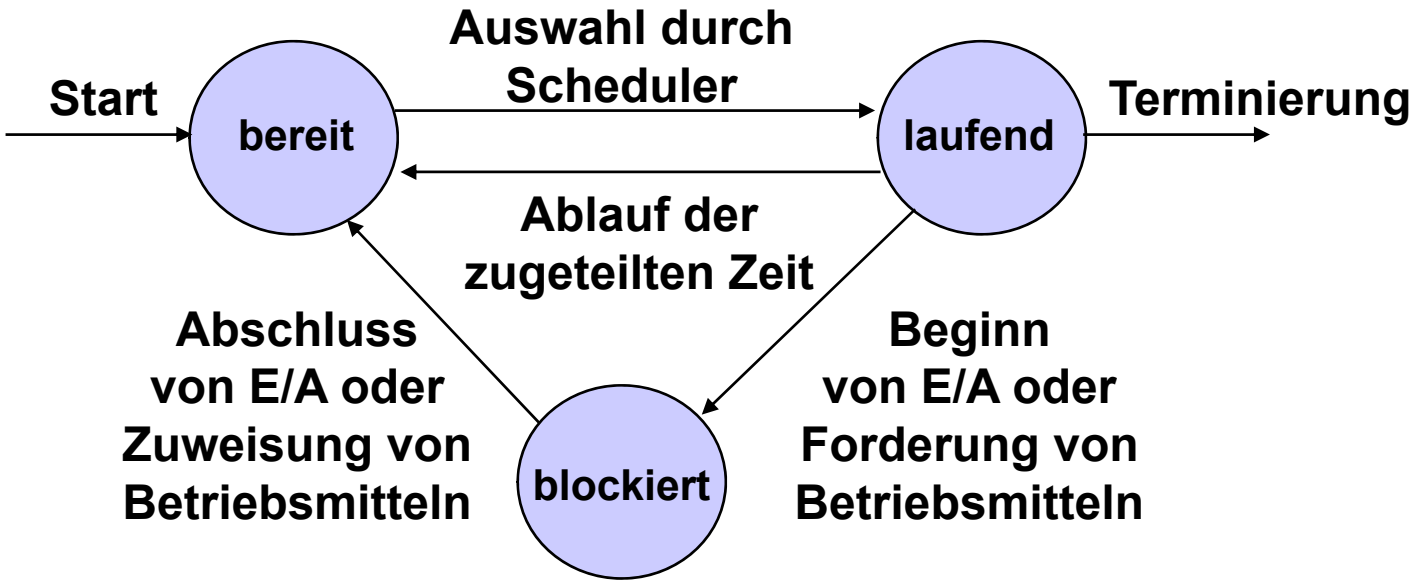
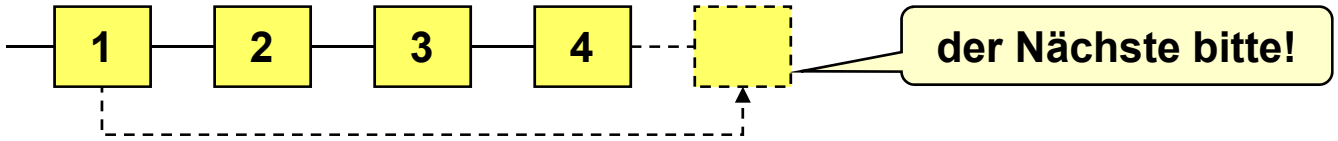
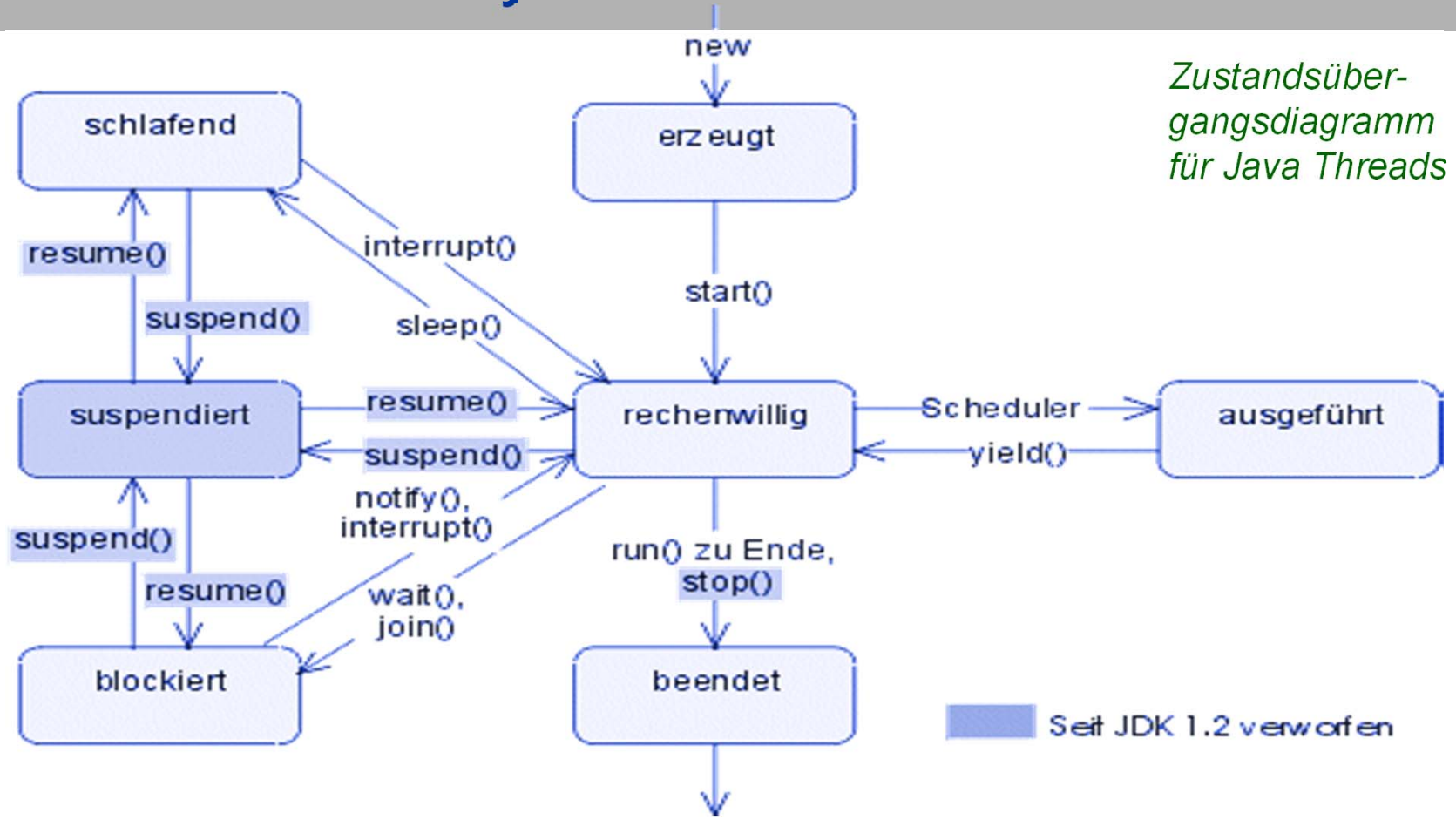
Prozessorzuteilung durch Scheduler



- Status "bereit" kann mehrere Warteschlangen mit verschiedener Priorität besitzen
- Einordnung von Prozessen nach dem "Round-Robin"-Verfahren



Lebenszyklus von Java-Threads



- start() *bewirkt Aufruf der Methode* run() *und nebenläufige Ausführung des Threads*
- *Thread terminiert, wenn* run() *terminiert oder* stop() *ausgeführt wird*
- *Prädikat* isAlive() *liefert true, wenn Thread gestartet und noch nicht terminiert ist*
- *Laufender Thread kann Prozessor durch* yield() *aufgeben*
- *Thread kann durch* suspend() *blockiert und durch* resume() *de-blockiert werden*
- *durch* sleep() *wird Thread auf bestimmte Dauer blockiert*

Synchronisation in Java

Schlüsselwort *synchronized* bewirkt gegenseitigen Ausschluss von nebenläufigen Aktivierungen einer Methode in Java.

Prozessoperationen *wait* und *notify* ermöglichen Prozessverwaltung.

Prozesssynchronisation in Java

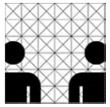
```
public void aMethod () {
        synchronized (anObject) {
        // kritischer Abschnitt
                }
}
```

Beispiel: Interferenz von nebenläufigen Zählerinkrementen verhindern

```
class Counter {
    int value = 0;
    synchronized void increment() {
        ++value;
    }
}
```

Java realisiert gegenseitigen Ausschluss von Methoden verschiedener Threads. Methoden gleicher Threads schließen sich nicht aus.

Mit *synchronized* werden kritische Abschnitte realisiert!



Java-Implementierung eines Semaphors

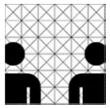
Ein Semaphor ist ein traditioneller Baustein für komplexere Synchronisierungsaufgaben, grundsätzlich in Java entbehrlich, weil beidseitiger Ausschluss durch *synchronized* geregelt werden kann.

```
public class Semaphore {
    private int value;
    public Semaphore (int initial)
        {value = initial;}
        synchronized public void P ()
        throws InterruptedException {
        while (value == 0) wait ();
        --value;}
        synchronized public void V () {
        ++value;
        notify ();}
}
```

Initialwert entspricht Zahl von Passagen vor Blockade

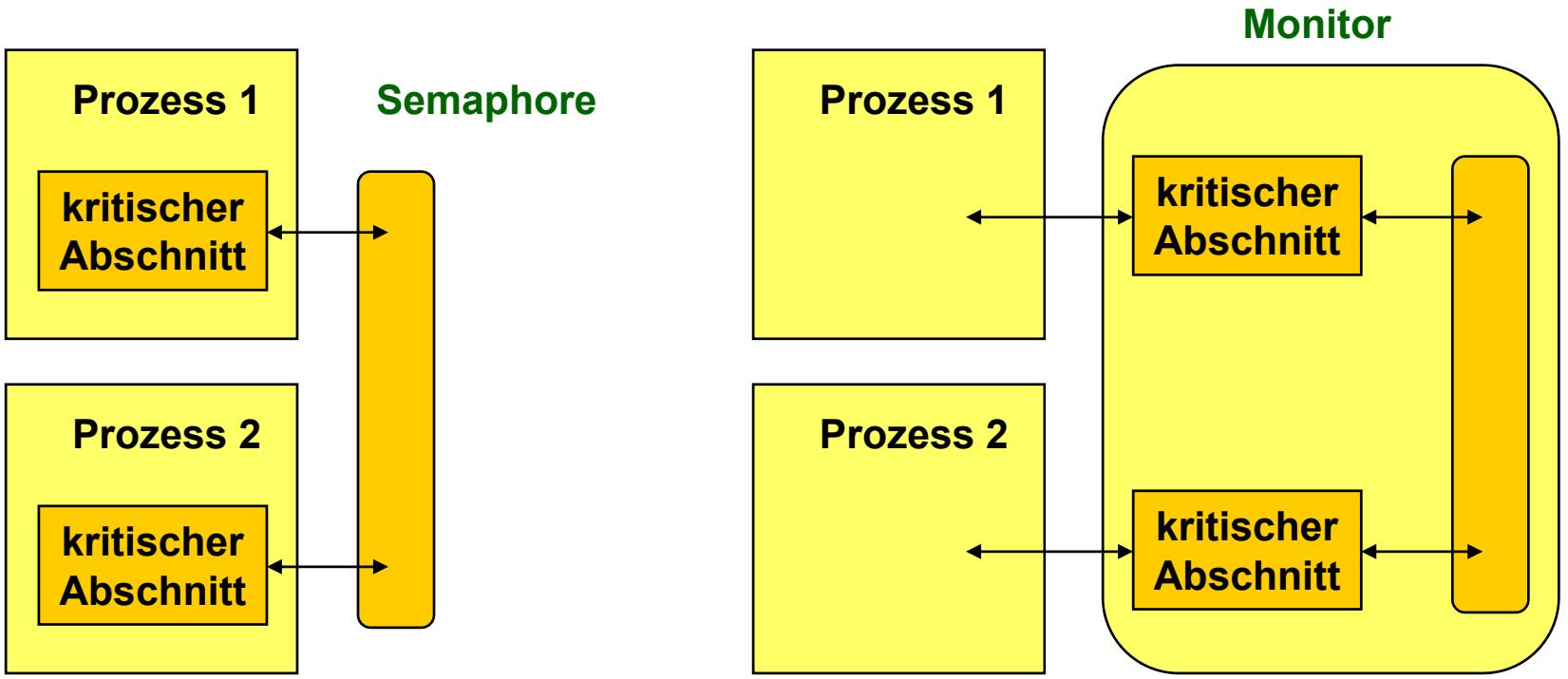
passives Warten, bis Semaphor positiven Wert hat, dann dekrementieren und Passage

nach Inkrementieren nächsten wartenden Prozess aktivieren



Semaphore versus Monitore

Kapselung der kritischen Abschnitte in gemeinsamer abstrakter Datenstruktur („Monitor“) kann größere Klarheit schaffen



gemeinsame Variable (Semaphore)

gemeinsame abstrakte Datenstruktur (Monitor)

Implementierung des Produzenten-Konsumenten-Problems (1)

```
class Produkt {
    private int Ware;
    private boolean verfügbar = false;
    public synchronized int konsumiert () {
        while (! verfügbar) {
            try {wait ();}
            catch (InterruptedException e) { }
        }
        verfügbar = false;
        notify ();
        return Ware;
    }
    public synchronized void produziert (int Warennummer)
{
        while (verfügbar) {
            try {wait();}
            catch (InterruptedException e) { }
        }
        Ware = Warennummer;
        verfügbar = true;
        notify ();
    }
}
```

Klasse* Produkt *muss bei Zugriff auf Ware durch Produzenten und Konsumenten:

1. gegenseitigen Ausschluss garantieren

2. zugreifende Prozesse blockieren und deblockieren

3. über Warenbestand Buch führen

Implementierung des Produzenten-Konsumenten-Problems (2)

```
class Produzent extends Thread {
    private Produkt eineWare;
    Produzent (Produkt c) {eineWare = c;}
    public void run () {
        for (int i = 0; i < 10; i++) {
            eineWare.produziert (i);
            System.out.println (
                i + "produziert");
        }
    }
}
```

```
class Konsument extends Thread {
    private Produkt eineWare;
    Verbraucher (Produkt c) {eineWare = c;}
    public void run() {
        for (int i = 0; i < 10; i++) {
        System.out.println (
            eineWare.konsumiert () +
            "konsumiert");
        }
    }
}
```

Testprogramm

```
class ProduzentKonsument {
    public static void main(String[ ] args) {
        Produkt c = new Produkt ();
        (new Produzent (c)).start ();
        (new Konsument (c)).start ();
    }
}
```

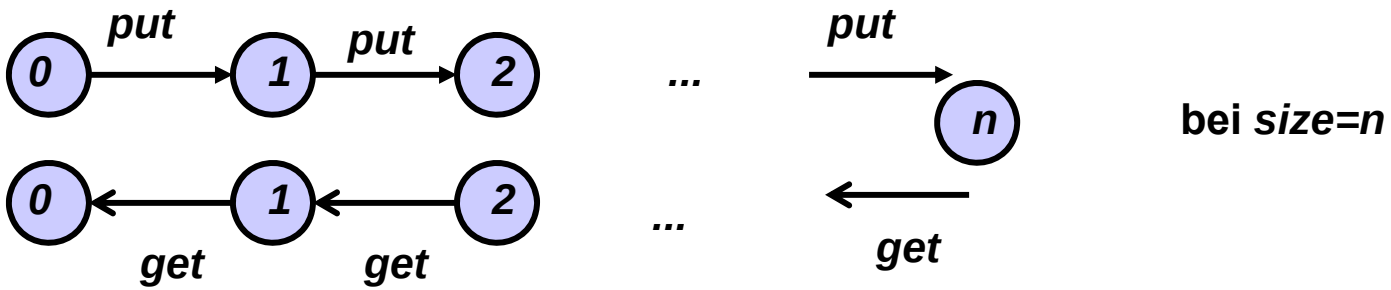
Ausdruck bei Testlauf zeigt abwechselnde Produktion und Konsumption:

```
0 produziert
0 konsumiert
1 produziert
1 konsumiert
2 produziert
2 konsumiert
...
```

Erweiterung des Produzenten/Konsumenten-Problems zum Pufferverwaltungsprogramm

- *Puffer nimmt maximale Zahl von Objekten auf*
- *Produzent füllt Puffer stückweise*
- *Konsument leert Puffer stückweise*

Automatenmodell

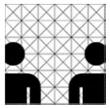


Java-Programme für Pufferverwaltung

- Puffer nimmt begrenzte Zahl von Objekten auf
- Produzent füllt Puffer stückweise
- Konsument leert Puffer stückweise

```
public interface Buffer {
    public void put (Object o)
        throws InterruptException;
    public Object get ()
        throws InterruptException;
}
```

- *Interface ist abgetrennt, um alternative Implementierungen zu ermöglichen*
- *Puffer hat feste Größe* size, *nimmt beliebige Objekte auf, ist als Ringpuffer organisiert*
- notify *nach* put, *falls abnehmender Prozess wartet*
- notify *nach* get, *falls liefernder Prozess wartet*



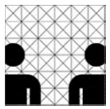

```
class BufferImpl implements Buffer {
    protected Object[ ] buf;
    protected int in = 0;
    protected int out = 0;
    protected int count = 0;
    protected int size;
    BufferImpl (int size) {
        this.size = size; buf = new Object[size];
    }
    public synchronized void put (Object o)
            throws InterruptException {
        while (count == size) wait ();
        buf [in] = o;
        ++count;
        in = (in + 1) mod size;
        notify ();
    }
    public synchronized Object get ()
            throws InterruptException {
        while (count == 0) wait ();
        Object o = buf[out];
        buf [out] = null;
        --count;
        out = (out + 1) mod size;
        notify ();
        return (o);
    }
}
```

Java-Programme für Puffer-Zugriff

```
class Producer implements runnable {
    Buffer buf;
    Object item;
    Producer (Buffer b) {buf = b};
    public void run () {
        try {
            while (true) {
                buf.put (new item);
            }
            catch (InterruptedException e){ }
        }
    }
}
```

```
class Consumer implements runnable {
    Buffer buf;
    Object item;
    Consumer (Buffer b) {buf = b};
    public void run () {
        try {
            while (true) {
                item = buf.get ();
            }
            catch (InterruptedException e){ }
        }
    }
}
```

- *Lieferant erzeugt Objekte* (new item) *in Endlosschleife und legt sie im Puffer ab*
- *Abnehmer entfernt Objekte aus Puffer in Endlosschleife (und tut hier nichts weiter damit)*



Prozesskommunikation

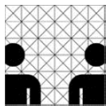
Bisher haben Prozesse über *gemeinsam zugreifbare Variable* interagiert. Sind keine gemeinsamen Datenbereiche vorhanden, müssen Informationen als Nachrichten oder Botschaften (messages) ausgetauscht werden.

Entsprechungen:			
	Schreiben	⟷	**Senden**
	Lesen	⟷	**Empfangen**
	gemeinsamer Datenbereich	⟷	**Kommunikations-kanal**

Nachrichtenaustausch ist eine mächtige Metapher für Synchronisierung, denn implizit gilt: (sende Nachricht) → (empfange Nachricht)

Ein Kommunikationskanal kann als abstrakter Datentyp realisiert werden und unterscheidet sich dann kaum von einem gemeinsamen Datenbereich:

send wert to kanal <=> kanal.send (wert)



Relevante Eigenschaften für Synchronisierung

Senden von Nachrichten

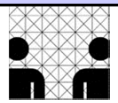
blockierend:	***Prozess wartet nach Sendeoperation auf Empfangsbestätigung***
nicht blockierend:	***Prozess läuft nach Sendeoperation weiter***

Empfangen von Nachrichten

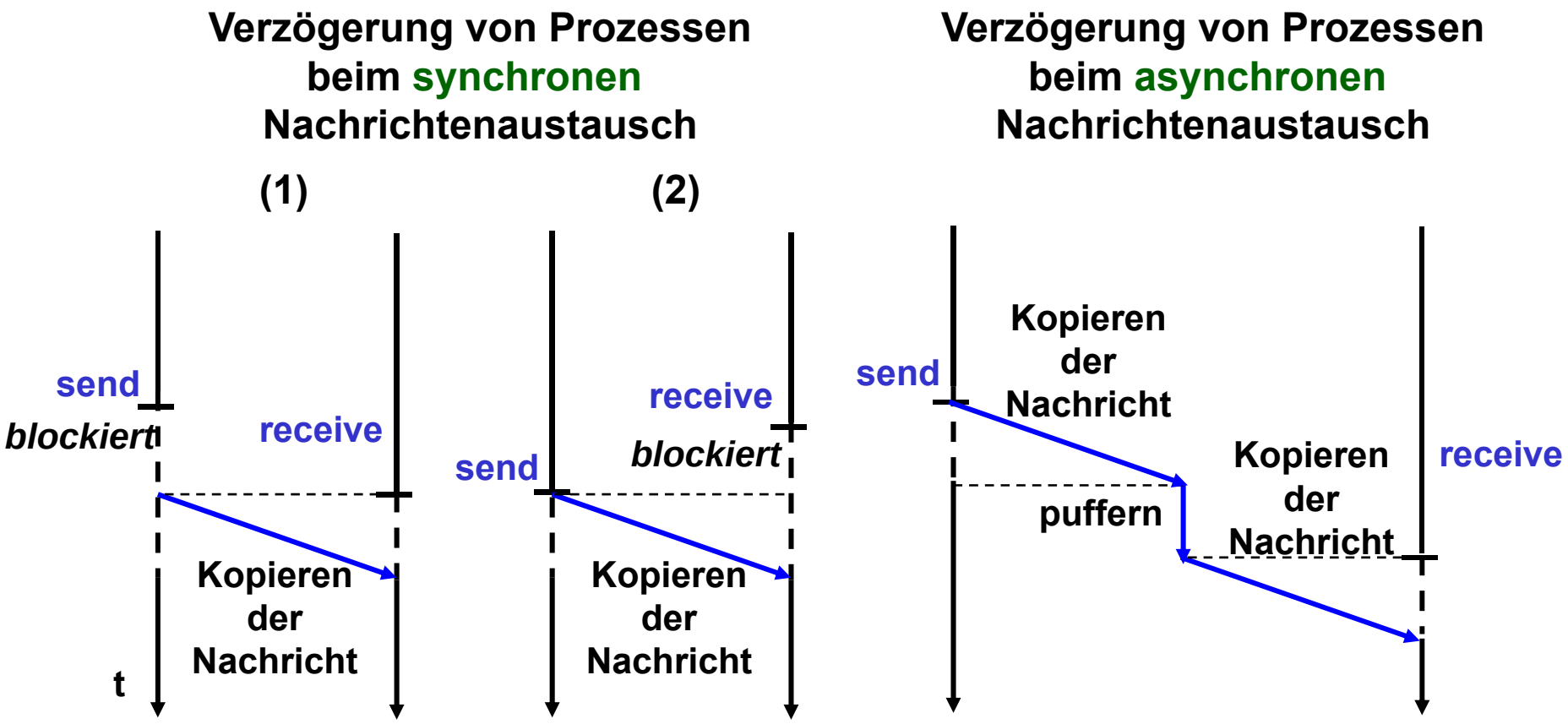
blockierend (üblich):	***Prozess wartet auf Nachrichtenempfang***
nicht blockierend:	***Prozess bleibt bei fehlender Nachricht aktiv (z.B. Test auf zu aktualisierende Werte)***

Kommunikationskanal

gepuffert:	***Nachrichten werden entsprechend der Sendefolge zwischengelagert***
ungepuffert:	***Nachrichten werden direkt vom Sender zum Empfänger kopiert***



Synchroner und asynchroner Nachrichtenaustausch



asynchrones Verhalten geht verloren, wenn
- Puffer voll ist und Sender blockiert wird
- Puffer leer ist und Empfänger blockiert wird

Nachrichtenaustausch zwischen mehr als 2 Prozessen

Rundsendung (broadcast)

Nachricht wird an alle denkbaren Empfänger gesendet

```
broadcast wert
```

Mehrfachsendung (multicast)

Nachricht wird an mehrere spezifizierte Empfänger gesendet

```
multicast wert to (kanal1, kanal2, kanal3)
```

Selektives Empfangen

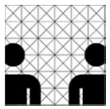
Nichtdeterministische Auswahl von eingetroffenen Nachrichten

```
select
    receive variable1 from kanal1 → anweisung1
    receive variable2 from kanal2 → anweisung2
    receive variable3 from kanal3 → anweisung3
end select
```

Bedingtes selektives Empfangen

Auswahl zwischen Nachrichten, für die eine Bedingung zutrifft

```
select
    (when B1 and receive variable1 from kanal1)
        → anweisung1
    (when B2 and receive variable2 from kanal2)
        → anweisung2
end select
```

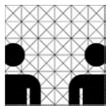


Konstrukte für Nachrichtenaustausch zwischen Java-Prozessen

Java bietet keine besonders eleganten Sprachelemente zum Nachrichten-austausch zwischen Prozessen.

Methoden der Basisklassen *Select* und *Selectable* steuern Auswahl aus Warteschlangen synchronisierter Objekte.

select.add	***fügt ein* selectable *Objekt in Warteschlange für selektives Empfangen ein***
select.choose	***führt selektives Empfangen von* selectable *Objekten aus, die in der Warteschlange sind***
selectable.guard	***testet* selectable *Objekt in Warteschlange für selektives Empfangen***



Java-Programm für selektiven Nachrichtenempfang

```
class Channel extends Selectable {
    public synchronized void send (Object v)
        throws InterruptException { ... }
    public synchronized Object receive ( )
        throws InterruptException { ... }
```

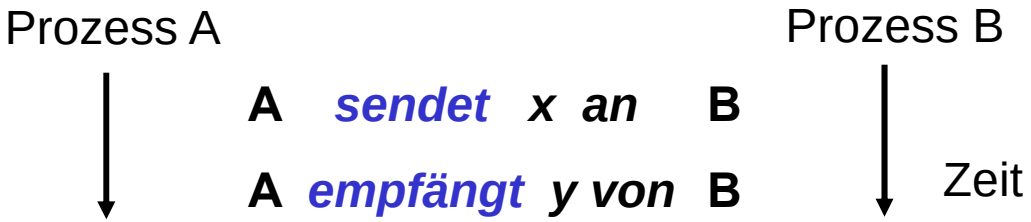
Implementierung eines Nachrichtenkanals **Channel** *mit Hilfe der Klasse selectable*

```
class MessageReceiver implements Runnable {
    private Channel arrive1, arrive2;
    public void run ( ) {
        try {
            Select sel = new Select ( );
            sel.add (arrive1);
            sel.add (arrive2)
            while (true) {
                arrive1.guard (<Bedingung1>);
                arrive2.guard (<Bedingung2>);
                switch (sel.choose ( )) {
                    case 1: arrive1.receive ( ); ...; break;
                    case 2: arrive2.receive ( ); ... ; break;
                }
            }
        } catch InterruptException{ }
    }
}
```

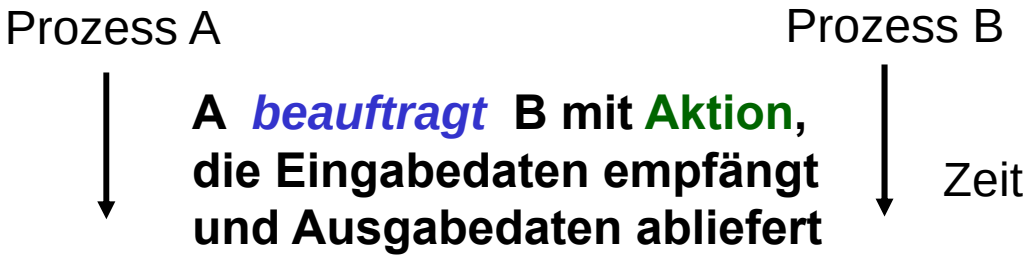
Selektive Auswahl von auswahlbereiten Nachrichten

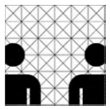
Abstraktion vom Nachrichtenaustausch

Bisher datenorientierter Nachrichtenaustausch mit typischem Muster:



Aktionsorientierter Nachrichtenaustausch bietet Abstraktionsmöglichkeit durch Zusammenfassen der Aktivitäten von B als Aktion:





Fernaufruf von Prozeduren

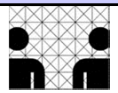
Die Beziehung zwischen Auftraggeber und Auftragnehmer lässt sich durch einen entfernten Prozedurfernaufruf (Remote Procedure Call, RPC) in vertrauter Weise (d.h. - fast - wie im lokalen System) modellieren.

Unterschied zum lokalen Prozeduraufruf (u.a.):

- **Auftraggeber und Auftragnehmer sind verschiedene Prozesse in verschiedenen Datenräumen**
- **Auftraggeber und Auftragnehmer sind nebenläufig, Synchronisationsbedarf je nach Art des Auftrags**

```
auftraggeber: process
eing: eTyp
ausg: aTyp
   repeat
   ...
   auftragnehmer.auftrag (eing, ausg);
   ...
   end repeat
end process
```

```
auftragnehmer: process
   export auftrag;
   auftrag: procedure (ein: eTyp; out aus: aTyp)
   ... // Auftrag bearbeiten
   end procedure
end process
```



Rendezvous

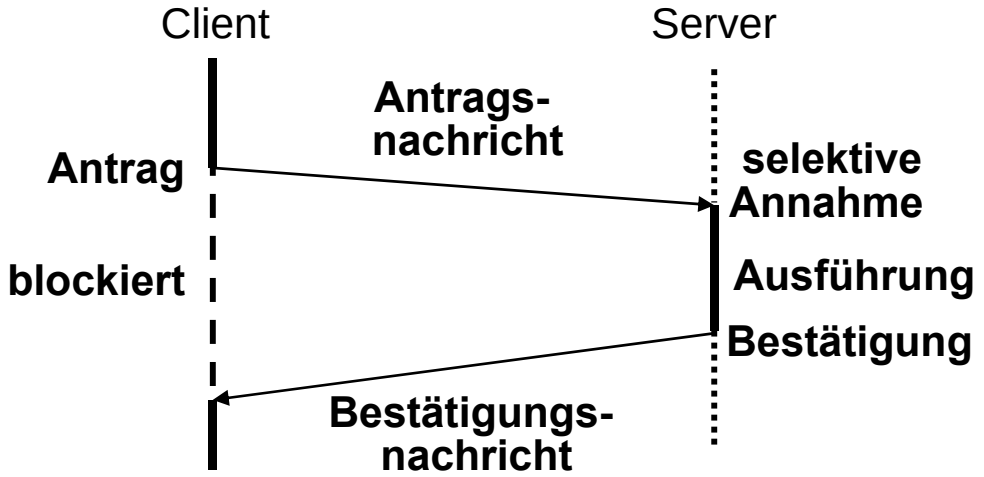
Rendezvous = Prozedurfernaufruf mit größerer Autonomie des aufgerufenen Prozesses (bestimmt selbst über Ausführung des Auftrags)

Sprachgebrauch:

Client **beantragt einen Dienst (request)**

Server
- **bietet Dienst an (offer)**
- **nimmt Dienstauftrag an (accept)**
- **führt Dienst aus (execution)**
- **bestätigt Dienst (reply)**

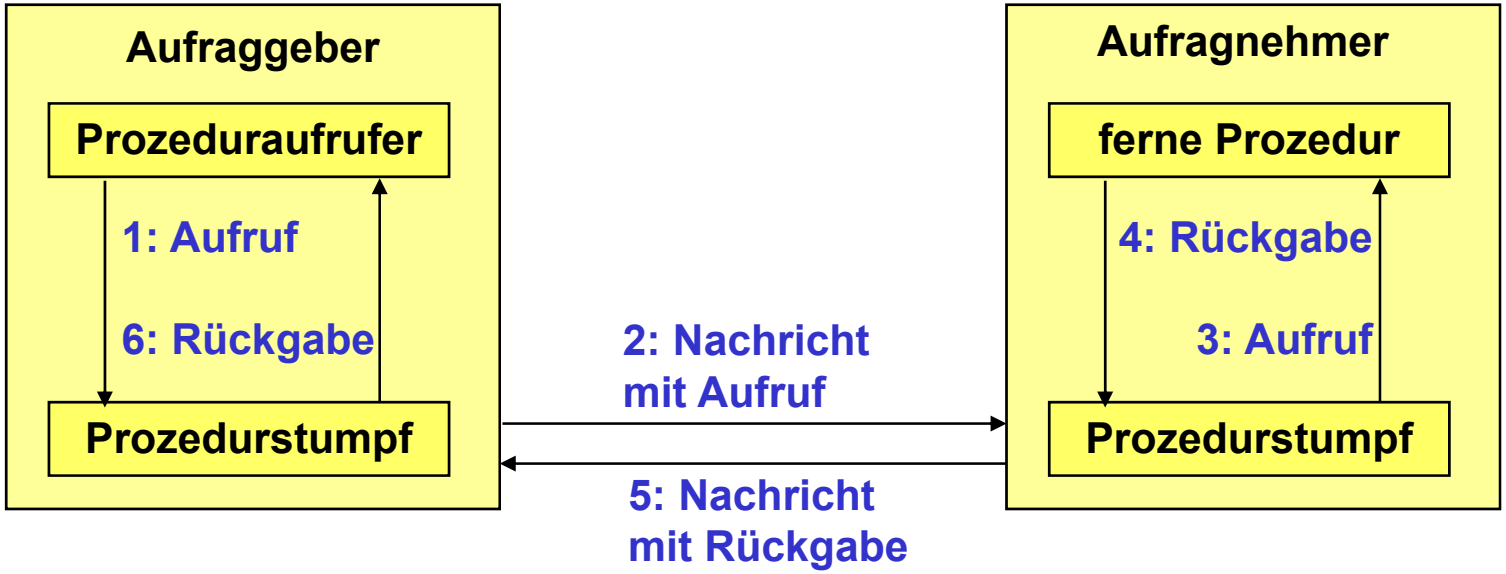
Rendezvous werden vom Server in der Regel selektiv eingegangen. Client ist während der Ausführung des Dienstes meist blockiert.



Implementierung von Prozedurfernaufrufen

Prozedurstumpf („stub“) auf Auftraggeber**seite repräsentiert ferne Prozedur im Adressraum des Auftraggebers und sorgt für Nachrichtenaustausch mit Auftragnehmer.**

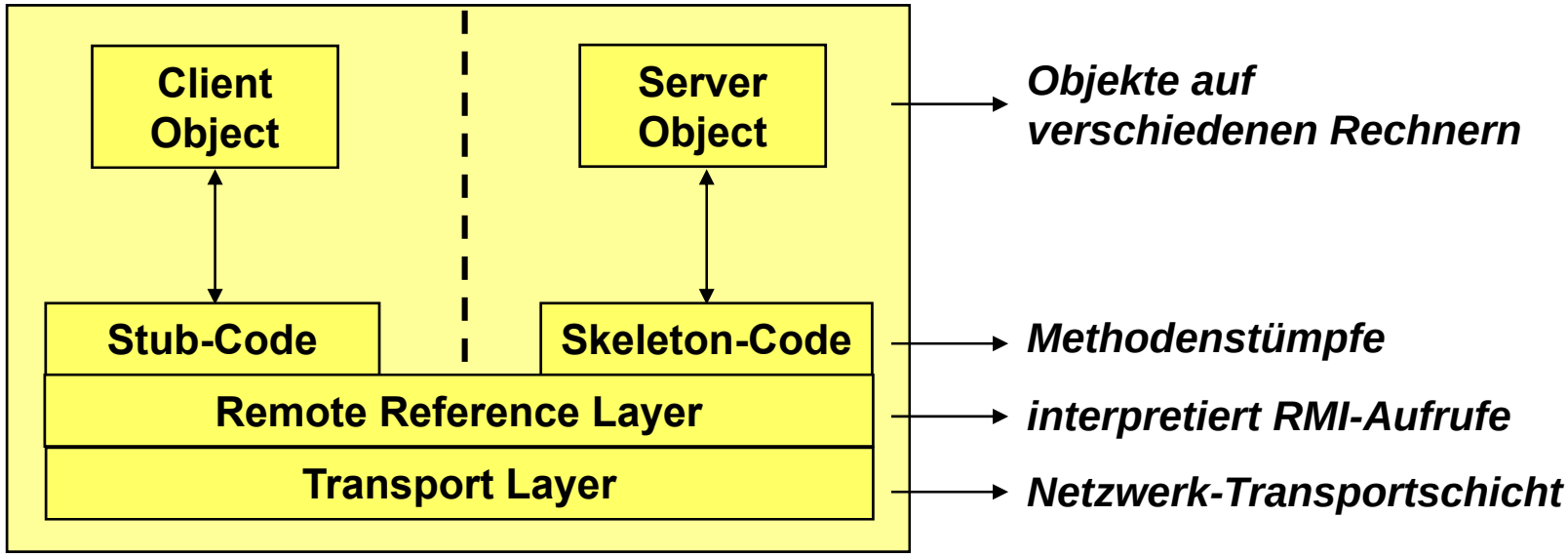
Prozedurstumpf auf Auftragnehmer**seite sorgt für Prozeduraufruf und Nachrichtenaustausch mit Auftraggeber.**



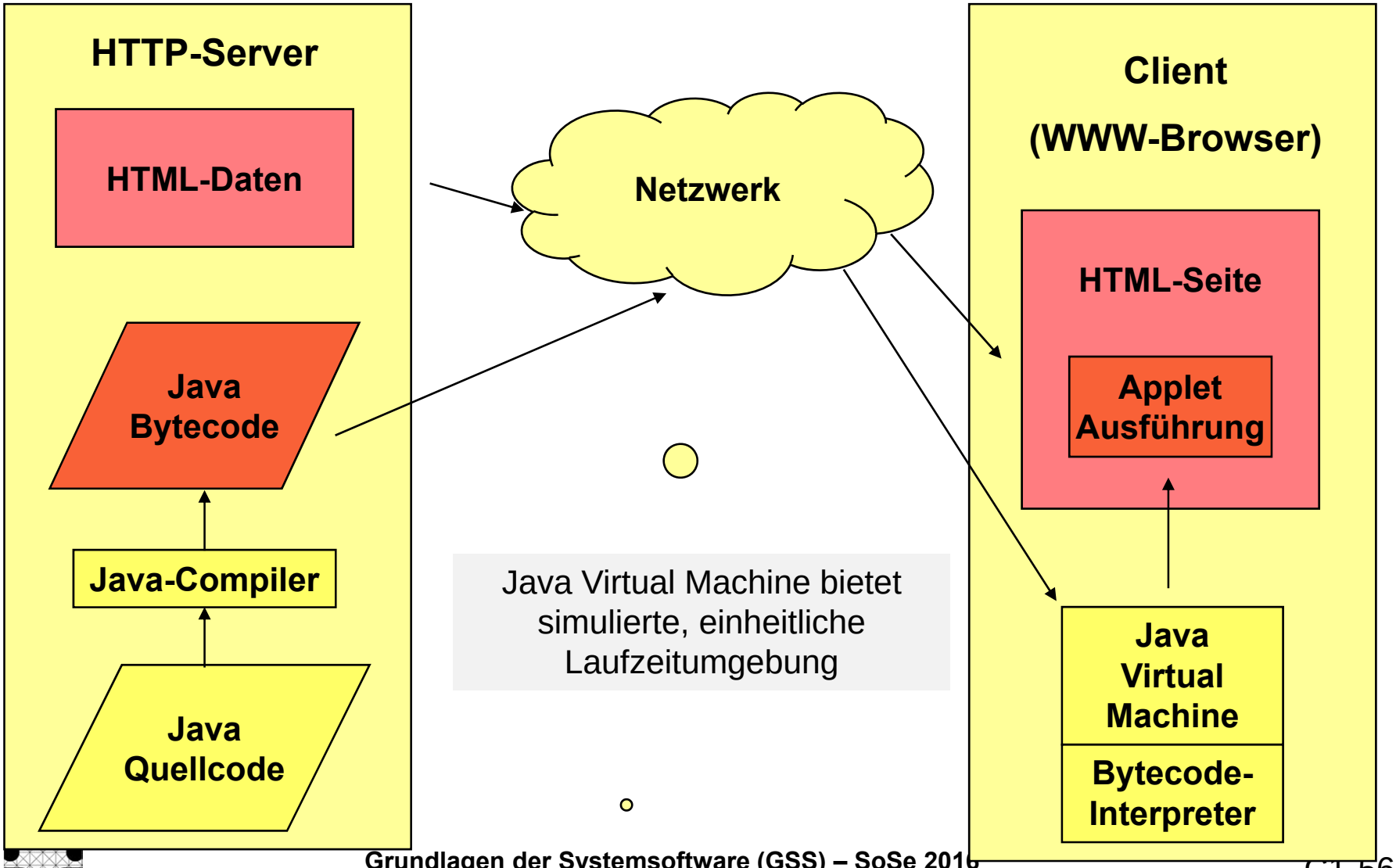
Methodenfernaufruf in Java

Java-Objekt auf Rechner A (Client) kann Methoden eines entfernten Objektes auf Rechner B (Server) durch „Remote Method Invocation“ (RMI) aufrufen.

Architektur des RMI-Systems:



Fernausführung von Java-Applets im WWW

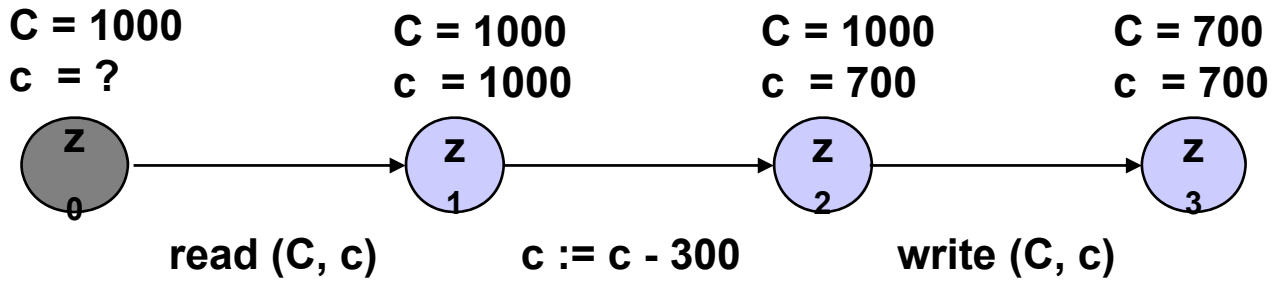


Aber: Nebenläufigkeitsprobleme erfordern oft abstrakte Modellierung

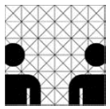
C 1.5: Abstrakte Modellierung
Modellierung von Prozessen durch endliche Automaten

Prozesszustände	**Zustände eines Automaten**
Aktivitäten	**Eingaben des Automaten, bewirken Zustandsübergänge**

Beispiel:



$Z = \{ z_0 \ldots z_N \}$ Menge der Zustände
$A = \{ a_0 \ldots a_M \}$ Menge der Aktivitäten
$E = \{ (z_i\ a_j\ z_k) \}$ Zustandsübergänge des endlichen Automaten

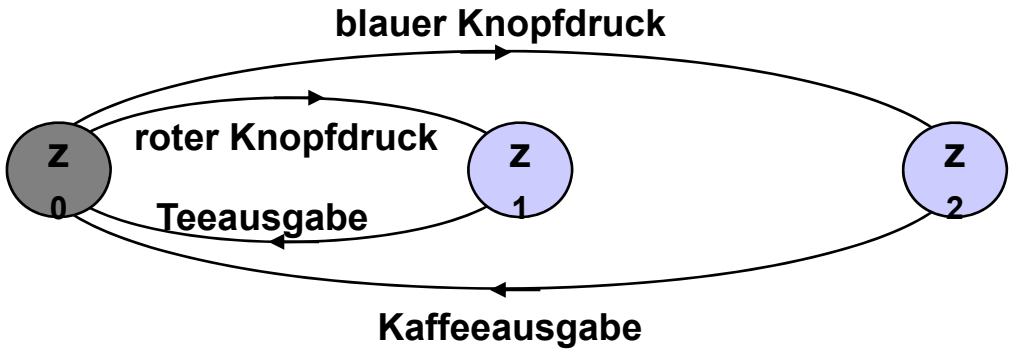


Deterministische Wahl

Eine deterministische Wahl besteht, wenn ein Prozess von einem Zustand aus durch unterschiedliche Aktivitäten in verschiedene – aber dann jeweils *eindeutige* – Folgezustände übergehen kann.

Beispiel:

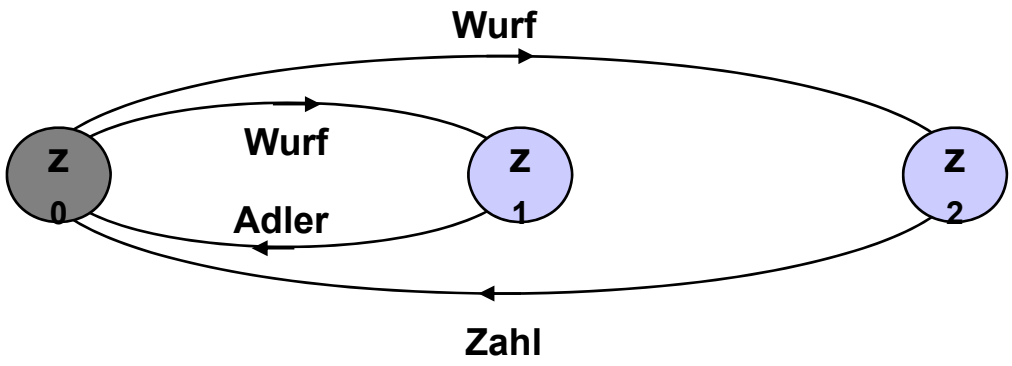
Getränkeautomat hat roten Knopf für Tee und blauen Knopf für Kaffee



Nichtdeterministische Wahl

Ein Prozess ist nicht-deterministisch, wenn er bei *gleicher* Aktivität in *verschiedene* Zuständen übergehen kann.

Beispiel: Münzwurf



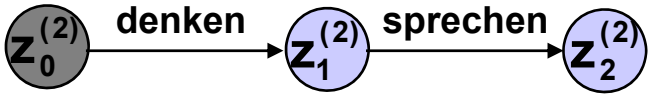
Parallele Ausführung nebenläufiger Prozesse

Beschreibung der möglichen Verzahnungen zweier nebenläufiger Prozesse durch einen Produktautomaten.

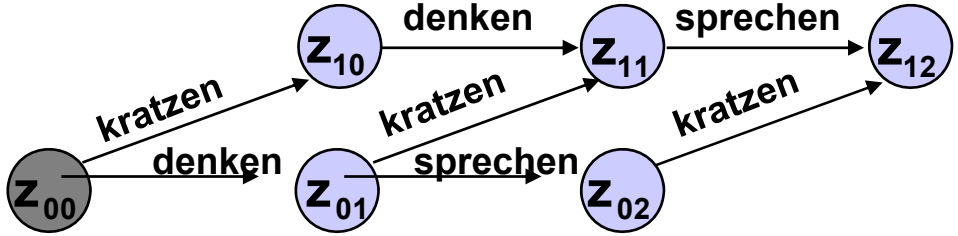
Beispiel:

Jucken

Vortragen



Jucken || Vortragen

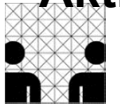


Definition Produktautomat:

Zustände $Z = \{ z_{ij} \mid z_i^{(1)} \in Z^{(1)} \wedge z_j^{(2)} \in Z^{(2)} \}$

Zustandsübergänge $E = \{ (z_{ij}\ a_k\ z_{mn}) \mid (z_i\ a_k\ z_m) \in E^{(1)} \vee (z_j\ a_k\ z_n) \in E^{(2)} \}$

Aktivitäten $A = A^{(1)} \cup A^{(2)}$



Nebenläufige Prozesse mit gemeinsamen Aktivitäten

Enthalten Prozesse gemeinsame Aktivitäten, so müssen diese gleichzeitig ausgeführt werden.

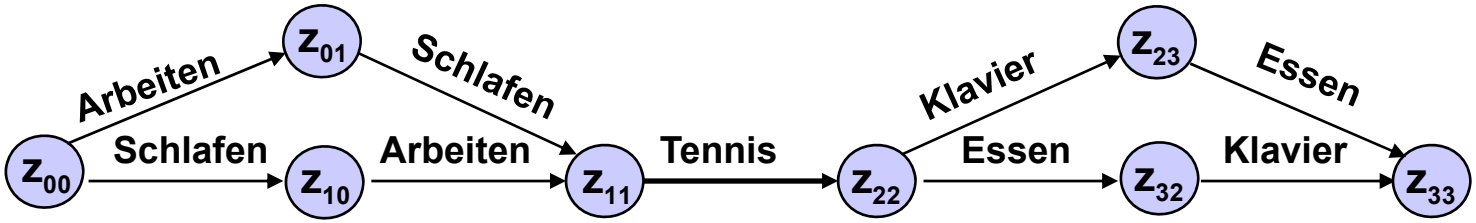
gemeinsame Aktivität "Tennis"

Beispiel:

$$\text{Felix} = \{(z_0^{(1)}\ \text{Schlafen}\ z_1^{(1)})(z_1^{(1)}\ \text{Tennis}\ z_2^{(1)})(z_2^{(1)}\ \text{Essen}\ z_3^{(1)})\}$$

$$\text{Marietta} = \{(z_0^{(2)}\ \text{Arbeiten}\ z_1^{(2)})(z_1^{(2)}\ \text{Tennis}\ z_2^{(2)})(z_2^{(2)}\ \text{Klavier}\ z_3^{(2)})\}$$

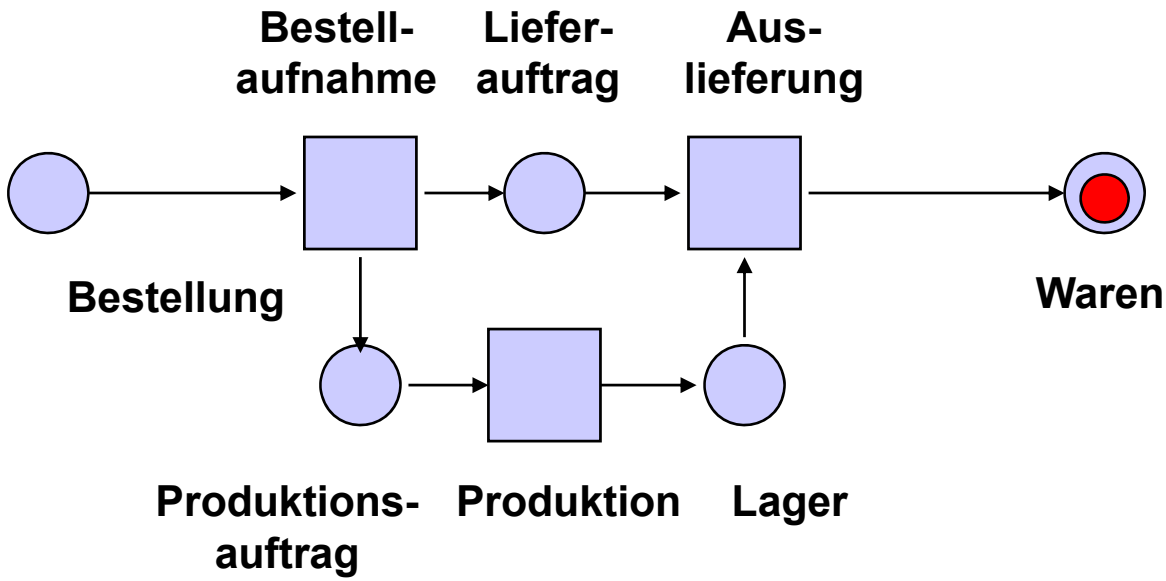
$$\text{Felix} \mid\mid \text{Marietta} = \{(z_{00}\ \text{Schlafen}\ z_{10})(z_{01}\ \text{Schlafen}\ z_{11})(z_{00}\ \text{Arbeiten}\ z_{01})(z_{10}\ \text{Arbeiten}\ z_{11})$$
$$(z_{11}\ \text{Tennis}\ z_{22})$$
$$(z_{22}\ \text{Essen}\ z_{32})(z_{23}\ \text{Essen}\ z_{33})(z_{22}\ \text{Klavier}\ z_{23})(z_{32}\ \text{Klavier}\ z_{33})\}$$

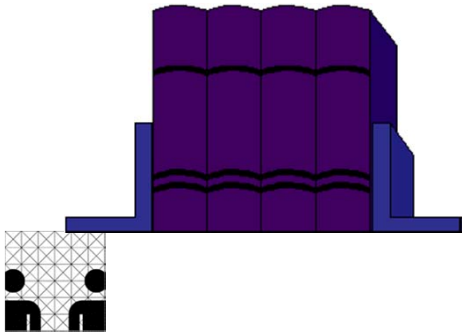


Petri-Netze

Anschauliche Modellierungsmethode für nebenläufige Prozesse und ihre Synchronisation.

Beispiel:
Materialverwaltung

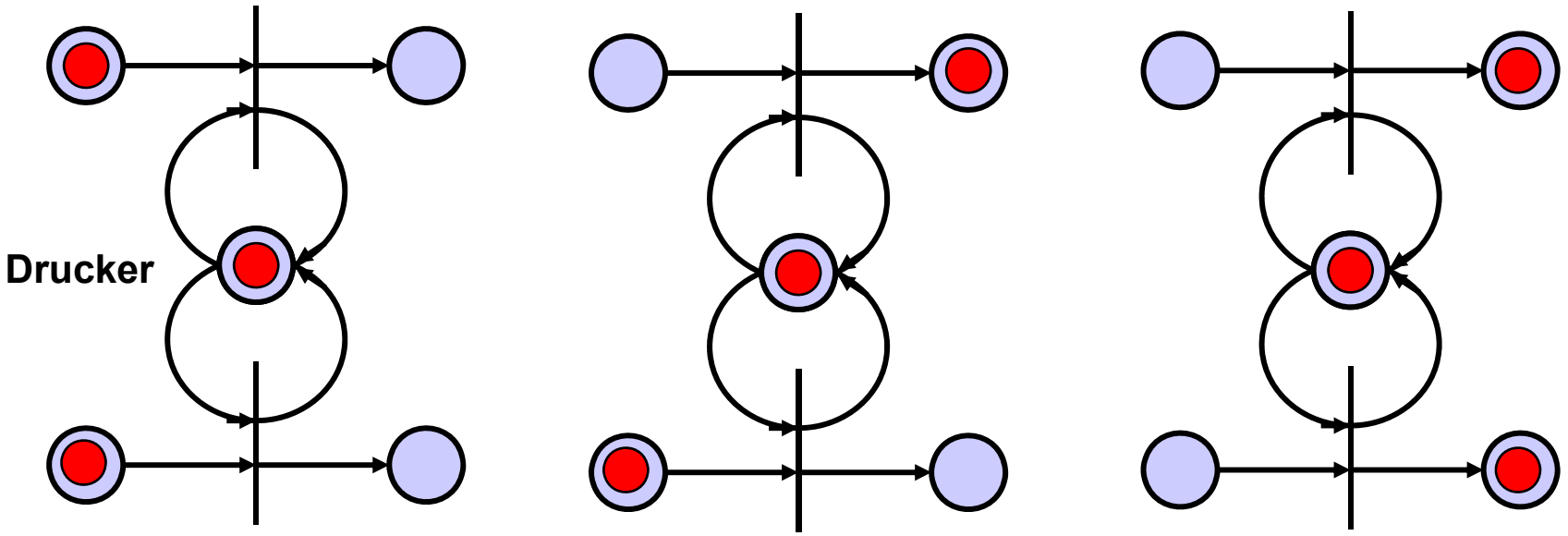




W. Reisig: Petri-Netze - Modellierungstechnik, Analysemethoden, Fallstudien, Vieweg/Teubner, 2010

Benutzung eines Betriebsmittels durch zwei Prozesse

Zwei Prozesse wollen einen Drucker benutzen ...



beide Transitionen feuerbereit

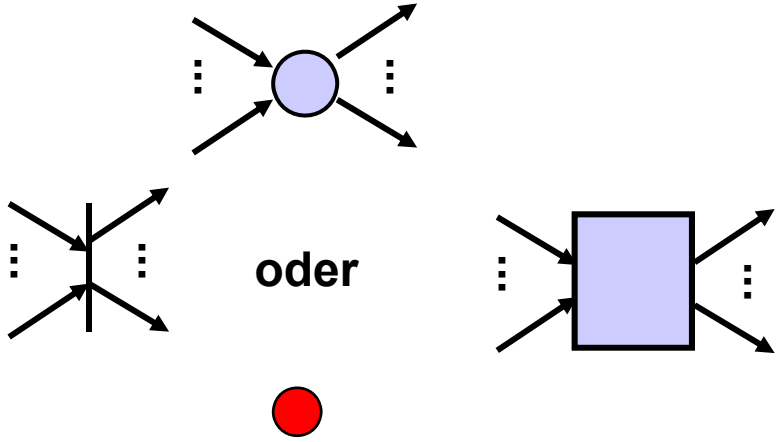
nur eine Transition kann feuern

die zweite Transition kann danach feuern

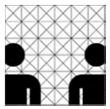
Erinnerung: Grundelemente von Petri-Netzen

Abstraktion interagierender nebenläufiger Prozesse durch S/T-Netz aus

- **Stellen (Plätzen)**
- **Transitionen (Übergängen)**
- **Marken, die nach bestimmten Regeln verschoben werden können**



- **Stellen sind nur mit Transitionen, Transitionen nur mit Stellen verbunden**
- **Eine Transition kann feuern, wenn alle Eingangsstellen mit Marken besetzt sind**
- **Beim Feuern einer Transition werden alle Eingangsstellen freigemacht, alle Ausgangsstellen besetzt**

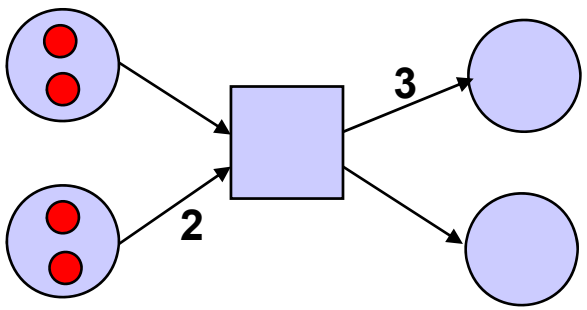


Kapazität und Gewichtung

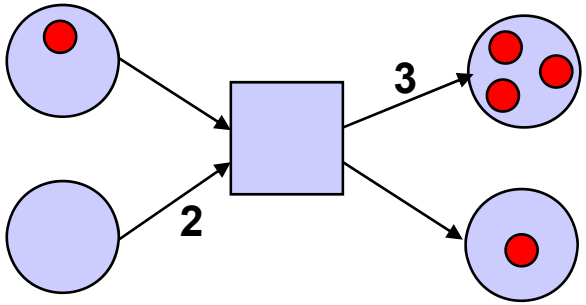
Die *Kapazität* einer Stelle ist die Zahl der maximal aufnehmbaren Marken dieser Stelle. Ohne Angabe ist die Kapazität ∞.

Kanten können eine *Gewichtung* tragen:

- **Zahl der Marken, die beim Schalten der Transition von einer Eingangsstelle entfernt werden müssen**
- **Zahl der Marken, die beim Schalten der Transition einer Ausgangsstelle zugefügt werden müssen**



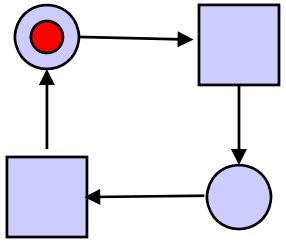
vorher



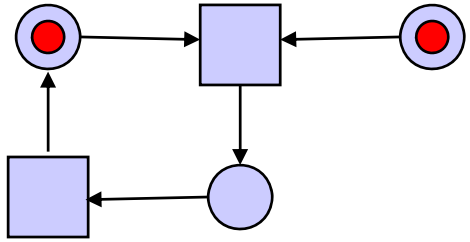
nachher

Lebendige und sichere Netze

Ein (Teil-) Netz heißt *lebendig*, wenn es keinen Zustand geben kann, wo es
- wegen zu wenig Stellen im Vorbereich, oder
- wegen zu viel Stellen im Nachbereich
nicht mehr schalten kann. Andernfalls heißt es *todesgefährdet*.

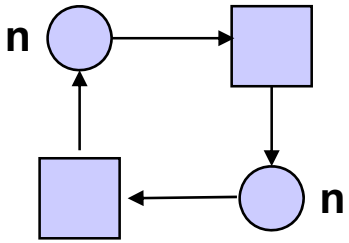


lebendig

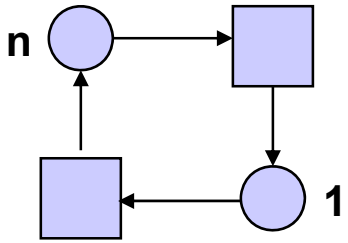


todesgefährdet

Ein Netz heißt *sicher*, wenn eine Erhöhung von Kapazitäten nicht zu mehr Schaltmöglichkeiten führt.



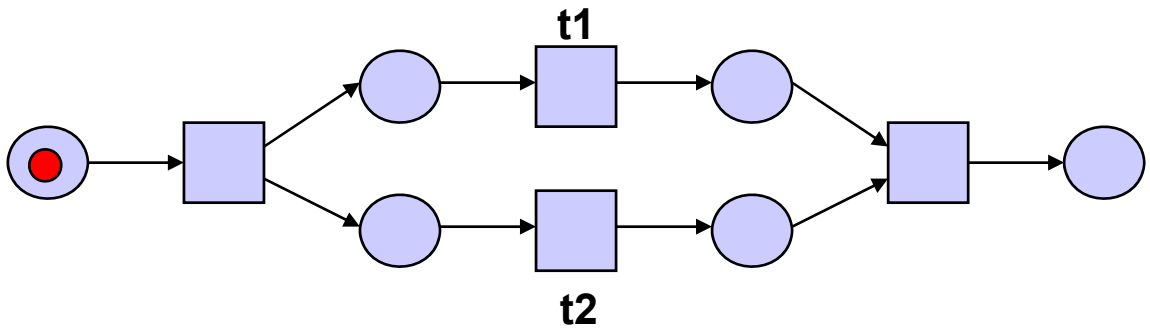
sicher



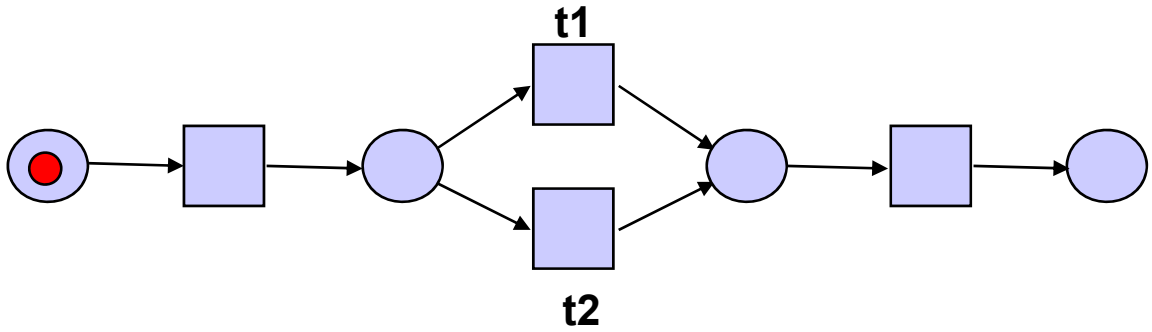
unsicher

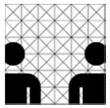
Netzmuster (1)

Nichtdeterministische Reihenfolge von t1 und t2



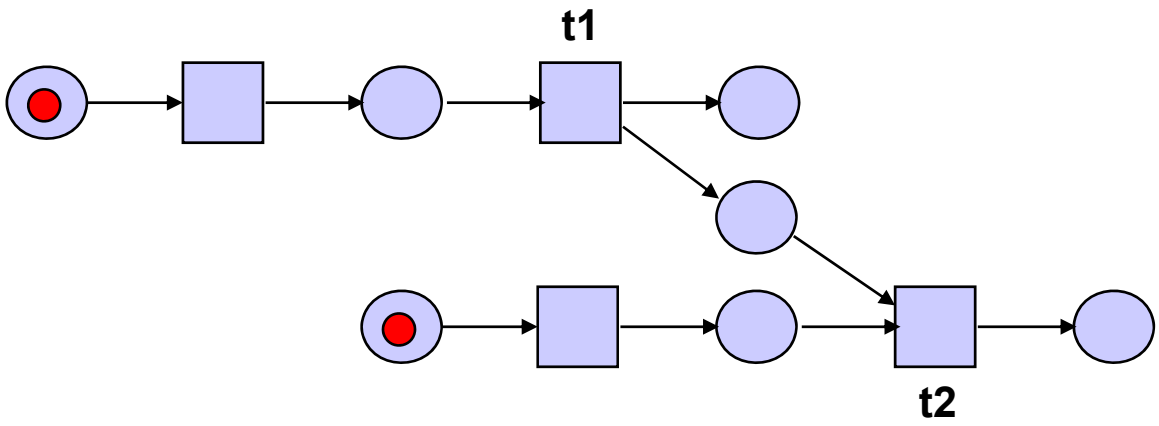
Nichtdeterministische Auswahl von t1 und t2





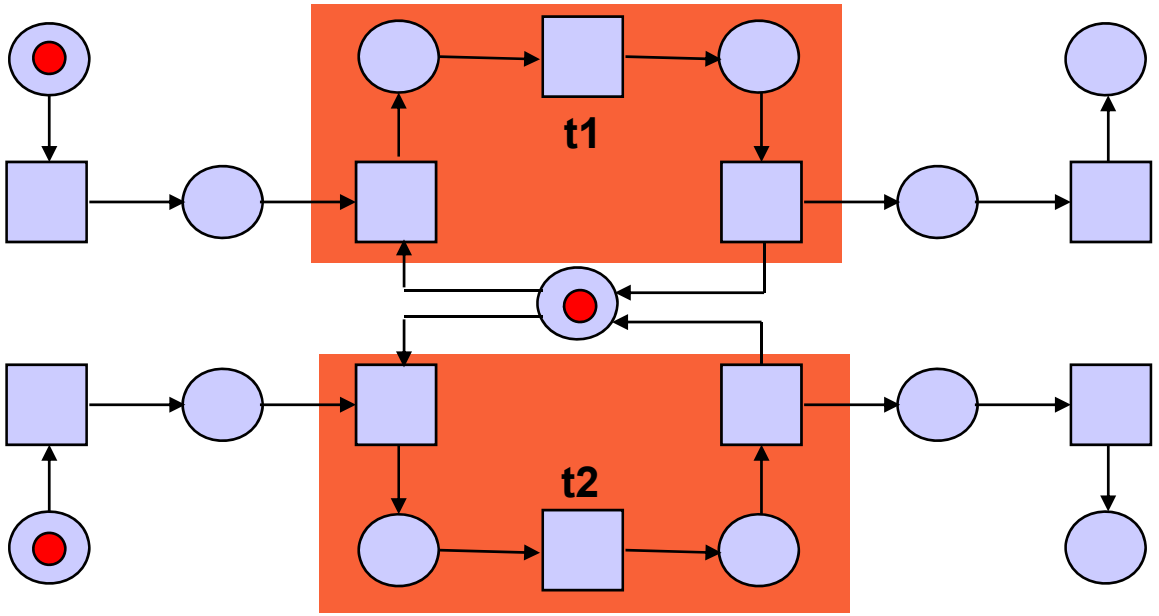
Netzmuster (2)

Einseitige Synchronisierung t1→ t2



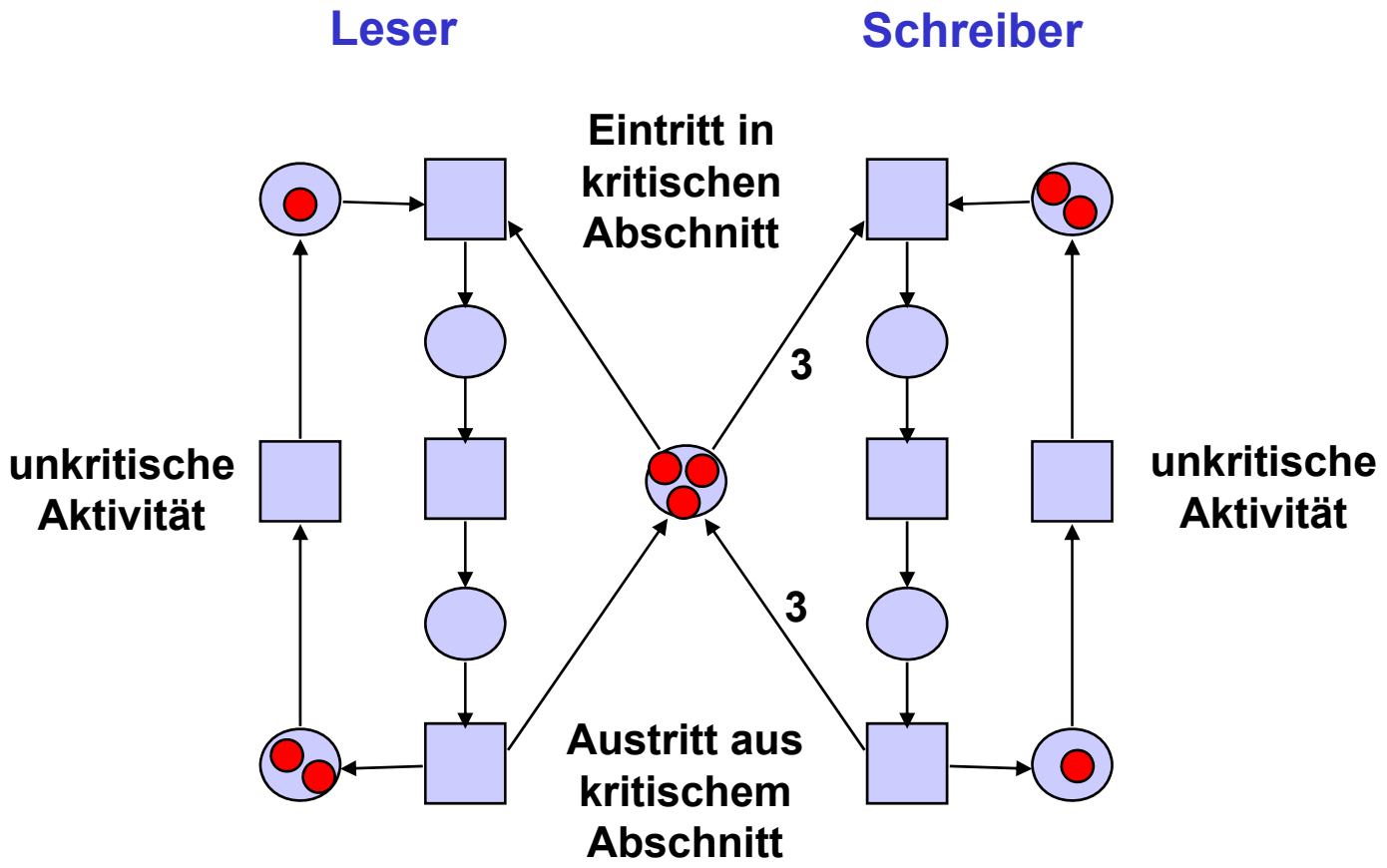
Netzmuster (3)

Gegenseitiger Ausschluss t1↔ t2



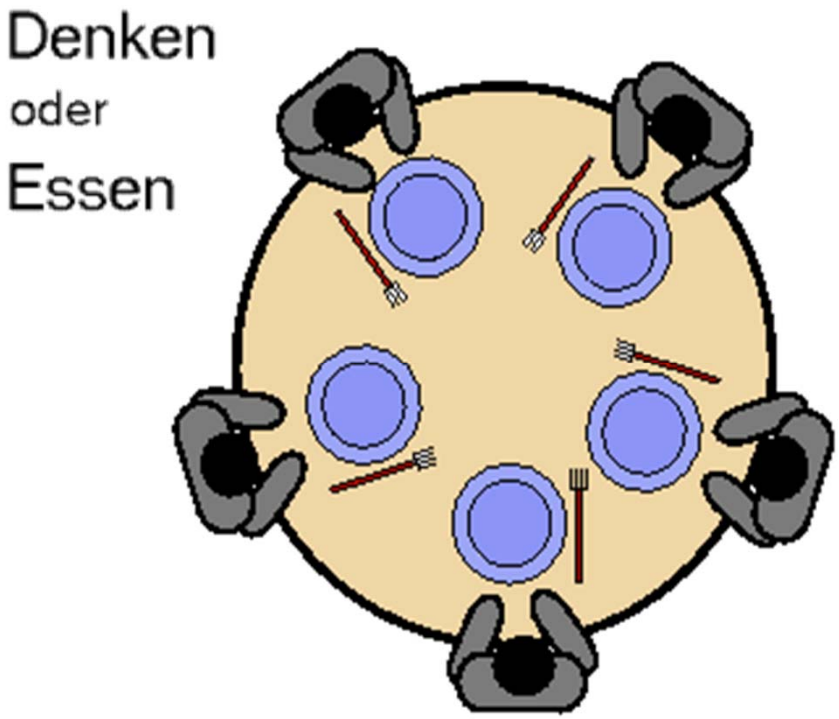
kritische Abschnitte

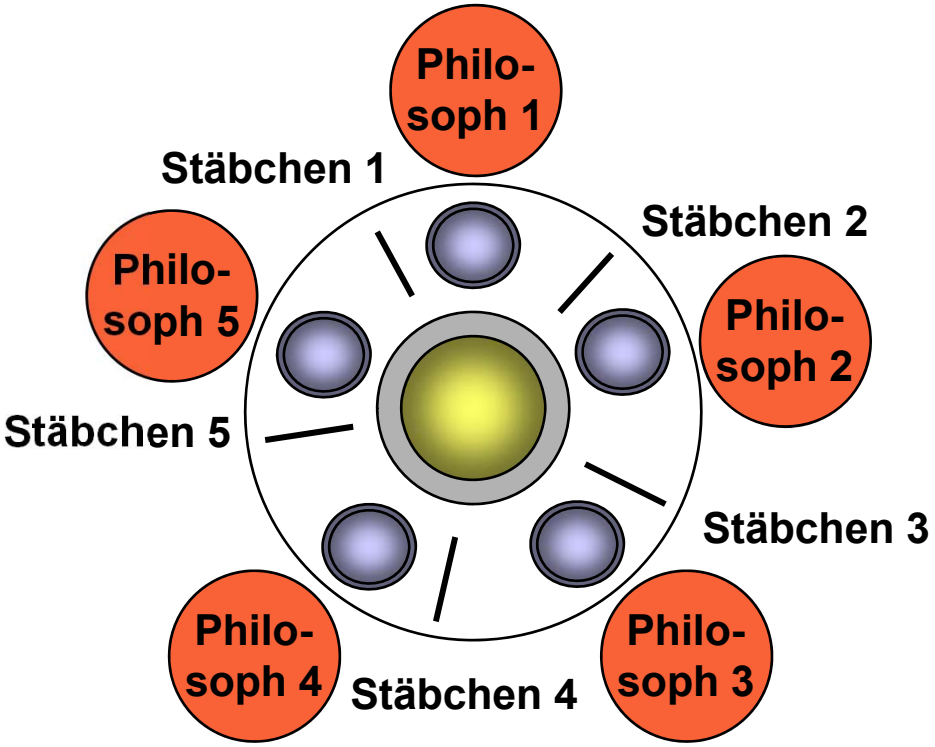
Leser und Schreiber



Markenbelegung für 3 Schreiber und 3 Leser

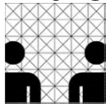
Beispiel: Die fünf speisenden Philosophen



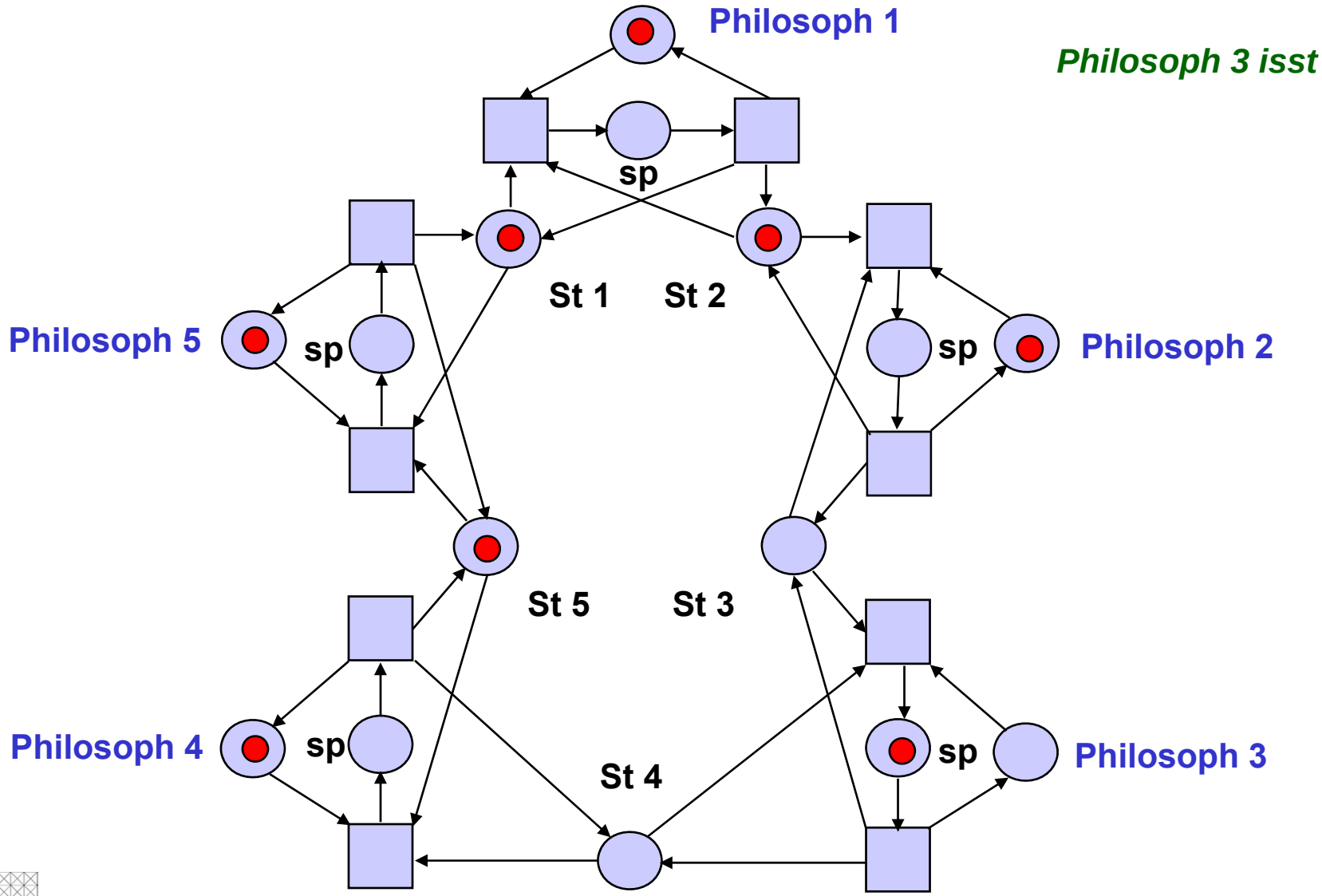


- Jeder Philosoph will entweder denken oder aus der großen Schale essen.
- Zum Essen braucht er zwei Stäbchen, sein eigenes (rechts) und das seines linken Nachbarn.

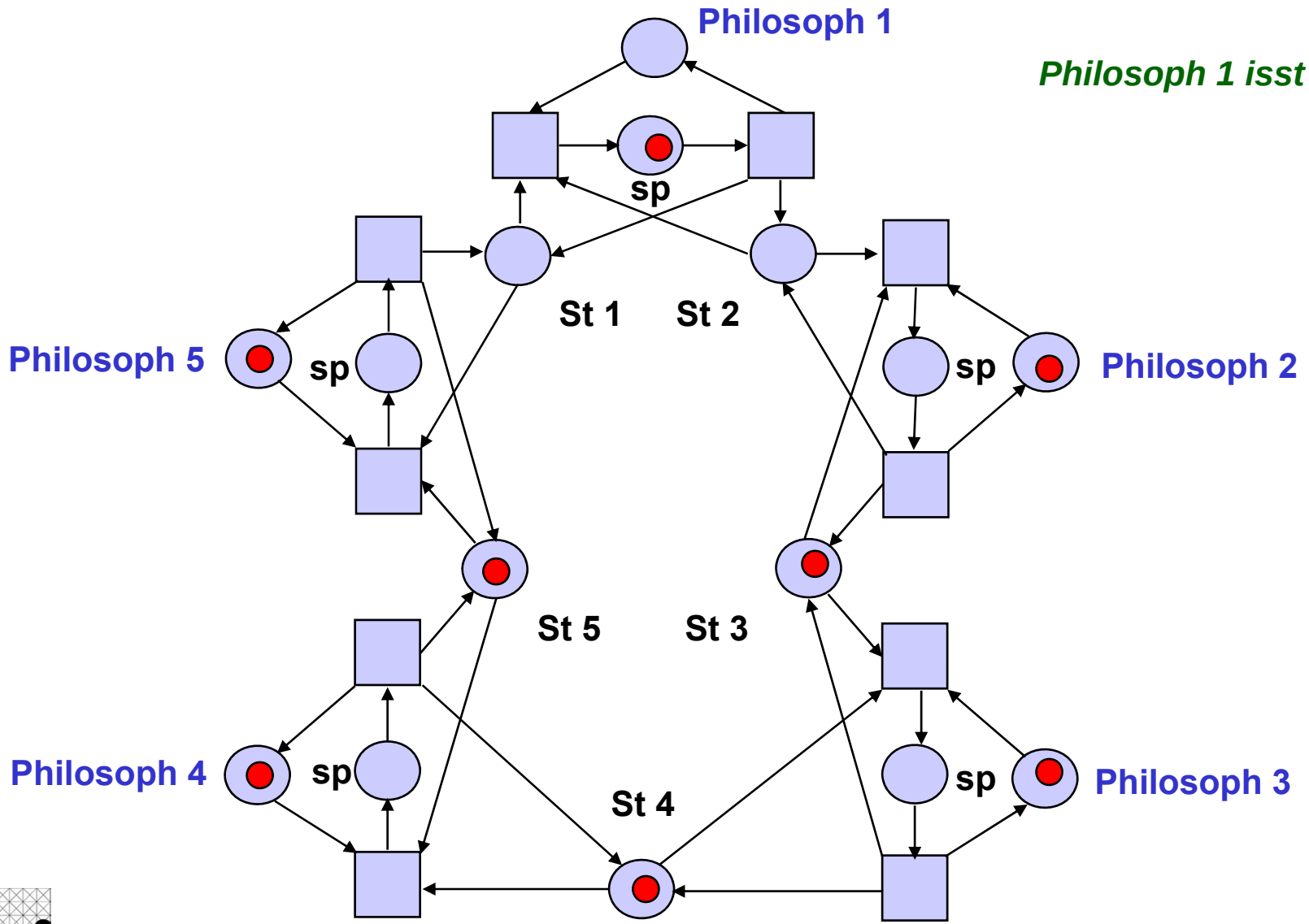
Wie können sich die Philosophen synchronisieren, so dass jeder einen fairen Anteil zu essen bekommt?



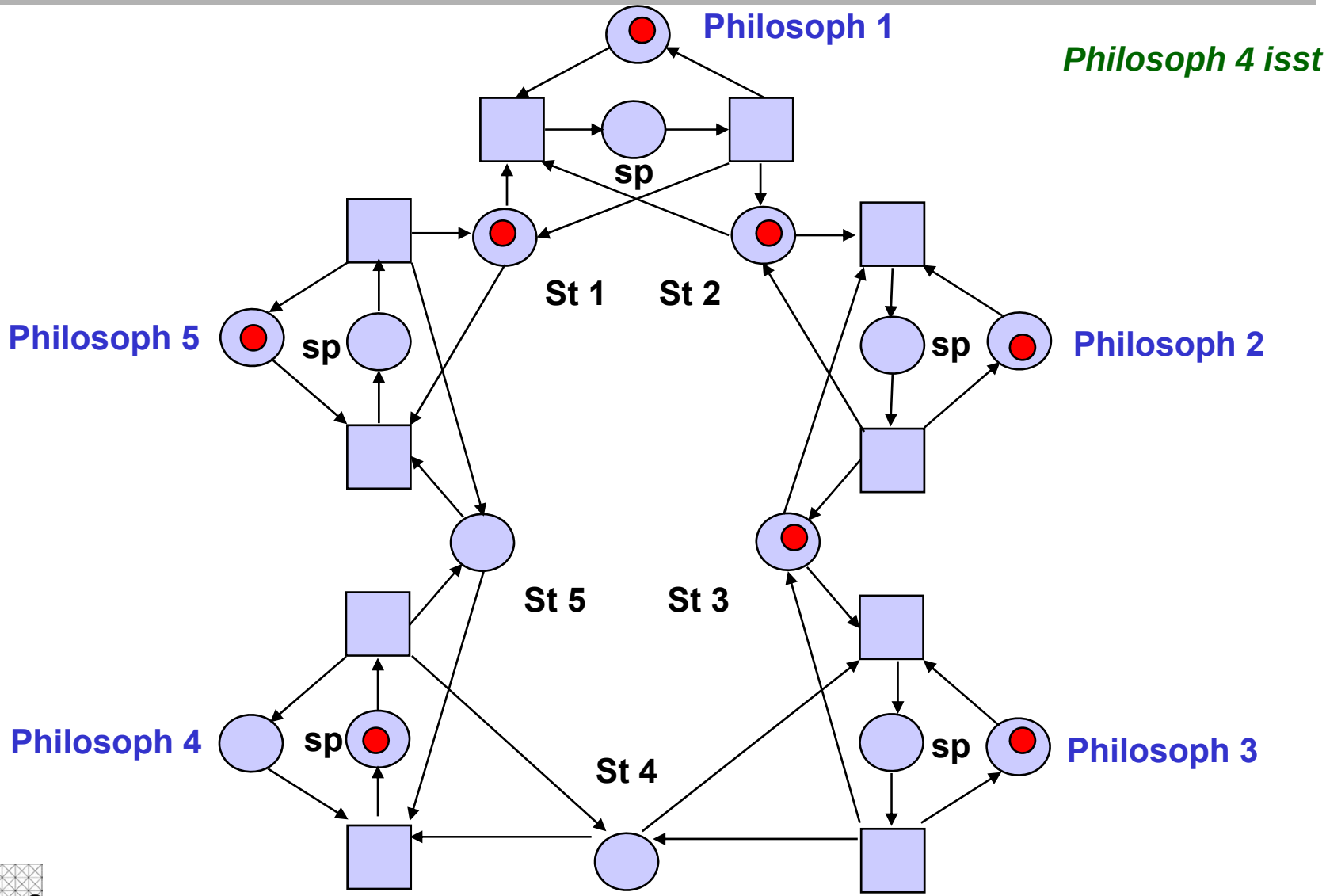
Petri-Netz für 5 speisende Philosophen (a)



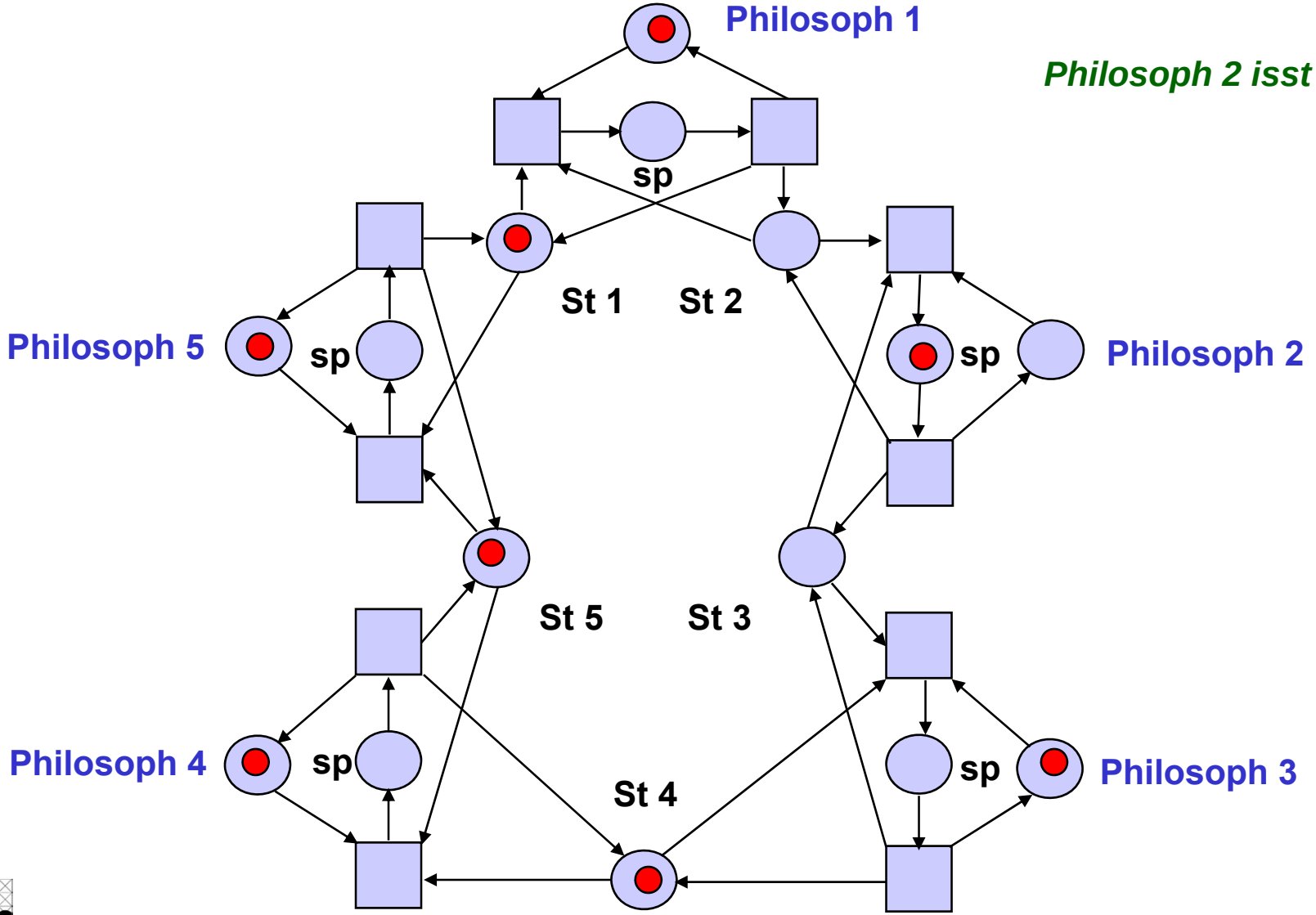
Petri-Netz für 5 speisende Philosophen (b)



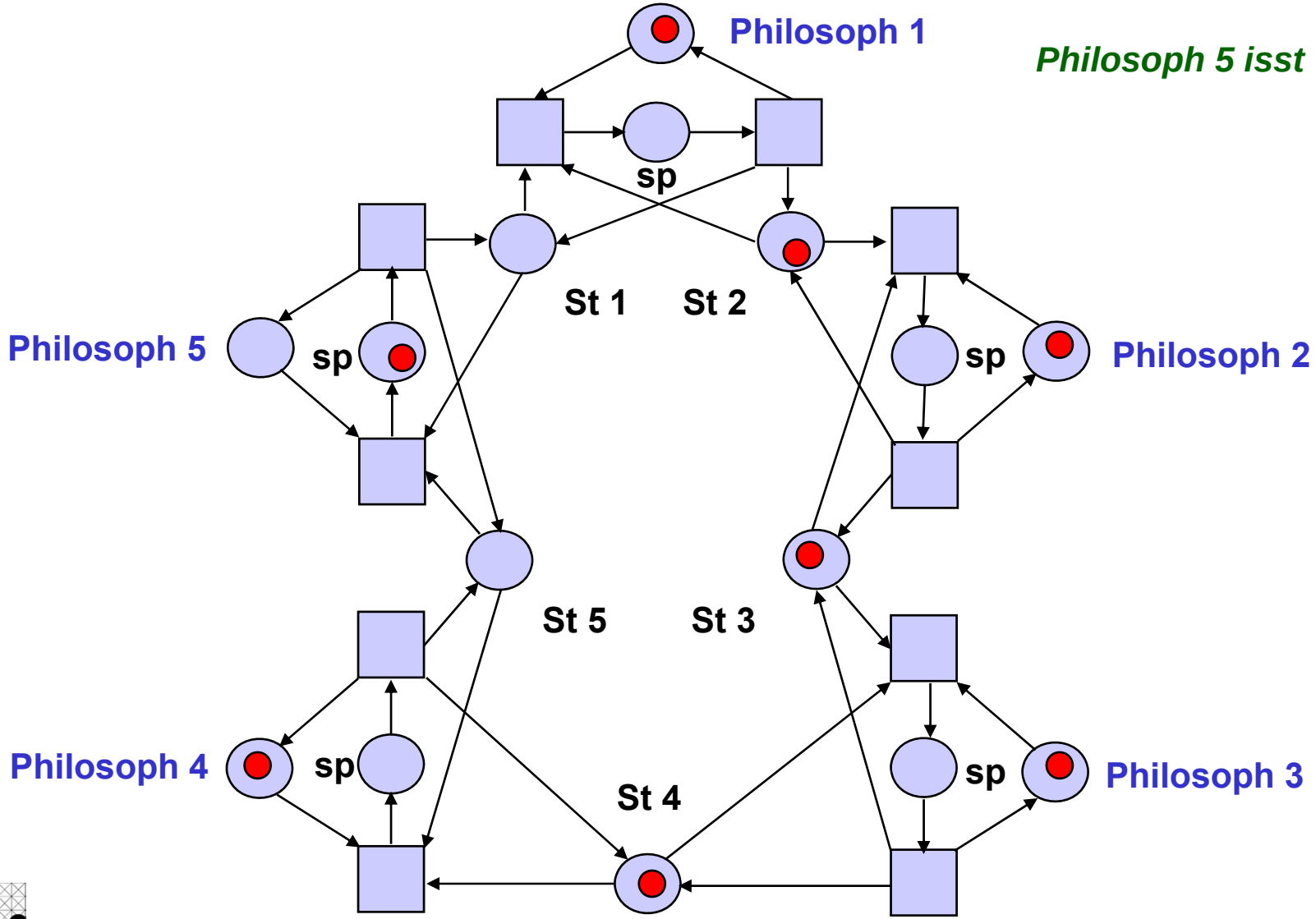
Petri-Netz für 5 speisende Philosophen (c)



Petri-Netz für 5 speisende Philosophen (d)



Petri-Netz für 5 speisende Philosophen (e)



Petri-Netz für 5 speisende Philosophen (f)