

Kurs OMSI im WiSe 2013/14

Objektorientierte Simulation mit ODEMx

Prof. Dr. Joachim Fischer
Dr. Klaus Ahrens
Dipl.-Inf. Ingmar Eveslage



fischer|ahrens|eveslage@informatik.hu-berlin.de

3. Prozess-Scheduling

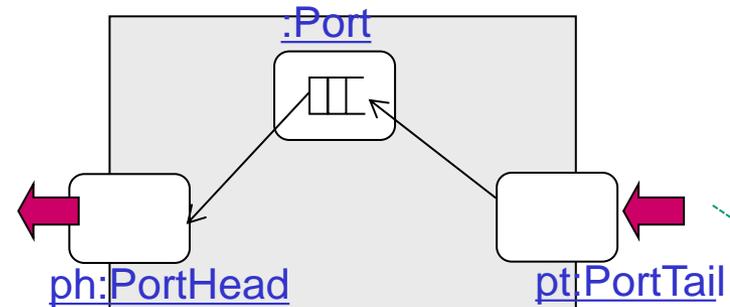
Letzte Vorlesung

1. Aufgaben von Klasse Simulation (Wdh.)
2. Process-Listen eines Simulationskontextes
3. Allgemeines Process-Scheduling
4. Weitere Process-Funktionalität
5. Prozesswarteschlangen: ProcessQueue
6. Spezielles Process-Scheduling (Memory)
7. Beispiel: Autofähre

Zusammenfassung (2): asynchroner Nachrichtenaustausch

C++ Hauptprogramm

```
PortHead *ph= new PortHead(...); // Entnahme  
PortTail *pt= ph.tail() // Einlagerung  
  
Producer *pro= new Producer(..., pt)  
pro->activate();  
  
Consumer *con= new Consumer(..., ph);  
con->activate();
```



```
main() { // Consumer-Aktionen  
...  
Msg* m= ph->get();  
...  
}
```

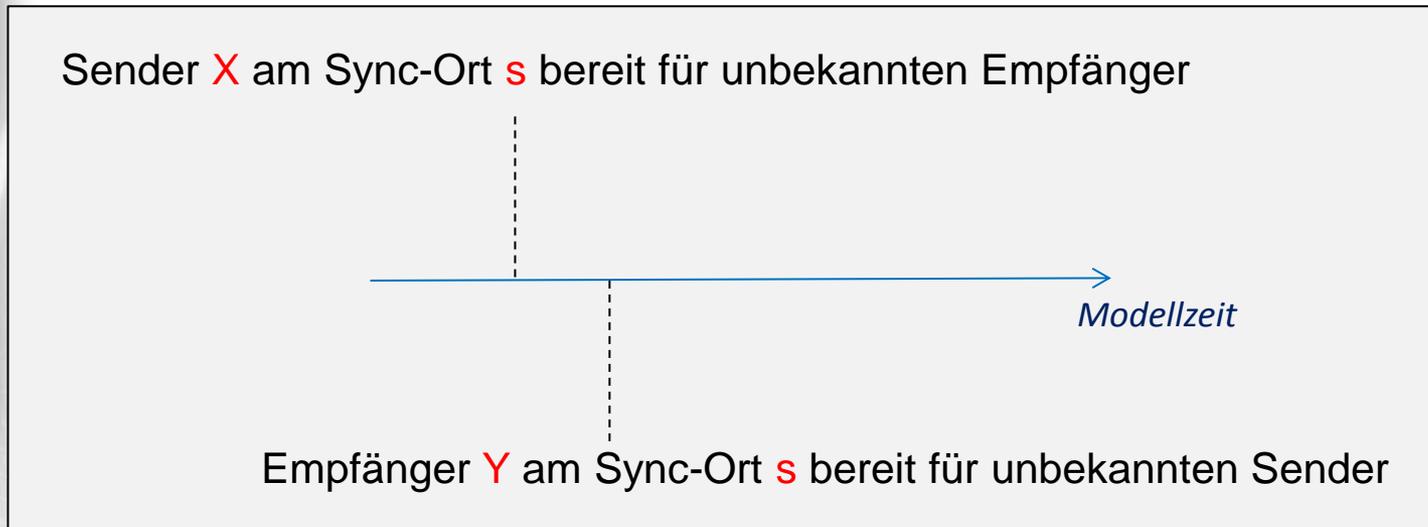
```
main() { // Producer-Aktionen  
...  
Msg* m= new Msg(...)  
pt->put (m);  
...  
}
```

put/get-Semantik ~ vom initial eingestellten Modus

default: Blockierung bei Aufnahme des blockierten Aufrufers
in Memory-Liste vom benutzten PortHead- bzw. PortTail-Objekt

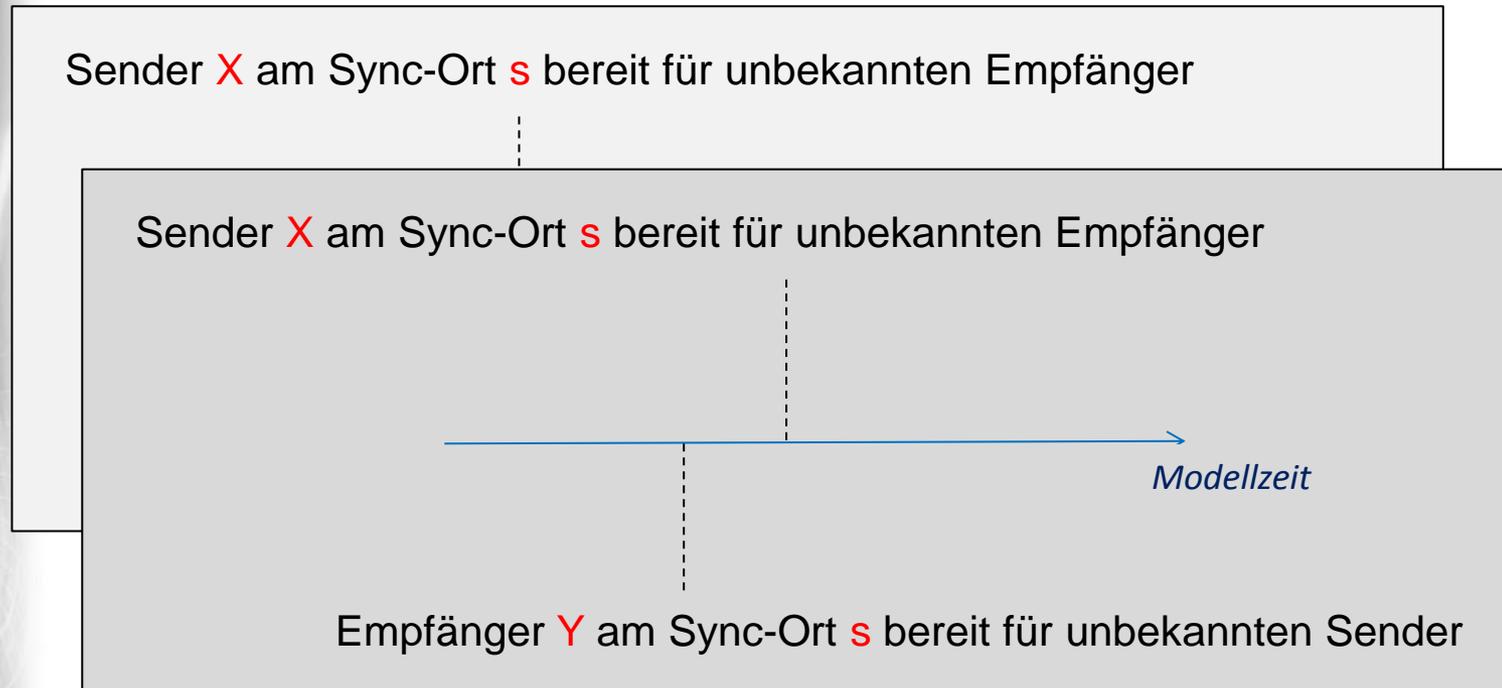
Aufgabe (Beispiel: Prüfungskarte)

- Skizzieren Sie eine (Synchronisations-) Klasse für einen **synchronen** Handshake-Informationsaustausch



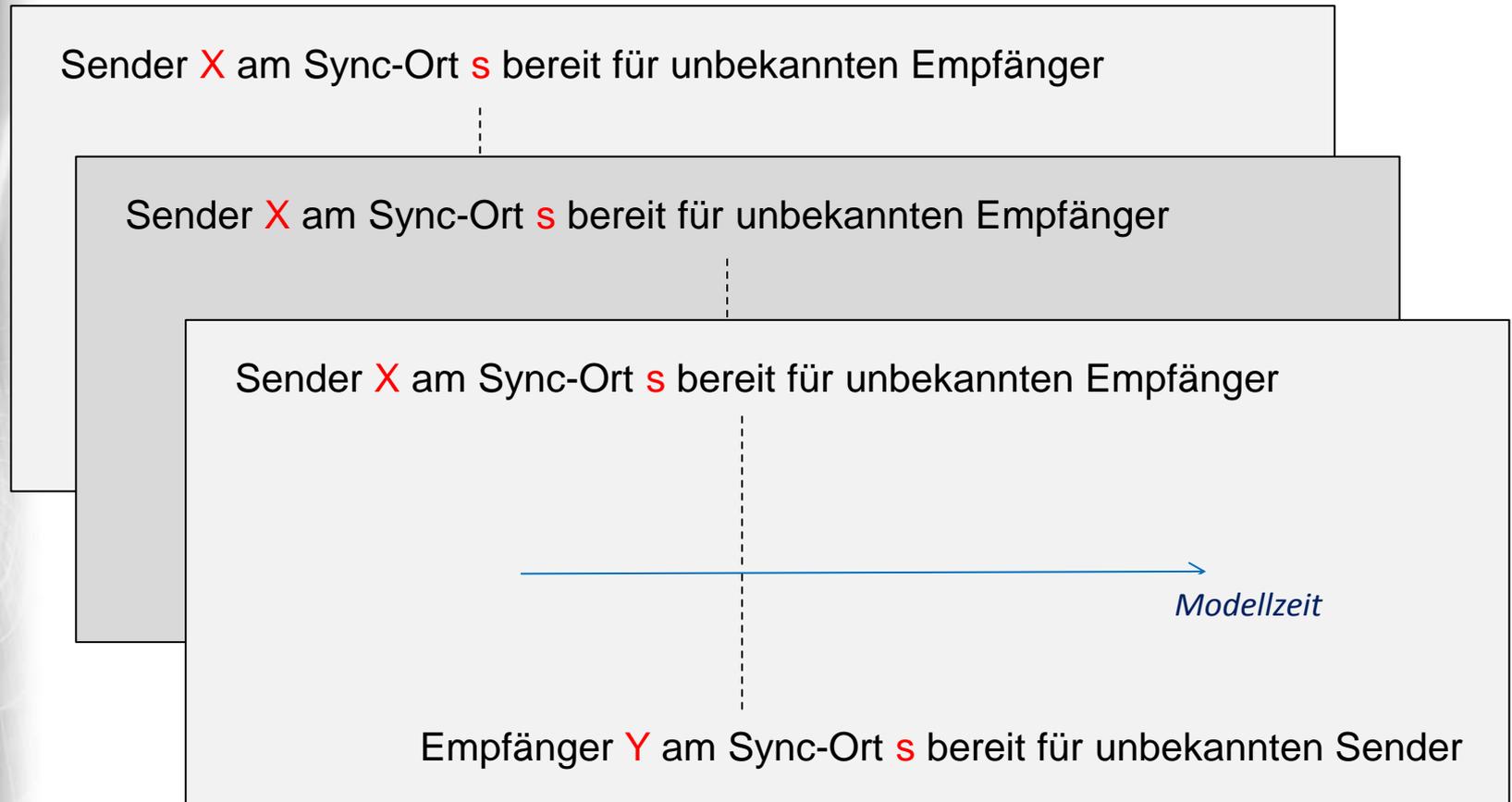
Aufgabe (Beispiel: Prüfungskarte)

- Skizzieren Sie eine (Synchronisations-) Klasse für einen **synchronen** Handshake-Informationsaustausch



Aufgabe (Beispiel: Prüfungskarte)

- Skizzieren Sie eine (Synchronisations-) Klasse für einen **synchronen** Handshake-Informationsaustausch



Klassen-Ensemble

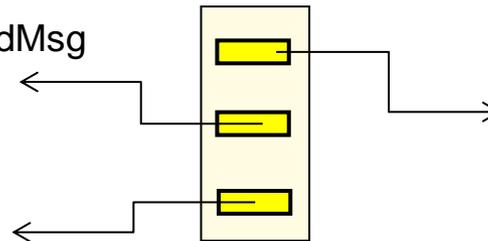
Member-Funktionen:

- **send** (Msg*)
- Msg* **receive**()

class SyncQ

Process* sender

Msg* receivedMsg



Process* receiver

Bibliothek

class UserMsg: public Msg

Anwendung

SyncQ* servicePoint;

class Producer: public Process

class Consumer: public Process

```
int main () {  
    ...  
    servicePoint-> send (new UserMsg);  
    ...  
}
```

```
int main () {  
    ...  
    UserMsg* m= servicePoint-> receive();  
    ...  
}
```

Zusammenfassung (3):

universelle wait-Funktion (**bedingte** Blockierung)

- Process-Member-Funktion `wait`

```
Memo* wait (Memo* m0,  
            Memo* m1= 0, Memo* m2= 0, Memo* m3= 0, Memo* m4= 0, Memo* m5= 0 )
```

liefert eines der Memo-Objekte zurück, sobald dieses „Verfügbarkeit“ liefert;
bis dahin bleibt der Aufrufer blockiert

- Beispiele

a)

```
PortHead *p1, *p2, *p3. *p;  
  
...  
p= wait (p1, p2, p3);  
...
```

Aufrufer-Prozess wartet(blockiert) bis in einem
der Buffer `p1`, `p2`, `p3`
eine Nachricht hinterlegt worden ist

b)

```
Memory *m;  
PortHead *ph;  
PortTail *pt;  
Timer t;  
  
...  
m= wait (ph,pt, t);  
switch (m->getMemoryType()) {  
  case TIMER: ...  
  case PORTHEAD: ...  
  default: ...  
}
```

Aufrufer-Prozess wartet(blockiert) bis in einem
der Puffer `ph` eine Nachricht abgelegt wird **oder**
in einem Puffer `pt` Platz geworden ist **oder** ein
Timeout anliegt

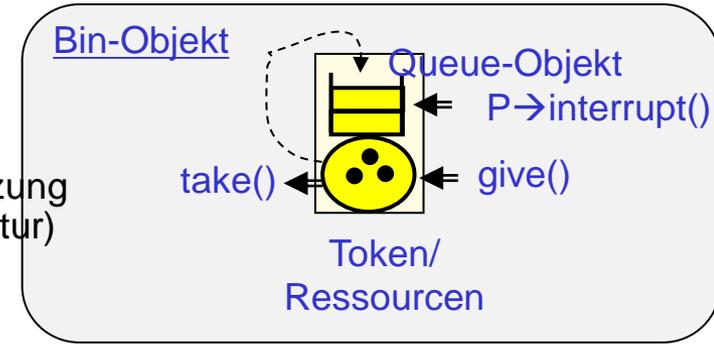
4. ODEMX-Modul Synchronisation: *Bin, Res*

- Verwaltung geteilt genutzter Ressourcen mittels *Bin* und *Res*
- Die Klasse *Bin*
- Behandlung von Unterbrechungen
- Die Klasse *Res*
- Beispiel: Autofährbetrieb (bei Einsatz von *Bin* und *Res*)

Klassen Bin und Res

Gemeinsamkeiten

- verwalten **Ressourcen** und deren geteilte Nutzung
Ressourcen sind abstrakt (Token ohne Struktur)
- bieten Funktionen zur
 - **Anforderung** und
 - **Freigabe** von Ressourcen
- ein Prozess wird in seiner Anforderung **blockiert**
und im lokalen Queue-Objekt **vermerkt**,
falls die Ressourcen in geforderter Anzahl z.Z. **nicht** zur Verfügung stehen
- ein blockierter Prozess wird **fortgesetzt**,
sobald die von ihm benötigte Anzahl von Token zur Verfügung steht
bei Entnahme der angeforderten Token



Achtung: Token sind keine Objekte
(d.h.: ohne Identität, Zustand u. Verhalten)

Unterschiede

- **Bin** kann **beliebig viele** Token (unabhängig von der initialen Ausstattung) aufnehmen, erlaubt dynamische „Vernichtung“ und „Generierung“ von Token
- **Res** **beschränkt** die maximal verfügbare Token-Menge, geht i.d.R. von einer unveränderlichen Token-Menge je **Res**-Objekt aus

4. ODEMx-Modul Synchronisation: *Bin, Res*

- Verwaltung geteilt genutzter Ressourcen mittels Bin und Res
- Die Klasse Bin
- Behandlung von Unterbrechungen
- Die Klasse Res
- Allgemeine Prozesslokalisierung (nicht Bin/Res-spezifisch)
- Ein Beispiel: Autofährbetrieb (Einsatz von Bin und Res)

Klasse Bin: Konzept

bisherige Beispiele

put - get

send – receive

- besitzt zur Verwaltung blockierter Prozesse (privates) `Queue`-Objekt:

`Queue* takeWait`

- per Konstruktorparameter wird initiale Tokenzahl (≥ 0) vermittelt

(Defaultwert 0)

typisches wiederkehrendes Muster
von Geben- und Nehmen

- Member-Funktion `int take(n), n>0`

- Rufer (**Nehmer-Rolle**) wartet evtl. mit Null-Zeit als „Durchläufer“
- i.d.R. wartet der Rufer solange (d.h. ohne Unterbrechung), bis `n` Token verfügbar sind
- aber: Warteaktion ist prinzipiell durch nebenläufigen Prozess unterbrechbar
 - Rückgabewert zeigt erfolgte Unterbrechung an: 0 (sonst `n`)

- Member-Funktion `give(n), n>0`

- Rufer (in **Geber-Rolle**) veranlasst (zeitlose) Eingabe oder Rückgabe von Token, verbunden mit Aktivierung evtl. wartender Nehmer-Prozesse
- Token müssen nicht zwingend demselben Bin-Objekt zurück gegeben werden

Bin- Semantische Präzisierung

- Prozesse, die auf verfügbare Token warten, werden in einer Warteschlange (lokales Objekt von Bin) erfasst
FCFS-Strategie
- bedeutet für folg. angenommenen Fall:
 - Bin-Objekt **o** habe aktuell **2** Token
 - **P1** wartet am längsten (benötigt **3** Token) in **o**
 - **P2** wartet ebenfalls in **o** (benötigt **1** Token)

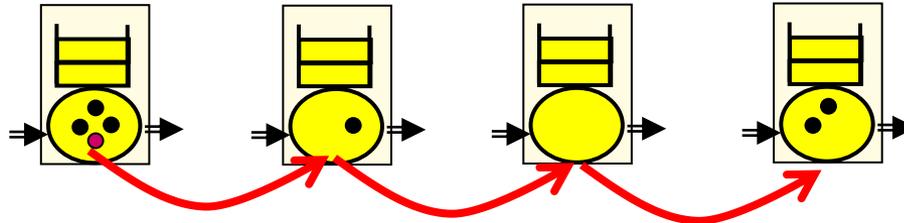
P2 wäre mit der verfügbaren Tokenzahl von **o** zufrieden, darf **P1** dennoch **nicht** überholen

Bin-Anwendungen

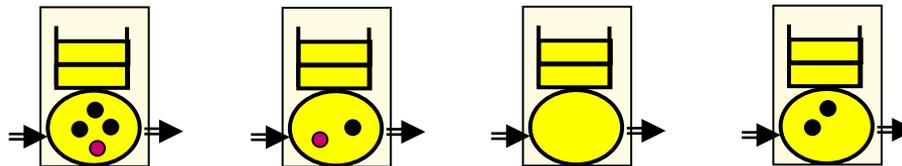
Token haben jedoch
bei einfachen Bins keine Identität

Später:
Bin-Templates

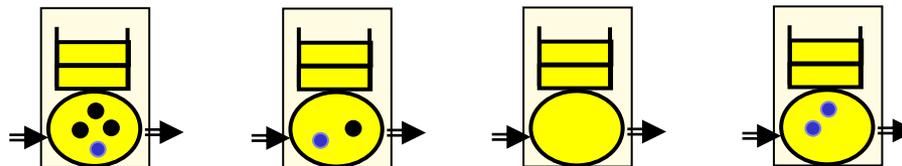
Basistyp
der Token
kann
als Parameter
vermittelt
werden



en lassen sich von Prozessen durch Bin-Objekt-Ketten ,bewegen‘



Token lassen sich von Prozessen in Bin-Objekt-Ketten erzeugen
(ohne dass sie zuvor irgendwo entnommen wurden)



Token lassen sich von Prozessen in Bin-Objekt-Ketten vernichten
(ohne dass sie danach irgendwo abgelegt werden)

Klasse Bin (Public-Member-Funktionen)

Public Member Functions

```
Bin (base::Simulation &sim, const data::Label &label, std::size_t initialTokens,  
      BinObserver *obs=0)
```

```
~Bin ()
```

```
const  
base::ProcessList & getWaitingProcesses () const  
//Get list of blocked processes.
```

Token management

```
std::size_t take (std::size_t n)  
Take n token.
```

```
void give (std::size_t n)  
// Give n token.
```

```
std::size_t getTokenNumber() const  
// Number of tokens available.
```

Klasse Bin (Private-Attribute)

private:

```
Simulation* env_  
unsigned int tokens_  
unsigned int initTokens_  
...;  
unsigned int users_  
unsigned int providers_  
double sumWaitTime_
```

// process management

```
Queue takeQueue_;
```

*Aufnahme blockierter
Prozesse*

Implementierung von Bin::take

```
std::size_t Bin::take( std::size_t n ) {
    // resource handling only implemented for processes
    if( ! getCurrentSched() || getCurrentSched()->getSchedType() != base::Sched::PROCESS ) error ...
    base::Process* currentProcess = getCurrentProcess();
    // compute order of service, insert the process into the queue
    takeQueue_.inSort( currentProcess );
    // if not enough tokens or not the first process to serve
    if( n > tokens_ || currentProcess != takeQueue_.getTop() ) {
        ...
        // statistics
        base::SimTime waitStart = getTime();
        // block execution
        while( n > tokens_ || currentProcess != takeQueue_.getTop() ) {
            currentProcess->sleep();
            if( currentProcess->isInterrupted() ) {
                takeQueue_.remove( currentProcess );
                return 0;
            }
        }
        // block released here
        // statistics, log the waiting
        ...
    }
    // remove from list
    takeQueue_.remove( currentProcess );
    // awake next process
    awakeFirst( &takeQueue_ );
    return n;
}
```

*aktiviert seinen Nachfolger,
dieser wird Verfügbarkeit für sich prüfen*

Implementierung von Bin::give

```
void Bin::give( std::size_t n ) {  
    // trace  
    ODEMX_TRACE << log ... ;  
    // observer  
    ...;  
    // release tokens  
    tokens_ += n;  
    // trace  
    ODEMX_TRACE << log ... ;  
    // statistics  
    ...;  
    // observer  
    ...;  
    // awake next process  
    awakeFirst( &takeQueue_ );  
}
```

aktiviert nur den unmittelbar nächsten Nachfolger

*sollten mehrere Prozesse hintereinander in der Wartschlange durch die Token-Rückgabe ihre jeweiligen Token-Forderungen fortgesetzt werden können, ist deren Aktivierung durch sukzessiven Vollzug der take-Operation beginnend mit diesem einen Nachfolger gesichert
(siehe take-Operation)*

4. ODEMX-Modul Synchronisation: Bin, Res

- Verwaltung geteilt genutzter Ressourcen mittels Bin und Res
- Die Klasse Bin
- **Behandlung von Unterbrechungen**
- Die Klasse Res
- Beispiel: Autofährbetrieb (Einsatz von Bin und Res)

Unterbrechung

...

- a) des **Wartevorgangs** auf verfügbare Ressourcen (Token)
- b) des anwendungsspezifischen zeitlich begrenzten **Nutzungsvorgangs** der entnommenen Ressourcen (Token)

in beiden Fällen mittels **interrupt** !

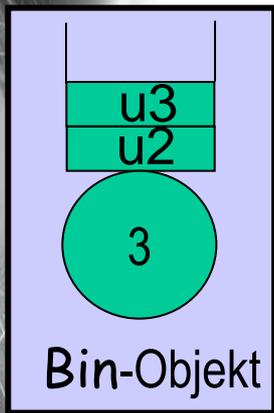
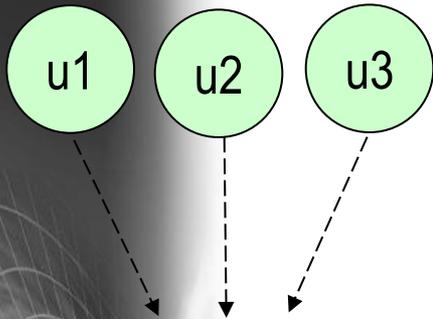
```
int res= servicePoint-> take(3); // liefert 0 Token, falls
                                // Blockierung unterbrochen worden ist
                                // sonst Anzahl der entnommenen Token

holdFor (...);                  // Verzögerung um Nutzungszeit der entnommenen
                                // Token

if (interrupted() ) { ...};

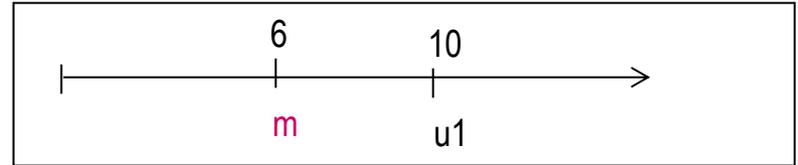
servicePoint-> give(3);          // Rückgabe von Token
```

Variante A: Unterbrechung des Wartevorgangs



Nutzer,
die gleichzeitig
gestartet werden

ein weiterer Prozess **m**
zum Zeitpunkt 6.0



```
Bin *servicePoint= new Bin ("service", 3);
```

```
class User: public Process {
    int no;
    User (int n), no(n), Process (...) { ...};

```

```
int main () {
    int ret;
```

```
    ret= servicePoint-> take (2);
    if (ret) {
        holdFor (10.0);
        servicePoint-> give (2);
    } else ....

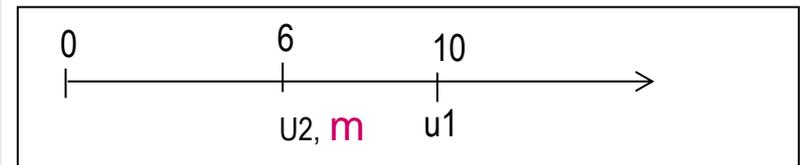
```

```
}
}
```

evtl. Nutzung eines anderen
Bin-Objektes nach Warteabbruch

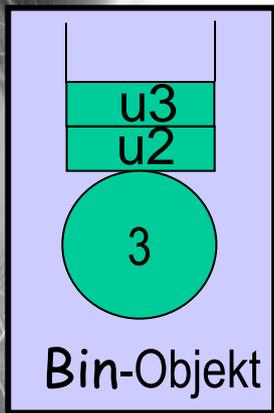
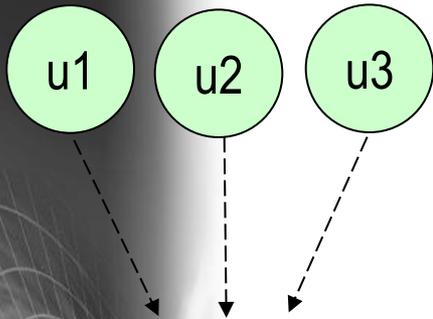
```
class Manager:
    public Process {
    ...

    int Manager::main() {
    ...
        u2->interrupt();
    ...
    }
}
```



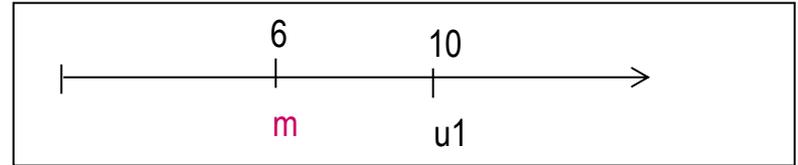
Unterbrechung

Variante B: Unterbrechung des Nutzungsvorgangs



Nutzer,
die gleichzeitig
gestartet werden

ein weiterer Prozess **m**
zum Zeitpunkt 6.0



```
Bin *servicePoint= new Bin ("service", 3);
```

```
class User: public Process {
    int no;
    User (int n), no(n), Process (...) { ...};

```

```
int main () {
    int ret;
    ...
    ret= servicePoint-> take (2);
    if (ret) {
        holdFor (10.0);
        servicePoint-> give (2);
    } else ....

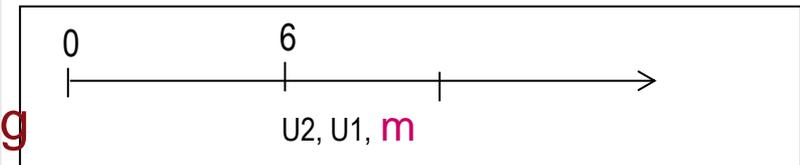
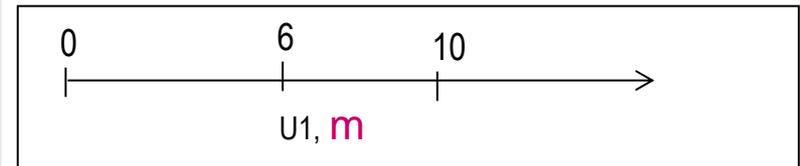
```

Unterbrechung

evtl. Nutzung eines anderen
Bin-Objektes nach Warteabbruch

```
class Manager:
    public Process {
    ...
    int Manager::main() {
    ...
        u1->interrupt();
    ...
    }
}

```



4. ODEMX-Modul Synchronisation: Bin, Res

- Verwaltung geteilt genutzter Ressourcen mittels Bin und Res
- Die Klasse Bin
- Behandlung von Unterbrechungen
- Die Klasse Res
- Beispiel: Autofährbetrieb (Einsatz von Bin und Res)

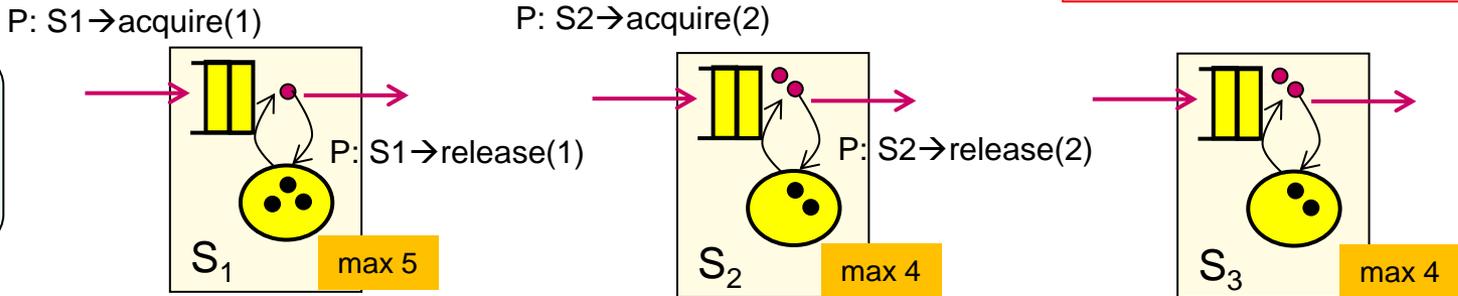
Besonderheiten von Res

- **Konstruktor**
sowohl die **initiale** (≥ 0) als auch **maximale** Tokenzahl (> 0) ist einzustellen
- **unsigned int acquire(n), n > 0**
 - unterbrechbare Warteaktion (evtl. mit Null-Zeit als „Durchläufer“)
 - ohne Unterbrechung wartet der Rufer solange, bis **n** Token verfügbar sind
 - Rückgabewert zeigt Unterbrechung an: **0**, sonst **n**
- **unsigned int release(n), n > 0**
 - Eingabe von Token
 - Token müssen nicht unbedingt diesem **Res**-Objekt vorab entnommen sein
 - wird maximale Tokenanzahl überschritten:
Fehlermitteilung und Fortsetzung bei Ignorierung der überschüssigen Token
- **Sonderfunktionen zur dynamischen Korrektur der maximalen Token-Zahl**
 - **void control(n), n > 0** Erhöhung der maximalen Token-Anzahl bei Aktivierung des nächsten wartenden Prozesses
 - **void uncontrol(n), n > 0** Reduktion falls **n** Token überhaupt noch verfügbar sind, sonst Anpassung von **n**

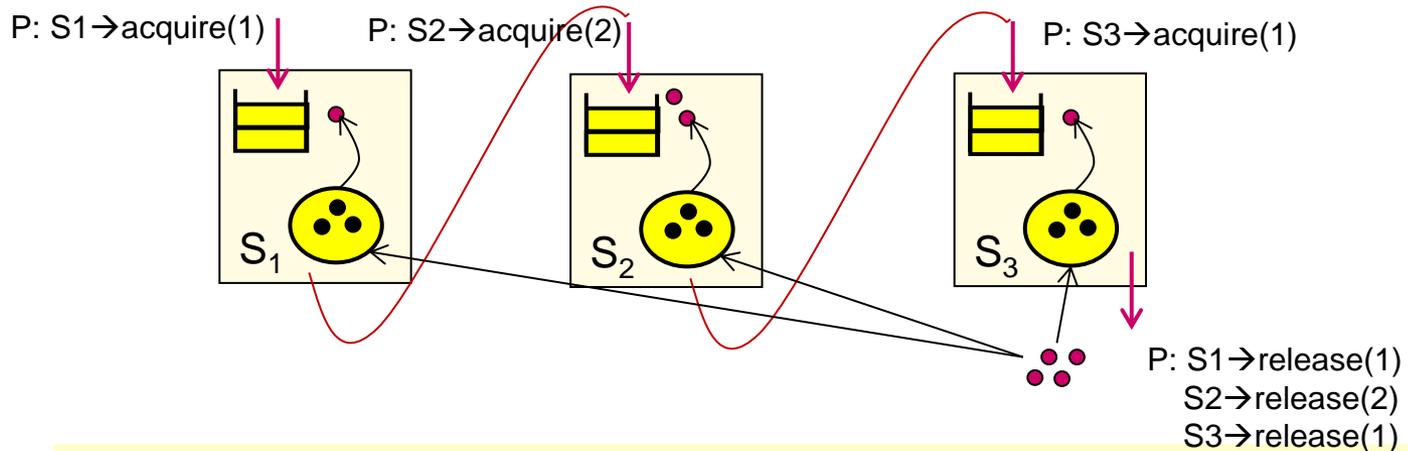
Muster
vom Geben- und Nehmen

Res-Anwendungen

Token haben jedoch weder Identität noch Typ



Fall a) Prozesse bewegen sich durch Stationsketten (Res-Objekte), benötigen/benutzen Ressourcen einer Ressource und geben diese nach der Nutzung an die aktuelle Station **komplett** zurück



Fall b) Prozesse benötigen Ressourcen unterschiedlicher Sorten (Res-Objekte) für einen Arbeitsgang

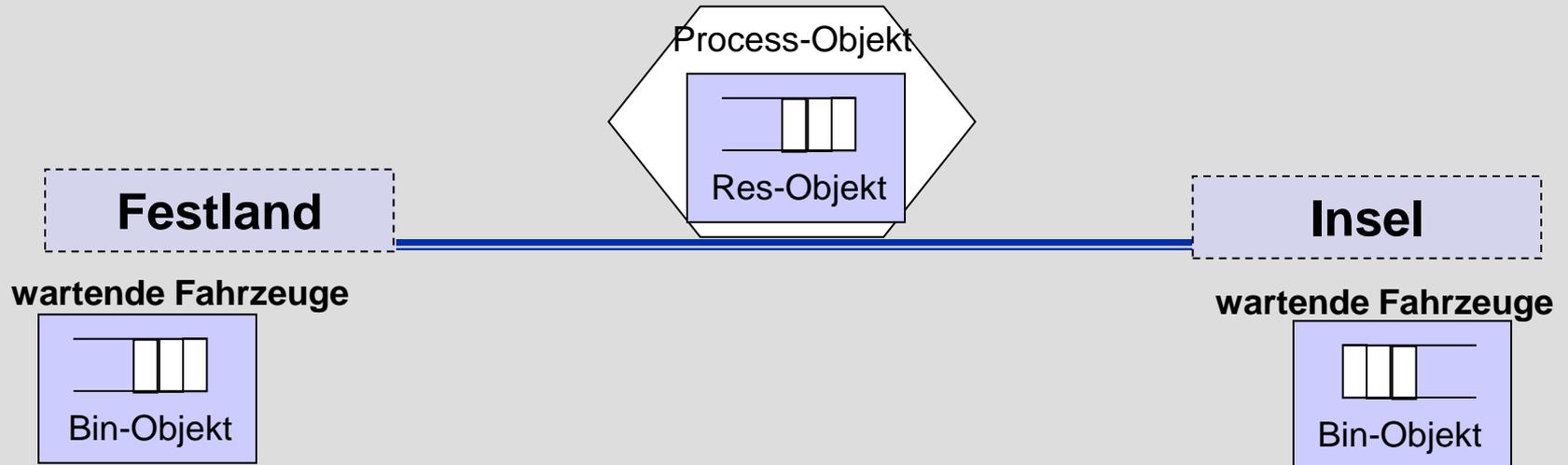
4. ODEMX-Modul Synchronisation: *Bin, Res*

- Verwaltung geteilt genutzter Ressourcen mittels Bin und Res
 - Die Klasse Bin
 - Behandlung von Unterbrechungen
 - Die Klasse Res
- Beispiel: Autofährbetrieb (Einsatz von Bin und Res)

Beispiel: Autofähre

- Erfassung von Bewertungsgrößen

1. Auslastung der Fähre (Res-Objekt)
2. Beladungsprofil (Tally-Objekt)



3. separate Zählung der Leer- und Frachtfahrten (Count-Objekte)
4. Benutzungsprofile der Bin-Objekte

5. ODEMx-Modul Random

1. Charakterisierung von Zufallsgrößen
2. Approximation von Zufallszahlen
3. ODEMx- Zufallszahlengeneratoren (Übersicht)
4. Einstellung von Startwerten
5. Protokollierung
6. Berechnung von Zufallszahlen ausgewählter Verteilungen

Bedeutung von Pseudozufallszahlen *in der Simulation*

- Einflüsse der Systemumgebung oder Systemabläufe selbst unterliegen häufig dem **Zufall**
- Simulationsergebnisse sind dann als **Stichprobe** eines statistischen Experiments anzusehen
- Auf der Grundlage vieler Stichproben (**Stichprobenraum**) können statistische Kennwertprofile und Konfidenzaussagen (Aussagen zur Zuverlässigkeit der Ergebnisse) abgeleitet werden
- **enorm wichtig:** der Einfluss des Zufalls muss im Simulationsexperiment **wiederholbar** dargestellt werden können (Test von Simulationsmodellen)
 - Verwendung von sogenannten **Pseudozufallszahlen**

Zufallszahlen im Original und Modell

1. Realität \leftrightarrow Modell

reale Zufallsgrößen werden durch mathematische Modelle (Funktionen) approximiert:

- *Verteilungsfunktion, Dichtefunktion, statistische Kenngrößen*

2. Modell $\leftarrow \rightarrow$ Modell

es bestehen mathematische Zusammenhänge zwischen einzelnen Verteilungsfunktionen:

- *beliebige Verteilungsfunktionen lassen sich durch $(0,1)$ -gleichverteilte Verteilungsfunktionen approximieren*

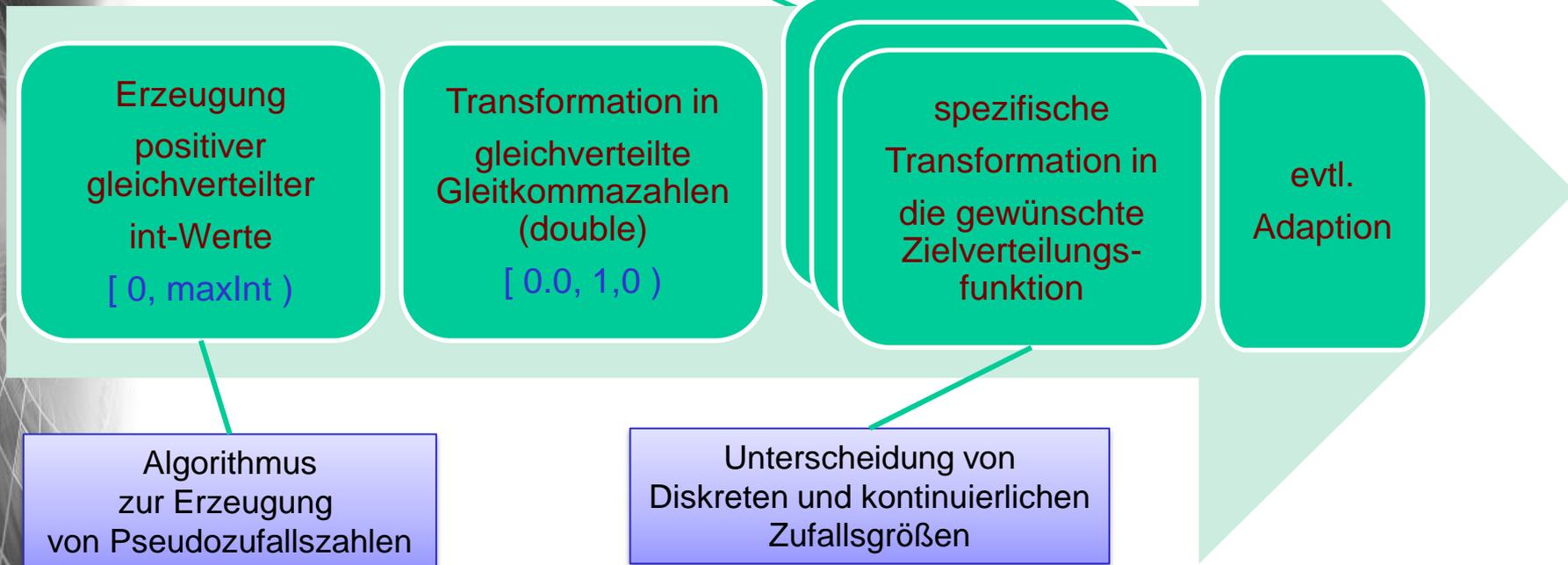
3. Modell $\leftarrow \rightarrow$ Berechnungsmodell

determiniert berechnete Folgen von $(0,1)$ -Werten lassen sich als Approximationen

von Verteilungsfunktionen realer Zufallsgrößen verwenden

Schema zur Berechnung von Zufallszahlen

- Theoretische Verteilungsfunktionen
- Empirische Verteilungsfunktionen
- (Bezier-Verteilungsfunktionen)



Algorithmus zur Erzeugung von Pseudozufallszahlen

Unterscheidung von Diskreten und kontinuierlichen Zufallsgrößen

Zufallsgrößen

Zufallsgrößen:= zufällige Ereignisse --> Zahlen

reale Zufallsgrößen und ihre Verteilungsfunktionen

Diskrete Zufallsgrößen:= Größen, die endliche oder abzählbar-unendlich viele verschiedene Werte annehmen können

Beispiel:

- Auszählen der Stillstände einer Maschine während einer Werksschicht
- Registrierung der Anzahl von Gesprächen in einer Telefonvermittlung

Stetige Zufallsgrößen:= Größen, die jeden beliebigen Wert innerhalb eines Intervalls der Zahlengerade annehmen können

Beispiel:

- Durchmesser von Antriebswellen (nach Bearbeitung an einem Drehautomaten): alle Werte innerhalb eines vorgeschriebenen Toleranzbereiches

Verteilungsfunktion einer Zufallsgröße

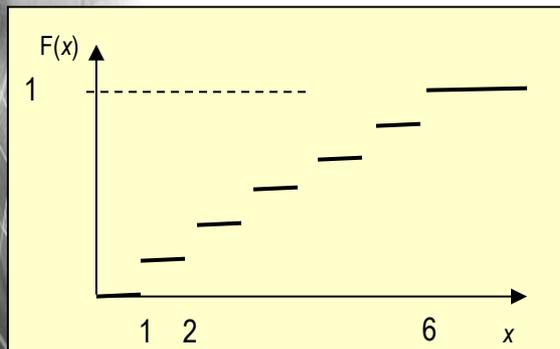
Charakterisierung einer Zufallsgröße X:

- X nimmt bei jedem Versuch zufällig einen bestimmten Wert an
- Werte genügen einer Verteilungsfunktion

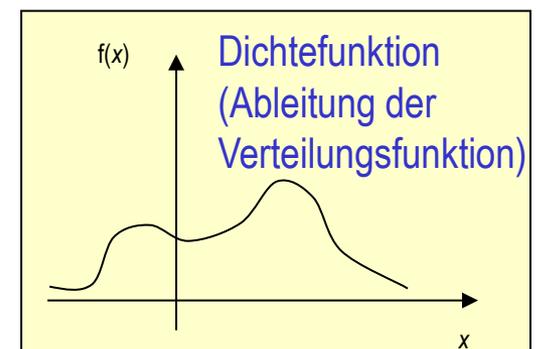
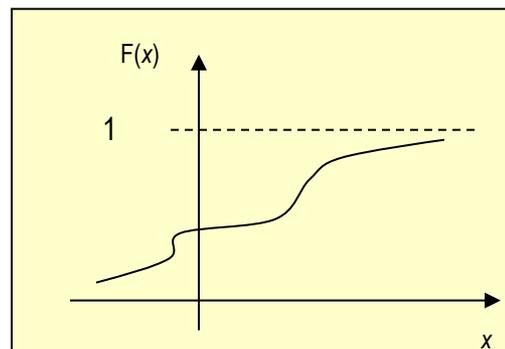
Verteilungsfunktion: $F_X(x) = P(X \leq x)$,
der Wert von F_X (Verteilungsfunktion der Größe X) ist
an der Stelle x ist **gleich der Wahrscheinlichkeit**,
dass X einen Wert unterhalb von x annimmt.

x durchläuft alle Werte der reellen Zahlengerade

diskret



kontinuierlich



Verteilungsfunktion als kumulative Dichtefunktion

Objektorientierte Simulation mit ODEMX

Diskrete Zufallsgrößen

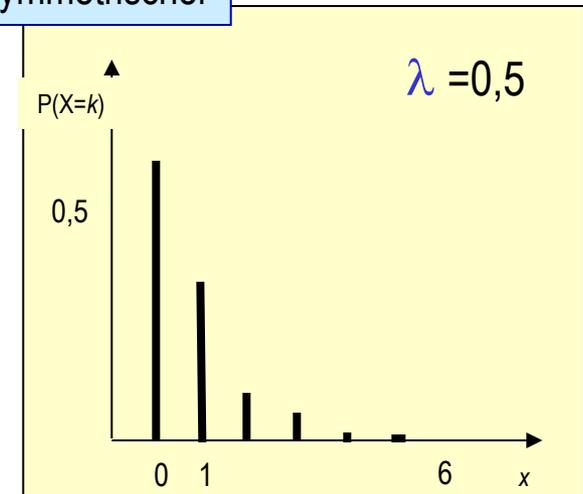
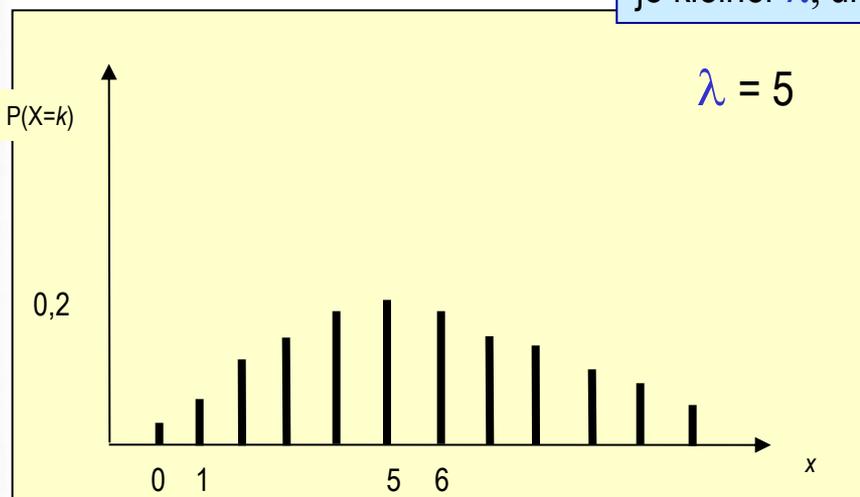
Charakterisierung einer diskreten Zufallsgröße X :

- X beschreibt Ereignisregistrierungen (Anzahl) in einem bestimmten Zeitbereich (X nimmt bei jedem Versuch zufällig einen best. Wert an)
- Werte genügen einem gleichen Typ von Verteilungsfunktion (**Poisson**)
- λ ist die mittlere Anzahl

Beispiel-Ereignisse

- Anzahl beobachteter Sternschnuppen in einer bestimmter Zeitspanne
- Anzahl registrierter Telefonanrufe in einer best. Zeitspanne
- Anzahl von **Ankünften von Kunden einer Service-Einrichtung** in best. Zeitspanne

je kleiner λ , umso unsymmetrischer



Verteilungsfunktion einer diskreten Zufallsgrößen

Poisson-Verteilung:

beschreibt Ereignisregistrierungen (Anzahl)
in einem bestimmten Zeitbereich

$$\text{Verteilungsfunktion:} = F_X(x) = \sum_{k < x} P(X=k) = \sum_{k < x} \lambda^k / k! * e^{-\lambda},$$

falls $x > 0$, sonst 0

$\mu = \lambda$, Erwartungswert
 $\sigma^2 = \lambda$, Streuung

Wichtig: Poisson-Verteilung steht im Zusammenhang
mit der Exponentialverteilung:

*Sei X poisson-verteilte Zufallsgröße mit Erwartungswert λ ,
dann ist die Zwischenankunftszeit zweier aufeinanderfolgender Ereignisse
exponential-verteilt mit Erwartungswert $1/\lambda$*

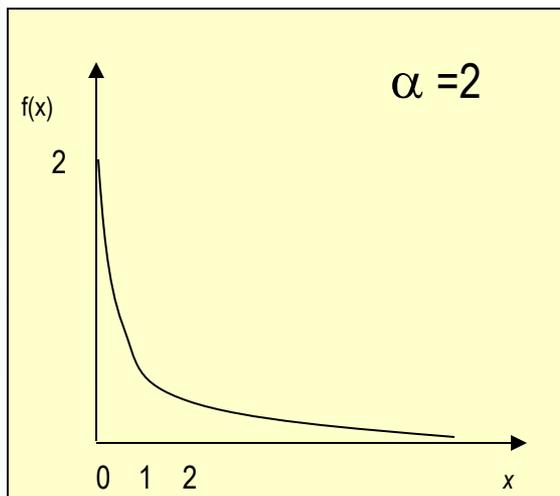
Stetige Zufallsgrößen

Beispiel einer stetigen Zufallsgröße X:

- X nimmt bei jedem Versuch zufällig einen bestimmten Wert an
- Werte genügen einer (negativen) Exponential-Verteilungsfunktion

Dichtefunktion $f(x) = \alpha e^{-\alpha x}$, falls $x \geq 0$, sonst 0

Verteilungsfunktion $F(x) = 1 - e^{-\alpha x}$, falls $x \geq 0$, sonst 0



- *Zeitabstände zwischen Ankunftsereignissen von Ringstapeln in einer Schicht*
- *Dauer von Telefongesprächen*
- *Lebensdauer von Lebewesen, Maschinen, ...*

$$\mu = 1/\alpha$$
$$\sigma^2 = 1/\alpha^2$$

5. ODEMx-Modul Random

1. Charakterisierung von Zufallsgrößen
2. Approximation von Zufallszahlen
3. ODEMx- Zufallszahlengeneratoren (Übersicht)
4. Einstellung von Startwerten
5. Protokollierung
6. Berechnung von Zufallszahlen ausgewählter Verteilungen

Pseudo-Zufallszahlen

Warum sind reproduzierbare Zufallszahlen enorm wichtig für die Simulation?

Iterationsverfahren

x_0 - beliebig aus $[1, 10]$, z.B.: 2

$$x_{k+1} \equiv 2 * x_k \bmod 11 \quad (k= 0, 1, 2, \dots)$$

→ Zahlenfolge mit Periode p :

$x_0 = 4, 8, 5, 10, 9, 7, 3, 6, 1, 2, 4, \dots$ (Wiederholung)

In Abhängigkeit vom Startwert ergibt sich eine periodische Folge von determinierten Zahlen, die als Zufallszahlenfolge interpretiert werden kann

Ein einfacher Pseudozufallszahlengenerator

```
class Random {  
    int u;  
    Random( int uu) u(uu) {  
        if u < 1 or u>10 { error(...); ...};  
    }  
    double next () {  
        u:= 2*u;  
        if u>11 { u= u-11; };  
        next= double(u) / 11.0;  
    }  
}
```

Individuelle Startwerte: eigene Folgen von Zufallszahlen

```
Random *s1, *s2, *s3;  
s1= new Random(2); s2= new Random(9); s3 = new Random(7);
```

```
s1->next(); // .636, .273, .545, ...  
s2->next(); // .364, .727, .455, ...  
s3->next(); // .273, .545, .091, ...
```

Linearer Generator

Generator für gleichverteilte 26-Bit-Zufallswerte

$$q = 67.099.547 = 2^{26} - 1$$

Kongruenzmethode

- multiplikative Variante (Iterationsverfahren mit Startwert x_0)
 $x_{j+1} \equiv k x_j \pmod{q}$

$$x_{j+1} \equiv 8192 x_j \pmod{67.099.547} \quad (j = 0, 1, 2, 3, \dots)$$

→ Zahlenfolge mit **sehr großer** Periode $p = 67.099.546$:

$$x_0, x_1, x_2, \dots, x_{p-1}, x_p = x_0, x_1, x_2, \dots$$

Algorithmus für Startwerte (unabhängiger) Generatoren

$$u_0 = 907 \quad (\text{erster Startwert eines Generators})$$

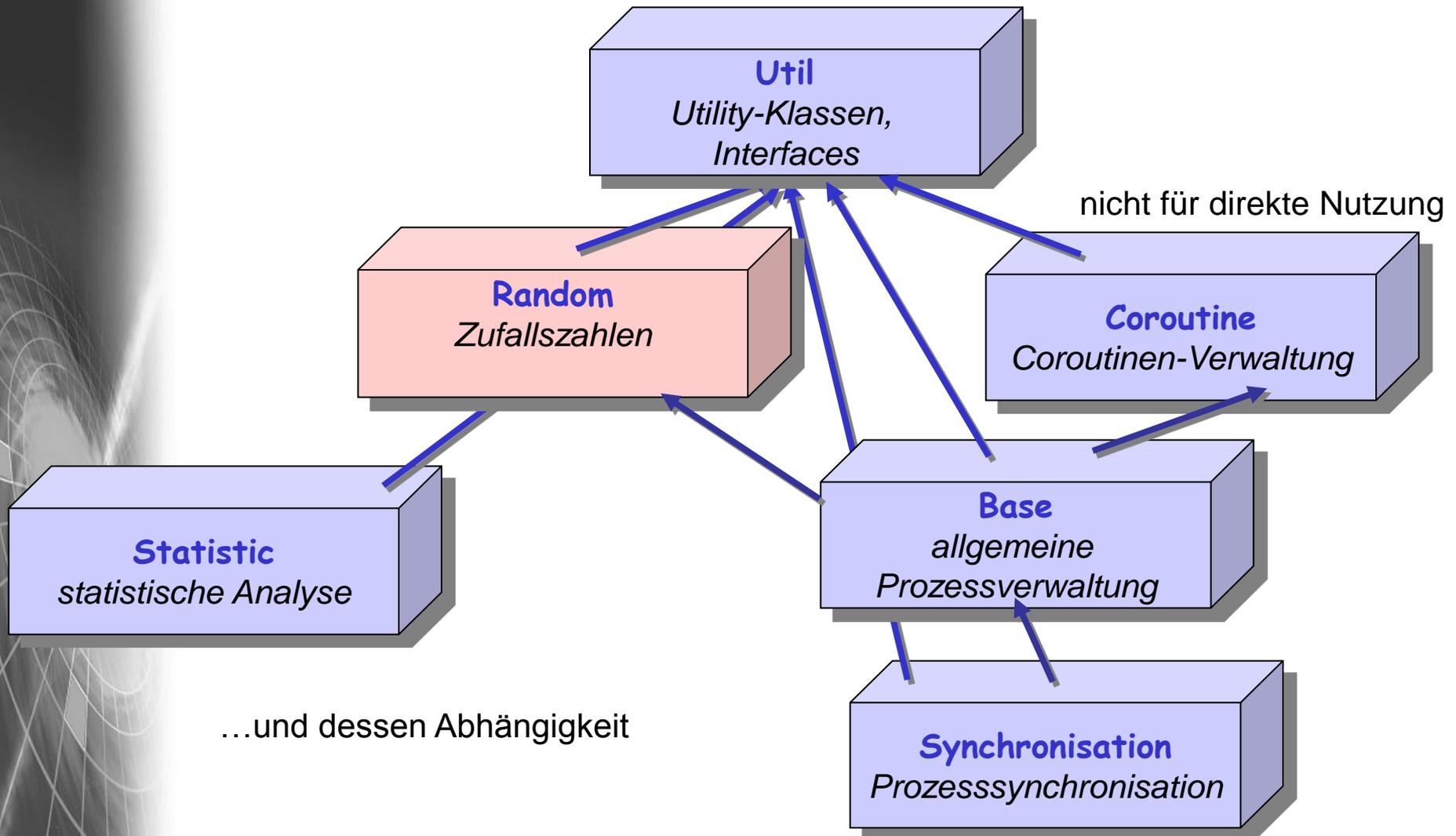
$$u_{k+1} \equiv 36.855 * u_k \pmod{67.099.547}$$

→ 500 Generatoren für unabhängige Zahlenfolgen der Länge 120.000 (ohne dass eine Folge, in die andere „hineinläuft“)

5. ODEMx-Modul Random

1. Charakterisierung von Zufallsgrößen
2. Approximation von Zufallszahlen
3. ODEMx- Zufallszahlengeneratoren (Übersicht)
4. Einstellung von Startwerten
5. Protokollierung
6. Berechnung von Zufallszahlen ausgewählter Verteilungen

Der ODEMx- Modul Random



...und dessen Abhängigkeit

Klassen des Random-Moduls

Dist ist **abstrakte** Basisklasse aller Zufallszahlengeneratoren

- bietet ganzzahligen Generator (lineares Kongruenzverfahren) für (0,1)- gleichverteilte Zufallsgrößen
- **Dist**- Ableitungen transformieren (0,1)- Folge in Folgen verschiedener Verteilungsfunktionen

