Kurs OMSI
im WiSe 2014/15

Objektorientierte Simulation mit ODEMx

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

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

3. Prozess-Scheduling

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

Rückgabewert von wait (Wdh.)

m ~polymorpher Memory-Zeiger

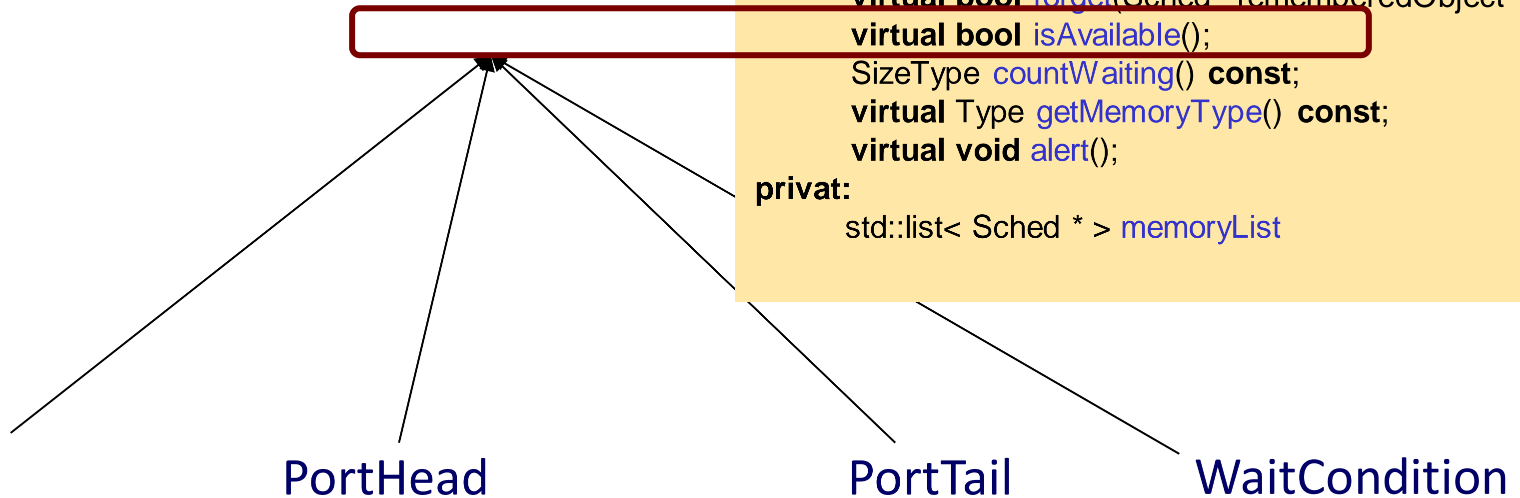
```
…
*Memory  m= wait (m1, m2, m3);
…
```

- Aufrufer-Prozess vermerkt sich in der jeweiligen lokalen Process*-Liste von m_1, m_2 und m_3, falls deren „Verfügbarkeit“ nicht gegeben ist und blockiert
- wird auf den per wait blockierten Prozess durch einen parallelen Prozess ein **interrupt**() angewendet, verlässt dieser die m_1-, m_2- und m_3-Liste

  und beendet die **wait()**-Anweisung mit der Rückgabe des NULL-Zeigers (**externe Unterbrechung** der Blockierung)
- Sobald die „Verfügbarkeit“ von mindestens einem m_i gegeben ist, wird **wait** mit Rückgabe von m_i verlassen (der Prozess hat zuvor die lokalen Listen von m_1, m_2 und m_3 verlassen)
- Sollten mehr als zwei m_i‘s „Verfügbarkeit“ anzeigen, liefert **wait** den ersten von ihnen (nach Position in der Parameterliste)

Memory-Funktionalität (Wdh.)

```
class Memory {
public:
        Memory( …);
        virtual ~Memory();

        virtual bool remember(Sched* newObject );
        virtual bool forget(Sched* rememberedObject );
        virtual bool isAvailable();
        SizeType countWaiting() const;
        virtual Type getMemoryType() const;
        virtual void alert();
privat:
        std::list< Sched * > memoryList
```



Timer	PortHead	PortTail	WaitCondition
returns true if this timer is not scheduled	**returns** true if the buffer is not empty	**returns** true if the buffer is not full	**returns** condition-result
blockierte Prozesse warten auf Timeout-Signal	blockierte Empfangs-prozesse warten auf Nachrichtenverfügbarkeit	blockierte Sende-prozesse warten auf Pufferverfügbarkeit	blockierte Prozesse warten auf Bedingungserfüllung

Zusammenfassung

- 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

```
PortHead *p1, *p2, *p3, *p;
```

dynamic_cast <PortHead*> (...)

```
...
        p= wait (p1, p2, p3);
Msg* m= p->get();
        ....
```

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

```
Memory *m;
PortHead *ph;
PortTail *pt;
Timer t;
```

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

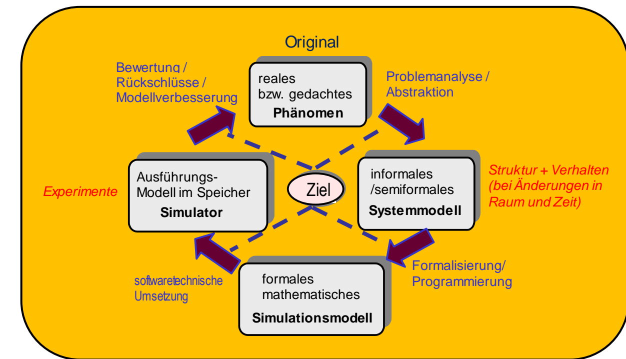
Aufrufer-Prozess wartet(blockiert) bis in einem der Puffer ph eine Nachricht abgelegt wird **oder** in einem Puffer pt Platz geworden ist **oder** ein Timeout anliegt

Warum wird get()-Aufruf hier niemals blockieren?

3. Prozess-Scheduling

1. Aufgaben von Klasse Simulation (Wdh.)
2. Process-Listen eines Simulationskontextes
3. Allgemeines Process-Scheduling
4. Weitere Process-Funktionaliät
5. Prozesswarteschlangen: ProcessQueue
6. Spezielles Process-Scheduling (Memory)
7. Beispiel: Autofähre (nur Entwurf)
8. Synchrone Prozesskommunikation (Handshake)

Beispiel: Autofähre



- Wortmodell u. informale Darstellung

begrenze Aufnahmekapazität

1 ZE = 1min

Autofähre

Festland

Insel

wartende Fahrzeuge

zufällige Fahrzeit

wartende Fahrzeuge

zufällige Ankunftszeiten

zufällige Ankunftszeiten

Vorschrift

- bei Ankunft fahren zunächst alle Autos von der Fähre
- dann fahren wartende Autos auf die Fähre
- mindestens 15 min verweilt die Fähre an einer Anlegestelle

Simulationsziel

- Simulation eines Tages
- separate Zählung von Fahrten und Leerfahrten
- Auslastung der Fähre

Memory (*abstract*)
Boolean isAvailable()=0

PortHead
Boolean isAvailable()

PortTail
Boolean isAvailable()

Timer
Boolean isAvailable()=0

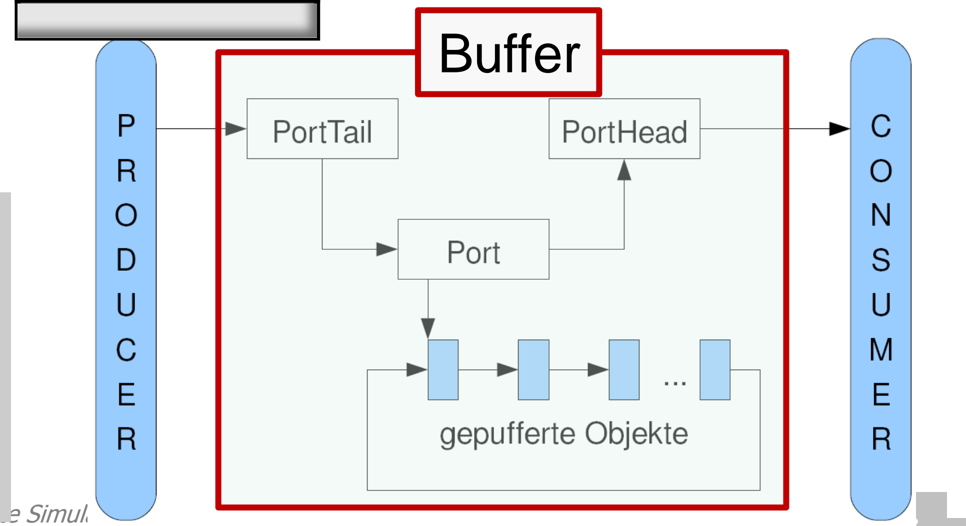
PortData (*abstract*)

Process (*abstract*)
virtual int main()=0

Arrival
int main()

Ferry
int main()

Vehicle



Attribut-Ausstattung

- “unbegrenzter” Vehicle-Puffer für Arrival
- “begrenzter” Vehicle-Puffer für Ferry
- Timer-Objekt für Ferry
- Position für Ferry

Skizze

Faehre zwischen Ort[x] u. Ort[nachfolgerVon(x)]

```
while (ferry->pt->size() >0) {
            v= ferry->pt>get();
            holdFor(…);
            //freigabe von v
}
```

```
do {
    goAhead= false;
    m= wait(faehreVoll, clock, amOrt[x]->ph);
    switch (m) {
        case PortHead:
                    v= amOrt[x]->ph->get()
                    fahre->pt->put(v);
                    holdFor(…);
                    goAhead= true;
                    break;
        case Timer: break;
        case Condition: break;
} while (goAhead);
```

```
holdFor (…);
x= nachfolgerVon(x);
```

FahrzeugAnkunft am Ort[x]

```
v= new Vehicle(…);
amOrt[x]->put(v);
dt= unbestimmte Zeit;
holdFor (dt);
```

FahrzeugAnkunft am Ort[nachfolgerVon(x)]

```
v= new Vehicle(…);
amOrt[x]->put(v);
dt= unbestimmte Zeit;
holdFor (dt);
```

Arrival

int main()

Ferry

int main()

3. Prozess-Scheduling

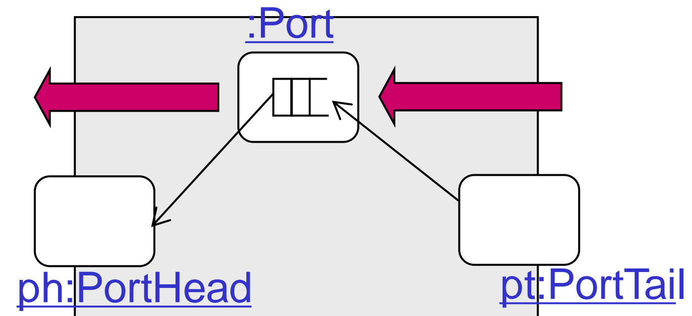
1. Aufgaben von Klasse Simulation (Wdh.)
2. Process-Listen eines Simulationskontextes
3. Allgemeines Process-Scheduling
4. Weitere Process-Funktionaliät
5. Prozesswarteschlangen: ProcessQueue
6. Spezielles Process-Scheduling (Memory)
7. Beispiel: Autofähre
8. Interrupt von wait, get, put
9. Synchrone Prozesskommunikation (HandsHake)

3. Prozess-Scheduling

1. Aufgaben von Klasse Simulation (Wdh.)
2. Process-Listen eines Simulationskontextes
3. Allgemeines Process-Scheduling
4. Weitere Process-Funktionaliät
5. Prozesswarteschlangen: ProcessQueue
6. Spezielles Process-Scheduling (Memory)
7. Beispiel: Autofähre (nur Entwurf)
8. Synchrone Prozesskommunikation (Handshake)

Asynchroner Nachrichtenaustausch

```
//Puffer-Konstruktion
PortHead *ph= new PortHead(…); // Entnahme
PortTail *pt= ph.tail()              // Einlagerung

Producer *pro= new Producer(…, pt)
pro->activate();


Consumer *con= new Consumer(…, ph);
con->activate();
```



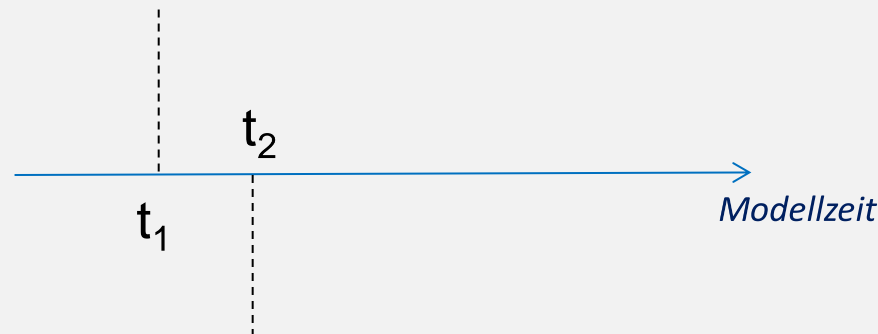

```
main() { // Consumer-Aktionen
    …
   Msg* m= ph->get();
   …
}
```

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

Producer und Consumer werden durch Nachrichtenpuffer **entkoppelt**

1. Aufgabe: Synchroner Nachrichtenaustausch

- Entwurf einer Klasse zur Realisierung eines synchronen Handshake-Informationsaustausches



- X wartet (t_2-t_1)ZE
- X übergibt an Y Information
- X und Y setzten zur Zeit t_2 ihre Lebensläufe fort

Lösung als Spezialisierung von PortHead, PortTail, PortData

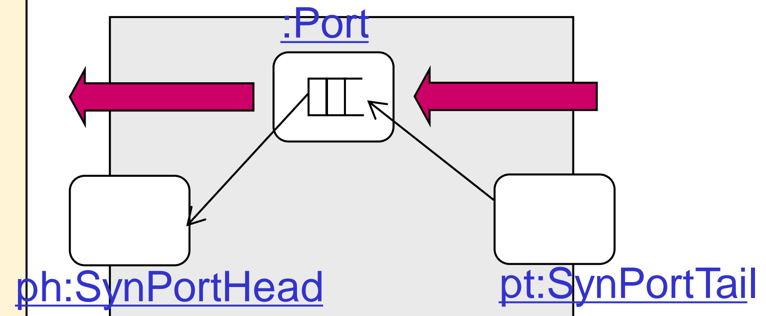
syn_get() syn_put()

Lösung (Skizze)

1

```
class Message: public PortData {
      Process* sender;
      …
}
```

```
//Puffer-Konstruktion
PortHead *ph= new SynPortHead(…); // Entnahme
SynPortTail *pt= ph.syntail()            // Einlagerung

Producer *pro= new Producer(…, pt)
pro->activate();


Consumer *con= new Consumer(…, ph);
con->activate();
```



2

```
main() { // Producer-Aktionen
   …
   Msg* m= new Message();
   pt->syn_put (m);
   …
}
```

```
m->sender= this;
put(m);
passivate();
```

3

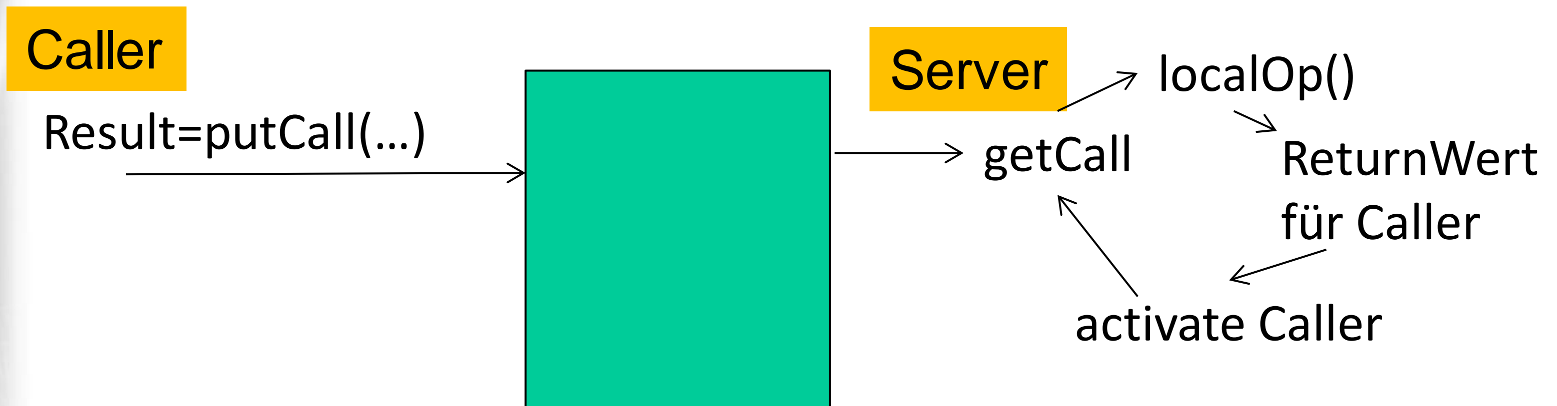
```
main() { // Consumer-Aktionen
   …
   Msg* m= ph->syn_get();
   …
```

```
m= ph->get();
m->sender->activate();
```

Producer und Consumer werden durch Nachrichtenpuffer **gekoppelt**

2. Aufgabe: Synchroner Remote-Procedure-Call

- Entwurf einer Klasse zur Realisierung eines synchronen Remote-Procedure-Calls



Lösung als weitere Spezialisierung von PortHead, PortTail, PortData

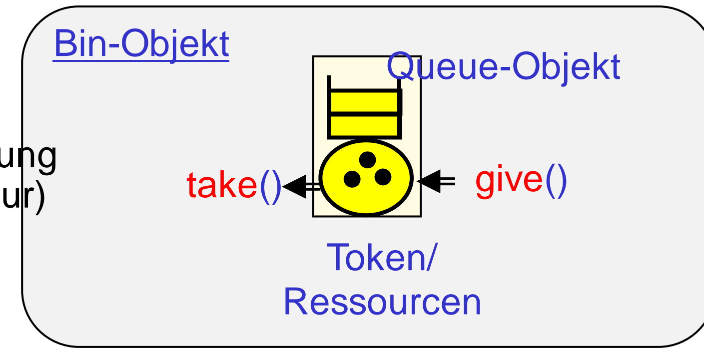
4. ODEMx-Modul Synchronisation: Bin, Res

- 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

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

- ein Prozess wird in seiner Anforderung (take()-Ruf) **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

  das Warten auf Token kann alternativ extern **unterbrochen** werden (Interrupt)

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
  FIFO-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)

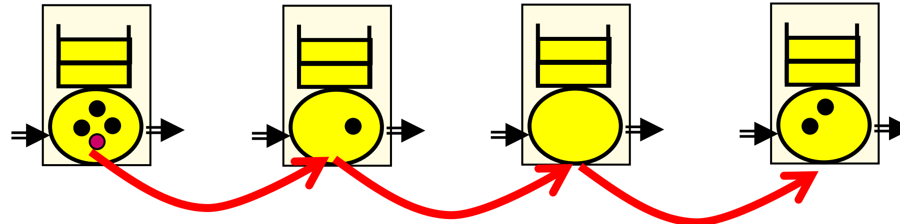
  P2 wäre mit der verfügbaren Tokenzahl von o zufrieden,
  darf P1 dennoch **nicht** überholen

Bin-Anwendungen

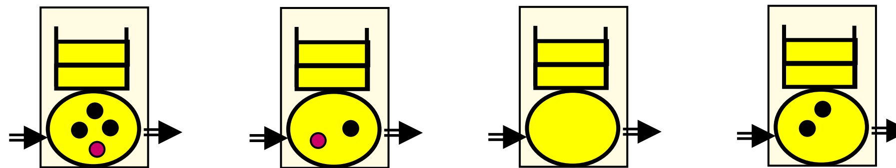
Token haben
bei einfachen Bins **keine** Identität

<u>Später:</u>
Bin-Templates

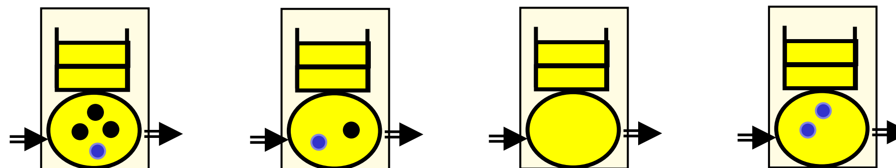
Basistyp
der Token
kann
als Parameter
vermittelt
werden



...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.
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

…

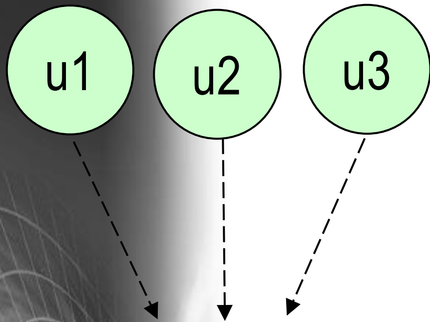
a) des **Wartevorgangs** auf verfügbare Ressourcen (Token)

b) des anwendungsspezifischen zeitlich begrenzten **Nutzungsvorgangs** der entnommenen Ressourcen (Token)

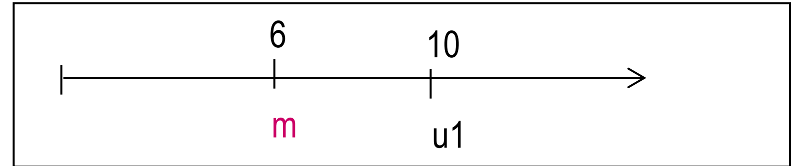
in beiden Fällen mittels interrupt !

```
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 (interupted() ) { …};
servicePoint-> give(3);          // Rückgabe von Token
```

Variante A: Unterbrechung des Wartevorgangs

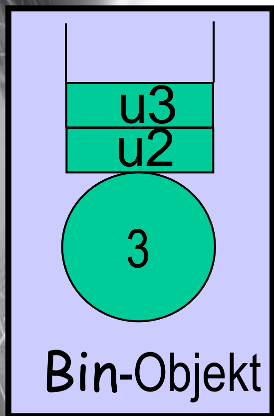


Nutzer,
die gleichzeitig
gestartet werden

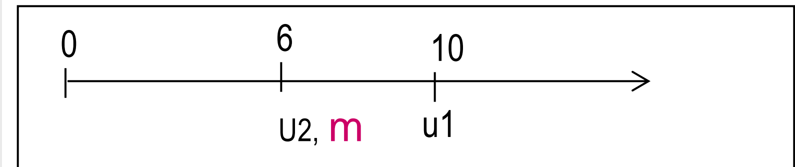


ein weiterer Prozess **m** zum Zeitpunkt 6.0

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

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




```
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 ….

   }
}
```

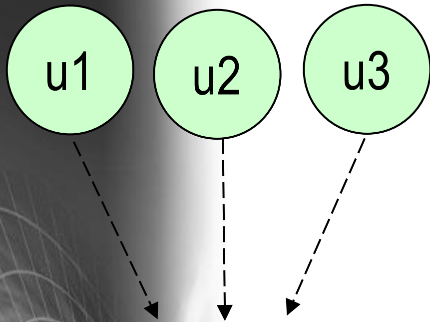


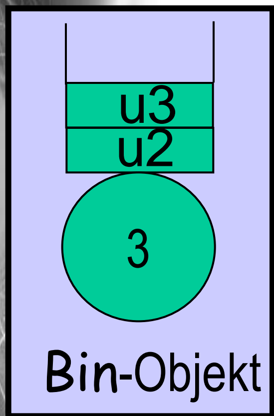
Unterbrechung

evtl. Nutzung eines anderen Bin-Objektes nach Warteabbruch

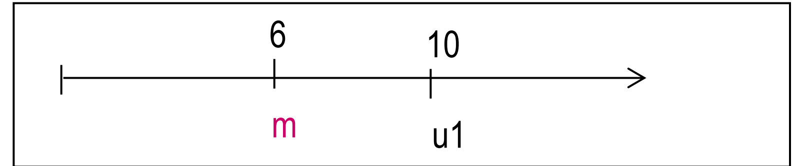
Variante B: Unterbrechung des Nutzungsvorgangs

Nutzer,
die gleichzeitig
gestartet werden





ein weiterer Prozess **m** zum Zeitpunkt 6.0

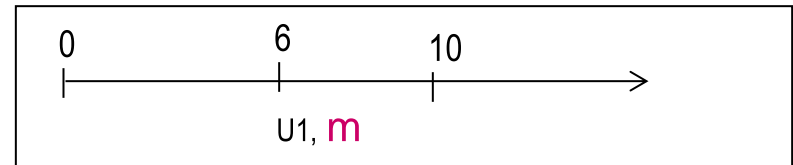


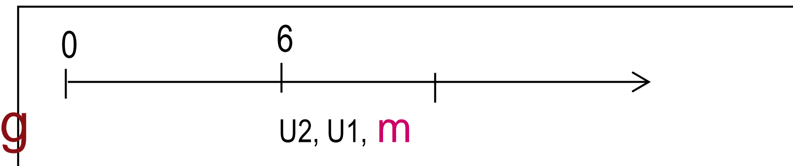

```
class Manager:
          public Process {
   ...

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

```
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





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

Muster
vom Geben- und Nehmen

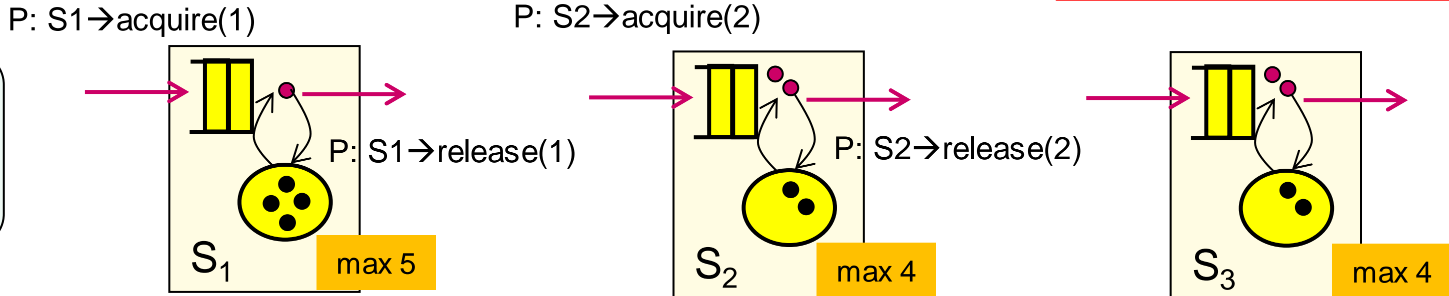
- `unsigned int release(n)`, n>0
  - Eingabe von Token
  - Token müssen nicht unbedingt diesem Res–Objekt vorab entnommen sein
  - wird maximale Tokenanzahl überschritten:
    Fehlermitteilung und Fortsetzung bei Ignorierung der überschüssigen Token

- **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

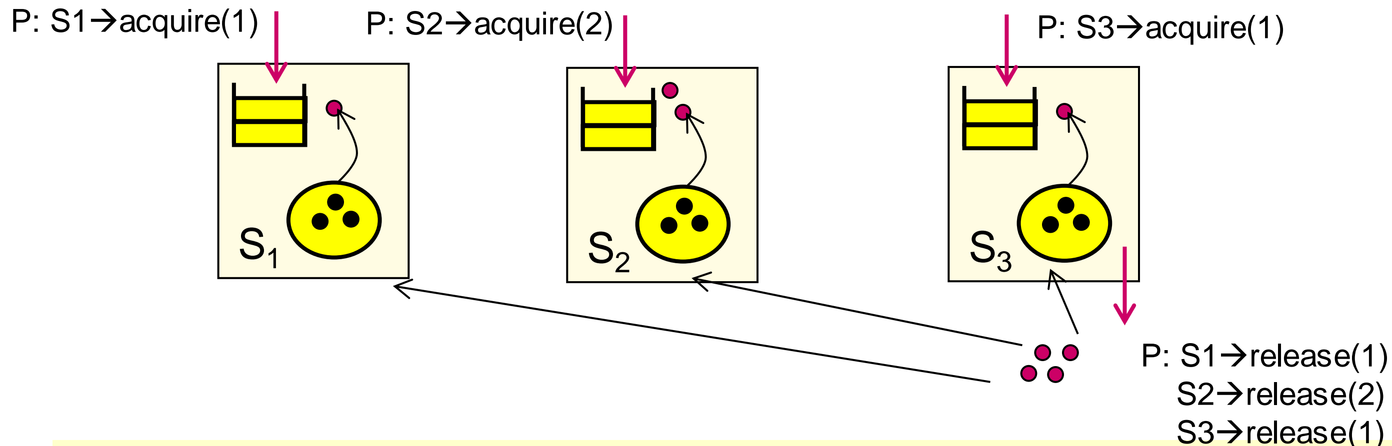
Res-Anwendungen

Token haben jedoch weder Identität noch Typ

- freie Ressource
- belegte -“-



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

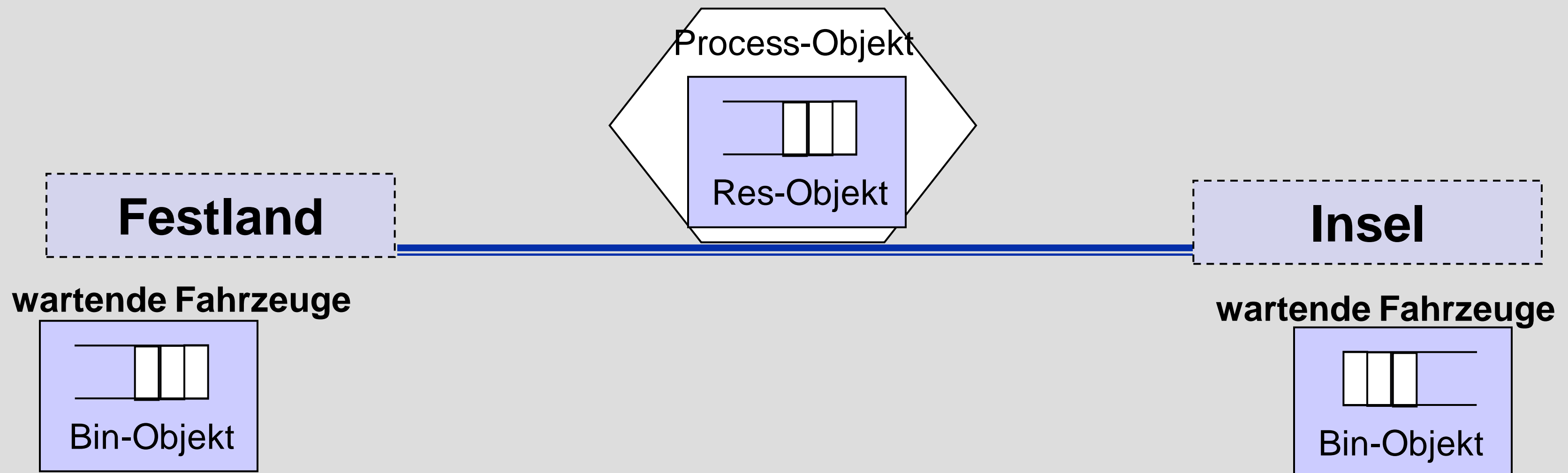
4. ODEMx-Modul Synchronisation:

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



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 ←→ Modell

reale Zufallsgrößen werden durch mathematische Modelle (Funktionen) approximiert:

- *Verteilungsfunktion, Dichtefunktion, statistische Kenngrößen*

2. Modell ← → Modell

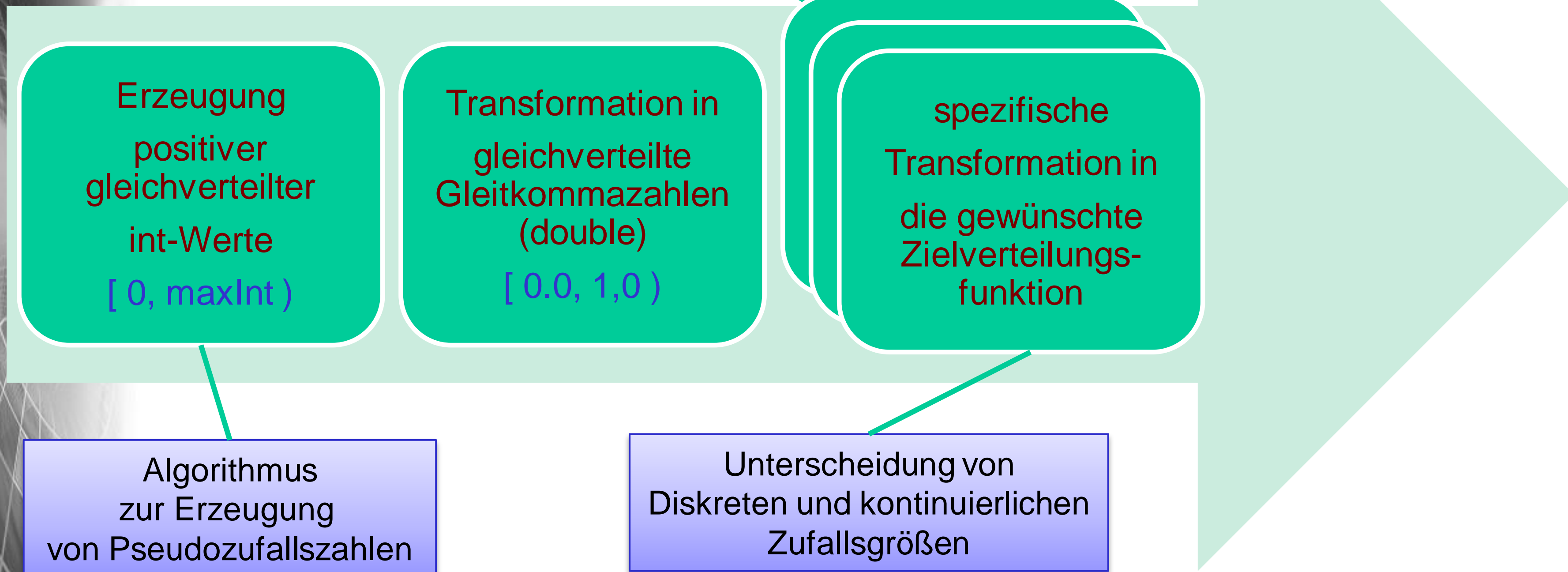
es bestehen mathematische Zusammenhänge zwischen einzelnen Verteilungsfunktionen:

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

3. Modell ← → Berechnungsmodell

determiniert berechnete Folgen von (0,1)-Werten lassen sich als Approximationen
von Verteilungsfunktionen realer Zufallsgrößen verwenden

Schema zur Berechnung von Zufallszahlen



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

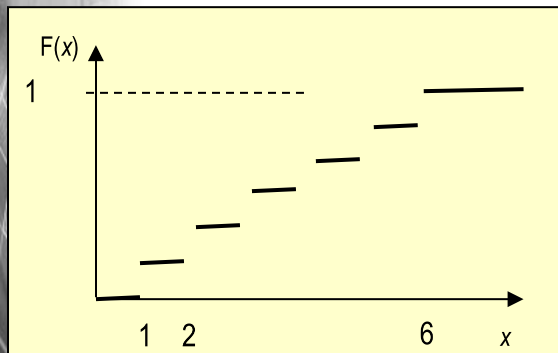
Charakterisierung einer Zufallsgröße X:

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

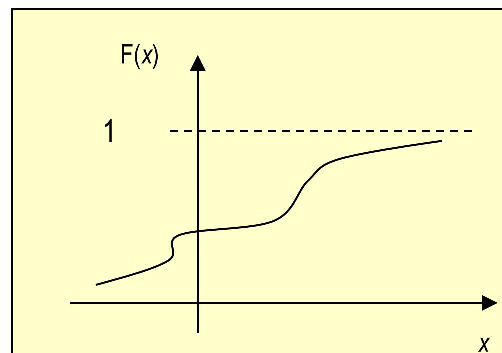
Verteilungsfunktion:= $F_X(x) = P(X \leq x)$,
der Wert von F_X (Verteilungsfunktion der Größe X) ist an der Stelle x ist **gleich der Wahrscheinlichkeit**, dass X einen Wert unterhalb von *x* annimmt.

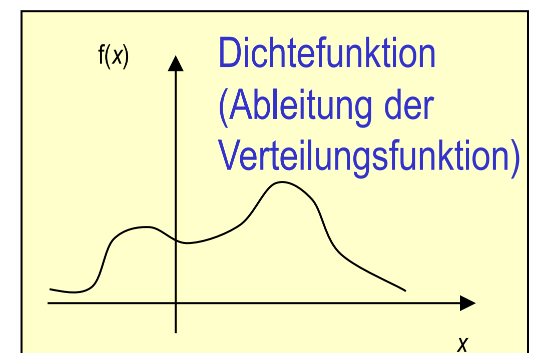
x durchläuft alle Werte der reellen Zahlengerade

diskret



kontinuierlich





Verteilungsfunktion als kumulative Dichtefunktion

Diskrete Zufallsgrößen

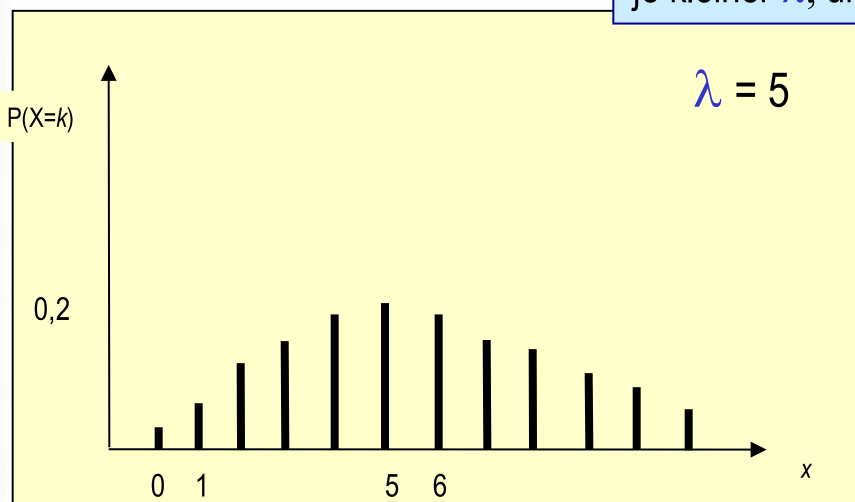
Charakterisierung einer diskreten Zufallsgröße X:

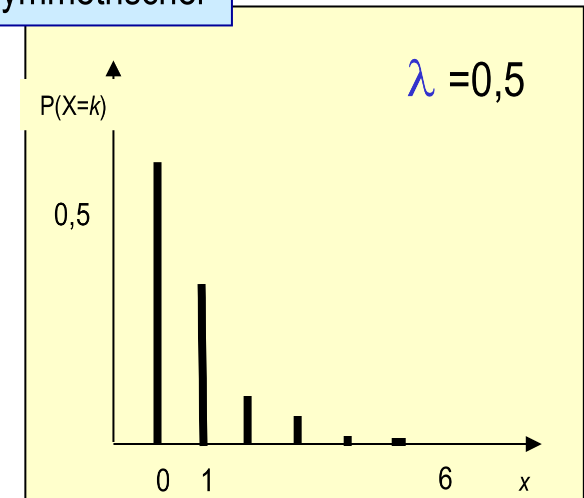
- X beschreibt Ereignisregistrierungen (Anzahl) in einem bestimmten Zeitbereich (X nimmt bei jedem Versuch zufällig einen best. Wert an)
- Werte genügen einem gleichen Typ von Verteilungsfunktion (**Poisson**)
- λ ist die mittlere Anzahl

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

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 $1/\lambda$

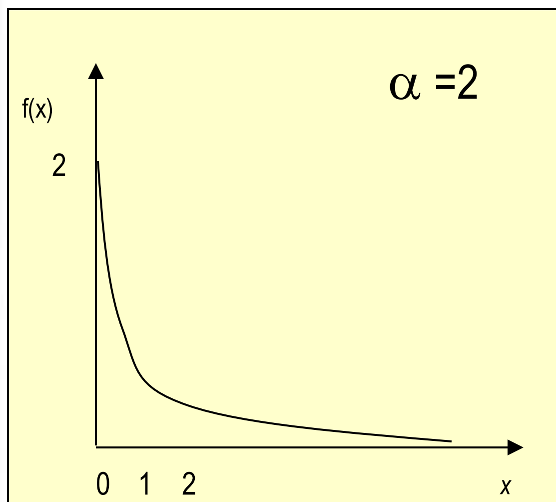
Stetige Zufallsgrößen

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>=0, sonst 0

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



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

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