Kurs OMSI
im WiSe 2012/13

Objektorientierte Simulation mit ODEMx

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

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

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

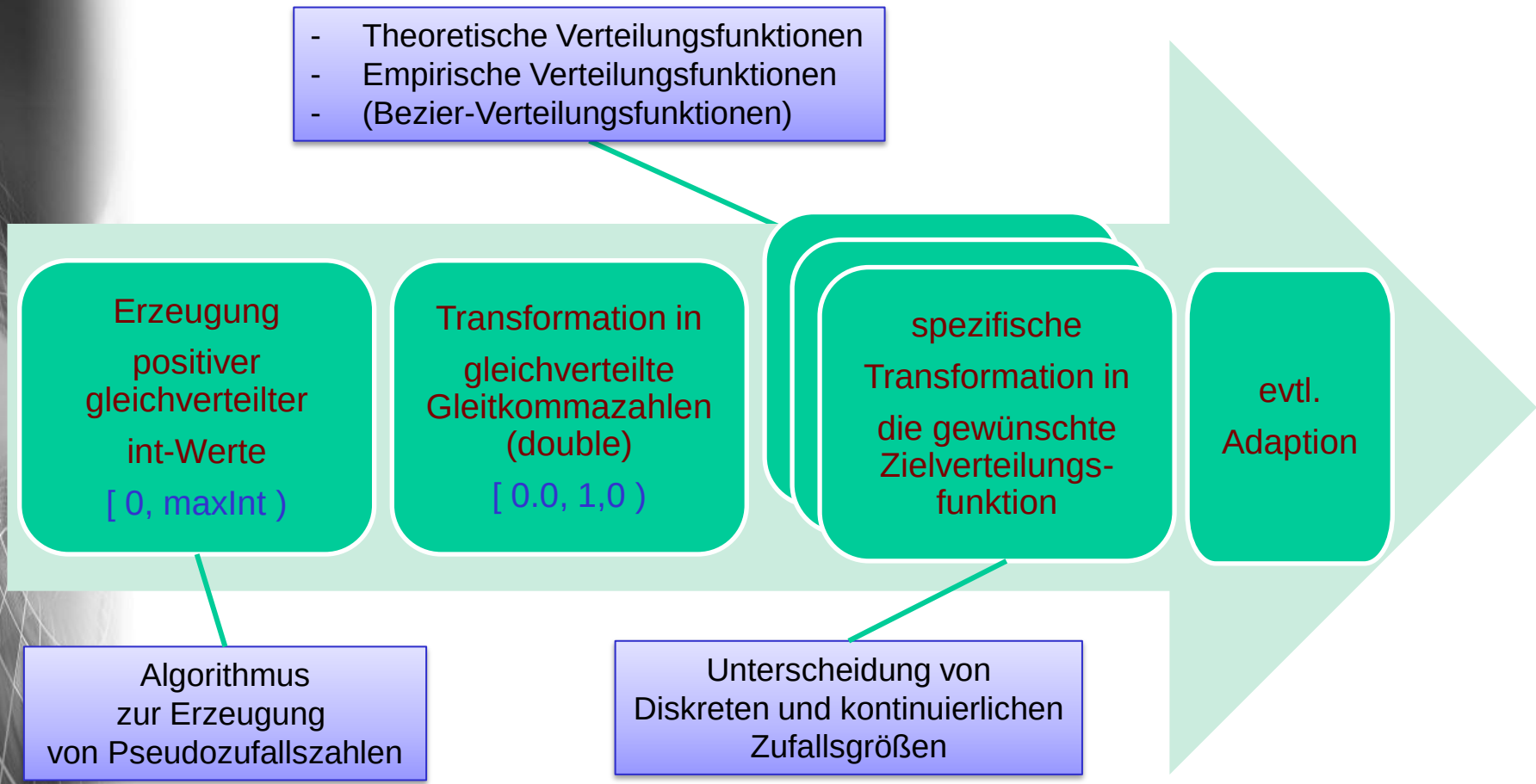
es bestehen mathematische Zusammenhänge zwischen einzelnen Verteilungsfunktionen:

- *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

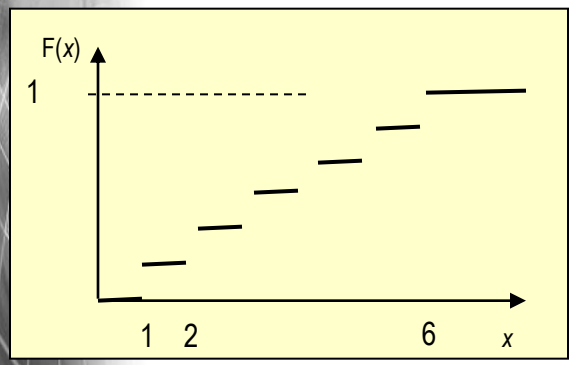
Charakterisierung einer Zufallsgröße X:

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

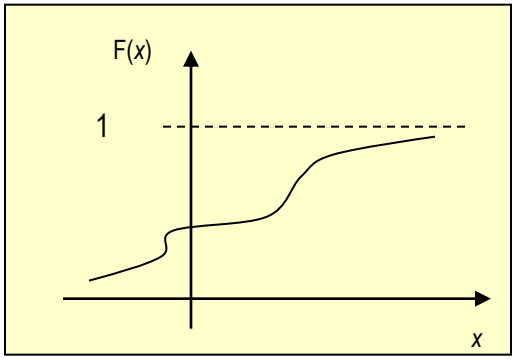
Verteilungsfunktion:= $F_X(x) = P(X \leq x)$,
der Wert von F_X (Verteilungsfunktion der Größe X) ist an der Stelle x ist **gleich der Wahrscheinlichkeit**, dass X einen Wert unterhalb von *x* annimmt.

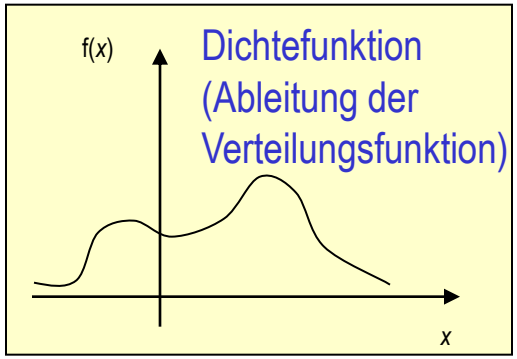
x durchläuft alle Werte der reellen Zahlengerade

diskret



kontinuierlich





Verteilungsfunktion als kumulative Dichtefunktion

Diskrete Zufallsgrößen

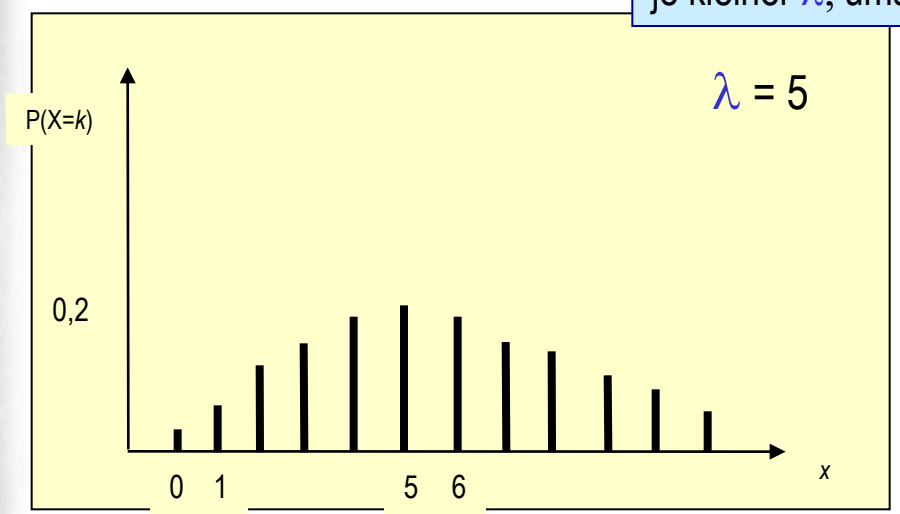
Charakterisierung einer diskreten Zufallsgröße X:

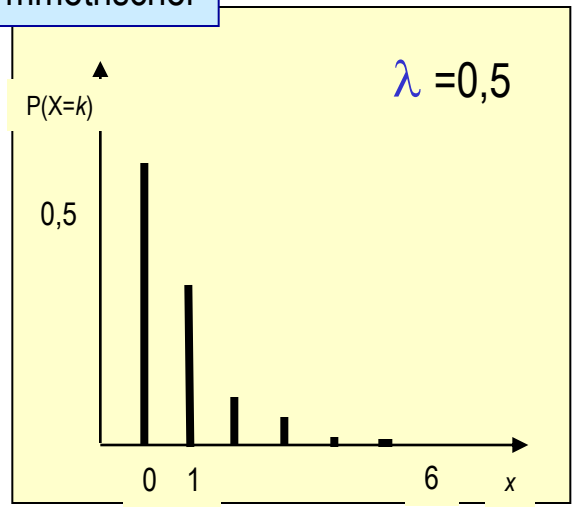
- X beschreibt Ereignisregistrierungen (Anzahl) in einem bestimmten Zeitbereich (X nimmt bei jedem Versuch zufällig einen best. Wert an)
- Werte genügen einem gleichen Typ von Verteilungsfunktion (**Poisson**)
- λ ist die mittlere Anzahl

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/λ

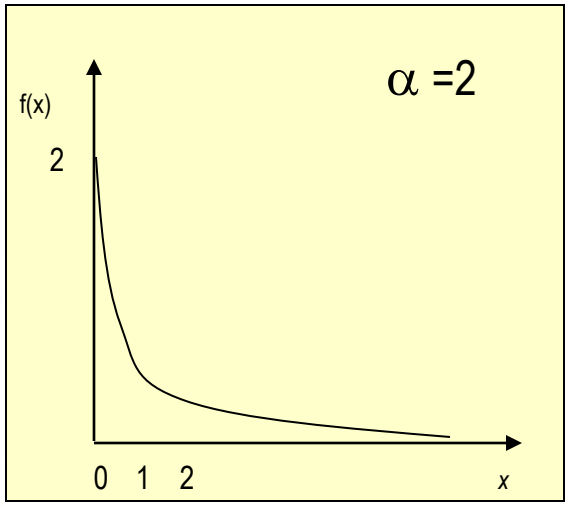
Stetige Zufallsgrößen

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, …*

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

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

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

➔ Zahlenfolge mit Periode p:
x_0= 4, 8, 5, 10, 9, 7, 3, 6, 1, 2, 4, … (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)

$x_{j+1} \equiv k\, x_j$ **mod** q

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

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

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

Algorithmus für Startwerte (unabhängiger) Generatoren

$u_0 = 907$ (erster Startwert eines Generators)

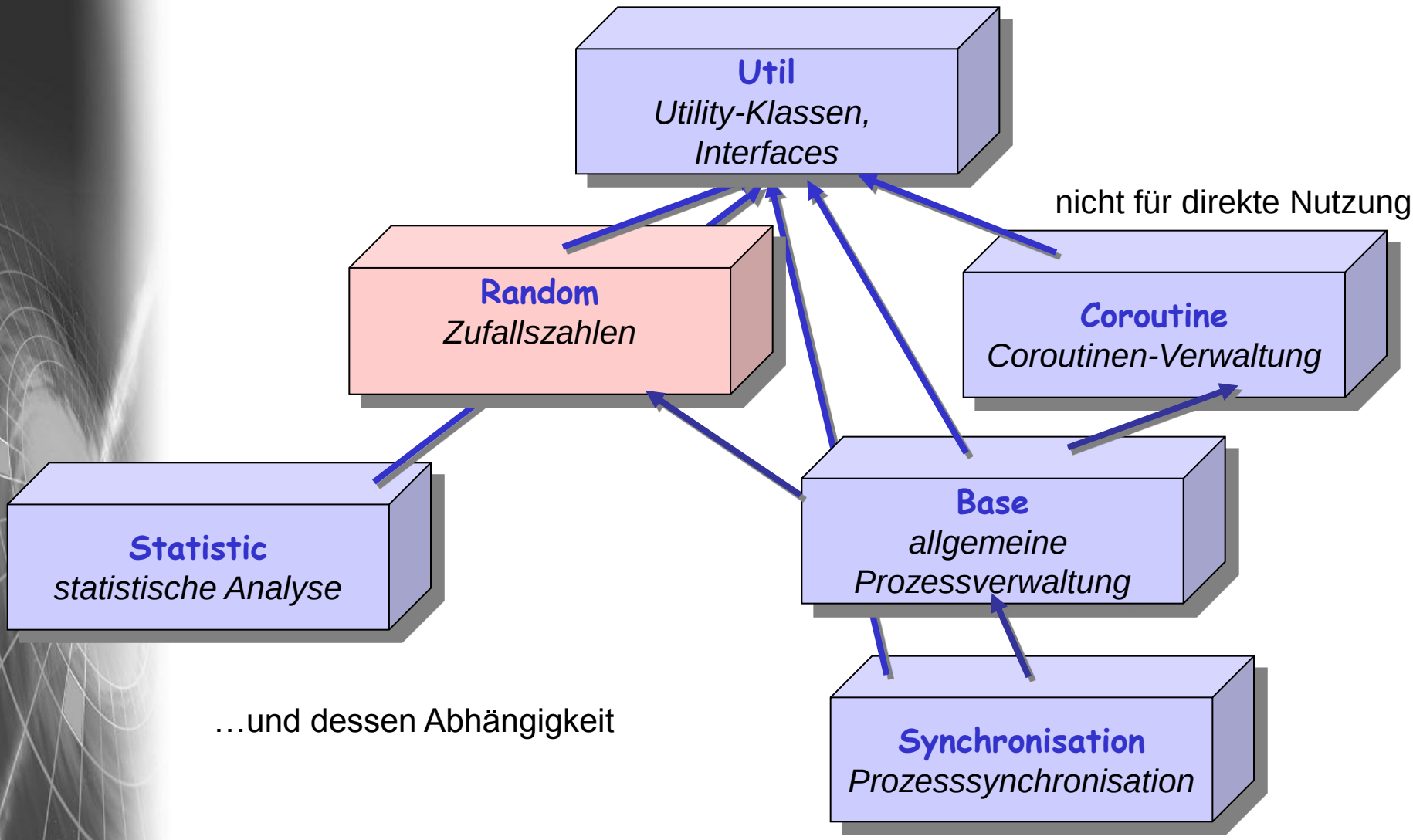
$u_{k+1} \equiv 36.855 * u_k$ **mod** $67.099.547$

➔ 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



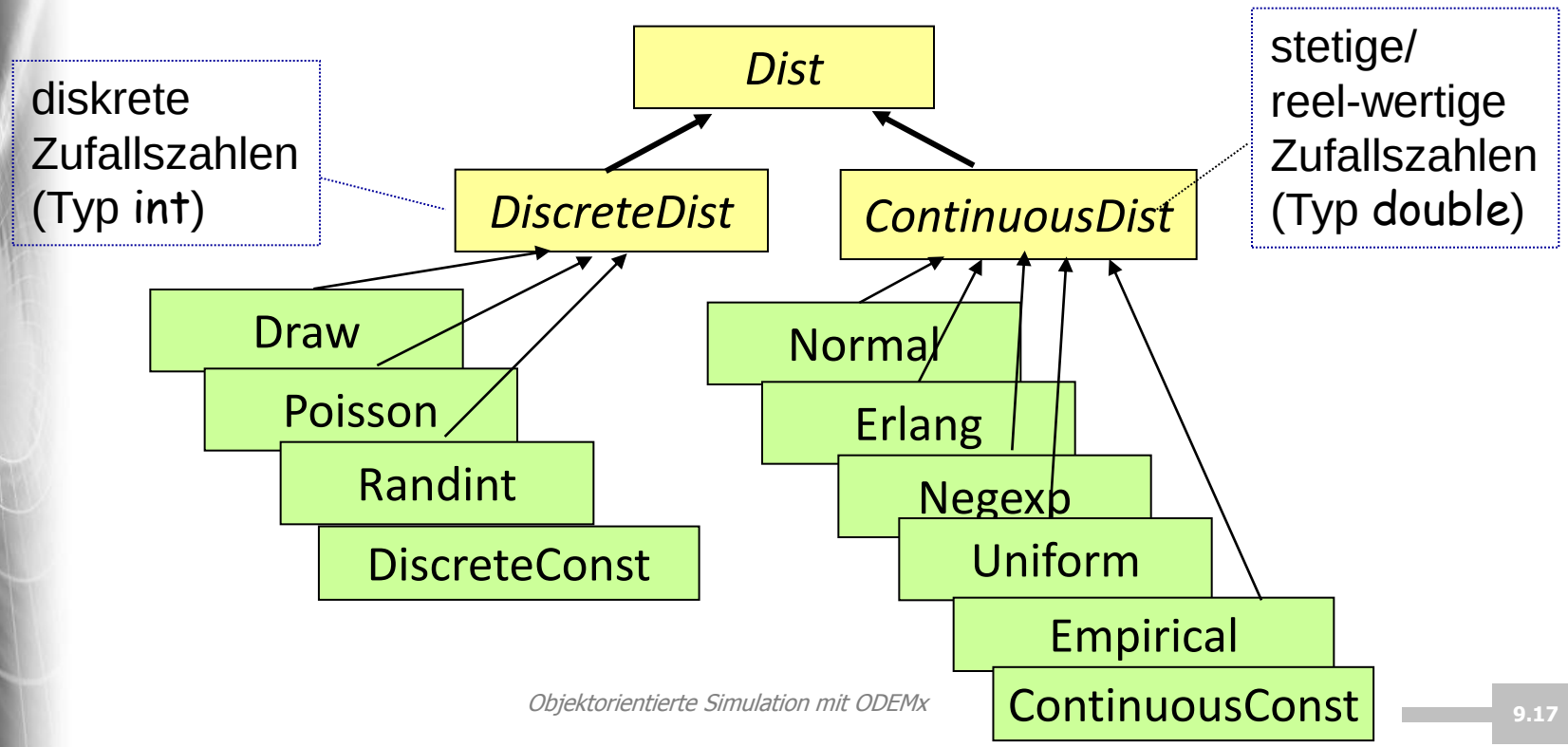
…und dessen Abhängigkeit

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



(0,1)- Pseudo-Zufallszahlen in ODEMx

Iterationsverfahren mit Startwert x_0

$$x_{j+1} \equiv k\, x_j \bmod q \quad (j = 0, 1, 2, \ldots)$$

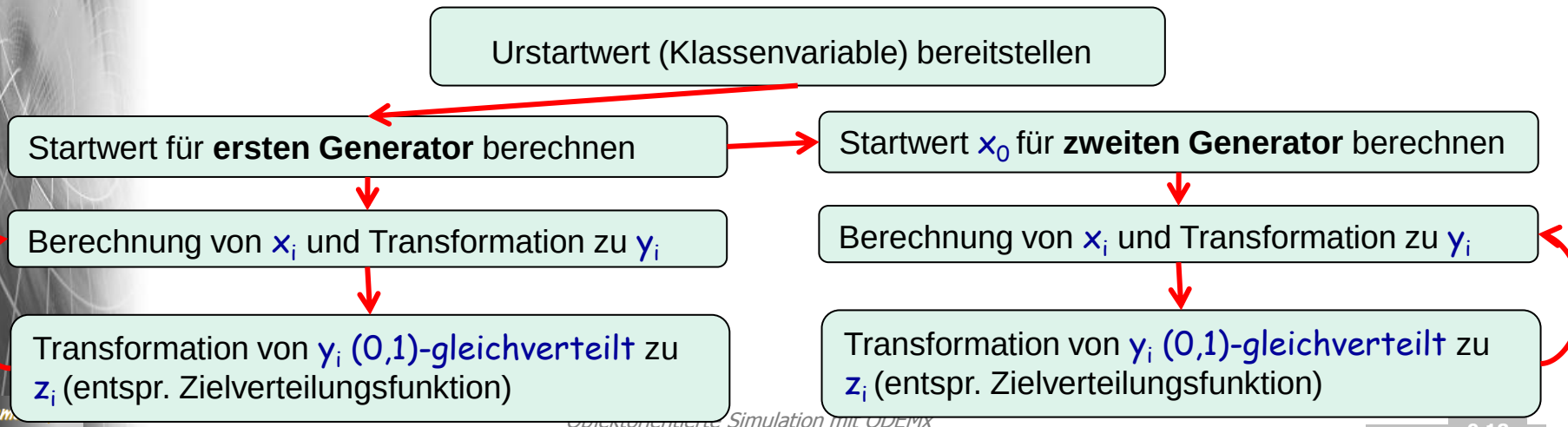
liefert Zahlenfolge mit Periode p:

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

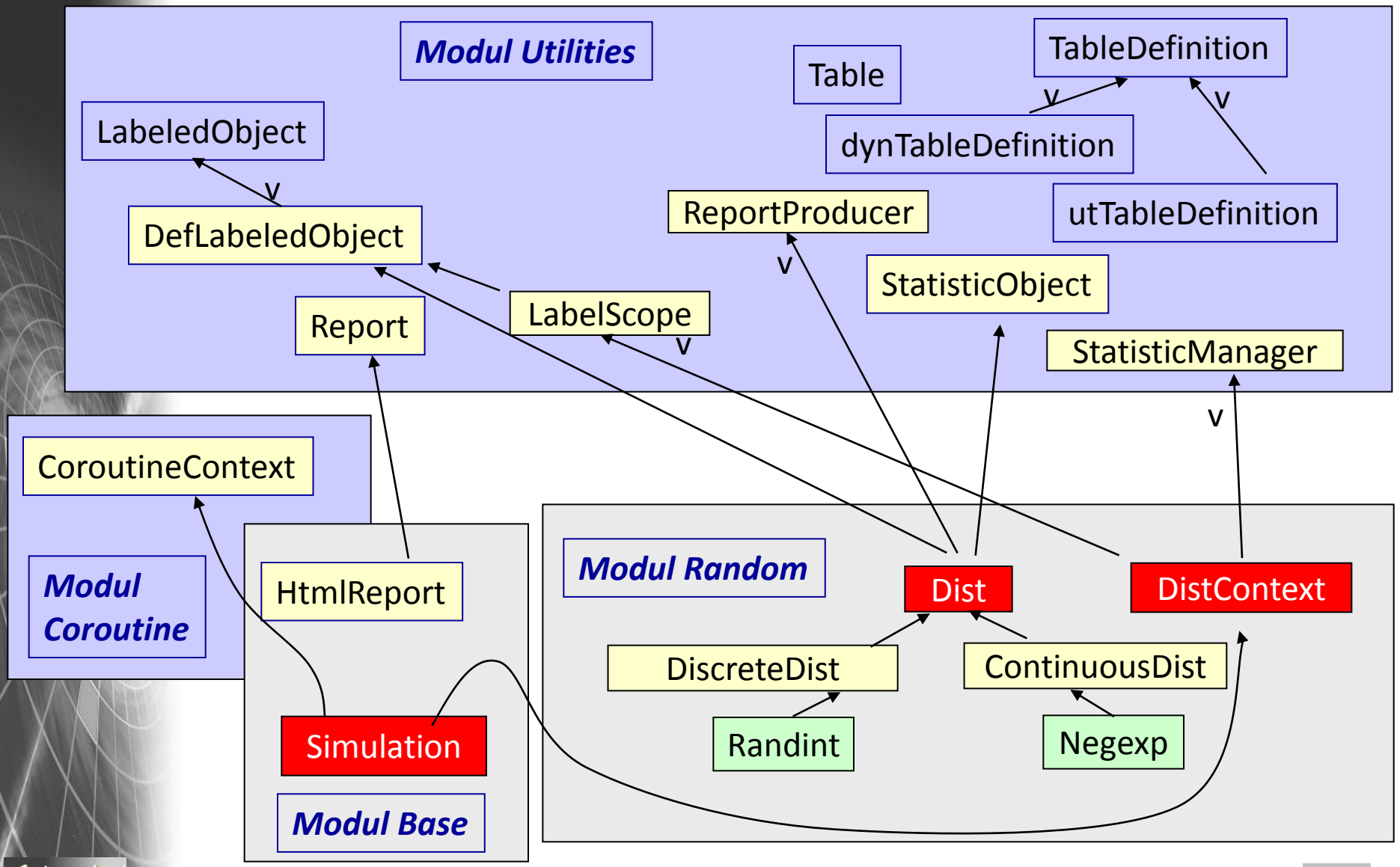
Generator für **gleichverteilte** (31 Bit-) Zufallswerte mit **maximaler Periode**

$q = 2^{31} - 1,$
$k = 7^5 = 16.807,$
$p = 2^{31} - 2 = 2.147.483.646$

Generator für **(0,1)-gleichverteilte Zufallswerte**
per Transformation : $y_i = x_i / q$



Generatoren in der ODEMx-Klassenhierarchie



Dist

Zufallszahlengeneratoren sind Objekte von Dist- Ableitungen

Dist ist nur für Erzeugung einer (0,1)-gleichverteilte Folge zuständig

```
class Dist : public DefLabeledObject, public StatisticObject,
                        public virtual ReportProducer    {
    protected:
        Dist(DistContext* c=0, Label label="");
        virtual ~Dist();
        ...

    public:
        virtual void setSeed( int n = 0);

    protected:
        double getSample(); //nächster (0,1)-Zufallswert
    private:
        DistContext* context;
        unsigned long u, ustart;
        unsigned int antithetics;
};
```

Woher bezieht ein Dist-Objekt seinen individuellen Startwert ?

i.d.R. von seinem (Simulations-)kontext über seinen Konstruktor!

oder nutzerspezifisch

u wird iterativ verändert

DistContext

- Ein DistContext-Objekt stellt einen gemeinsamen Kontext für verschiedene Zufallszahlengeneratoren dar

  (Simulation hat DistContext-Funktionalität geerbt)

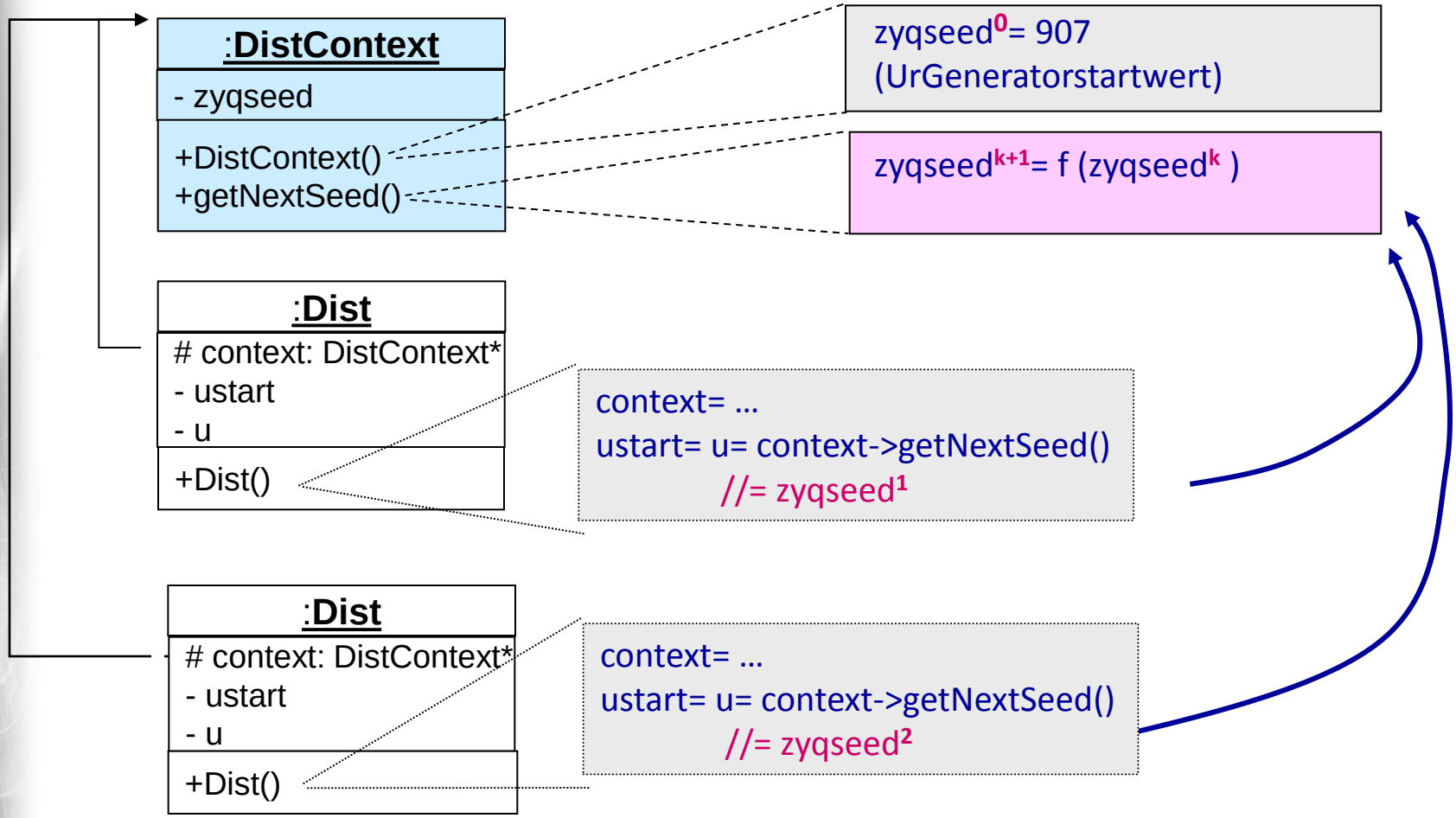
```
class DistContext : public virtual LabelScope,
                    public virtual StatisticManager {
    public:
        DistContext ();
        virtual ~DistContext();
        unsigned long getNextSeed();     // berechnet jeweils neuen
                                         // Startwert im Kontext
    protected:
        friend class Dist;
        getSeed();  //liefert aktuellen Startwert
    private:
        unsigned long zyqseed;    //aktueller Startwert
                                  //bzw. Urstartwert
};
```

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

Schema zur Bereitstellung von Startwerten

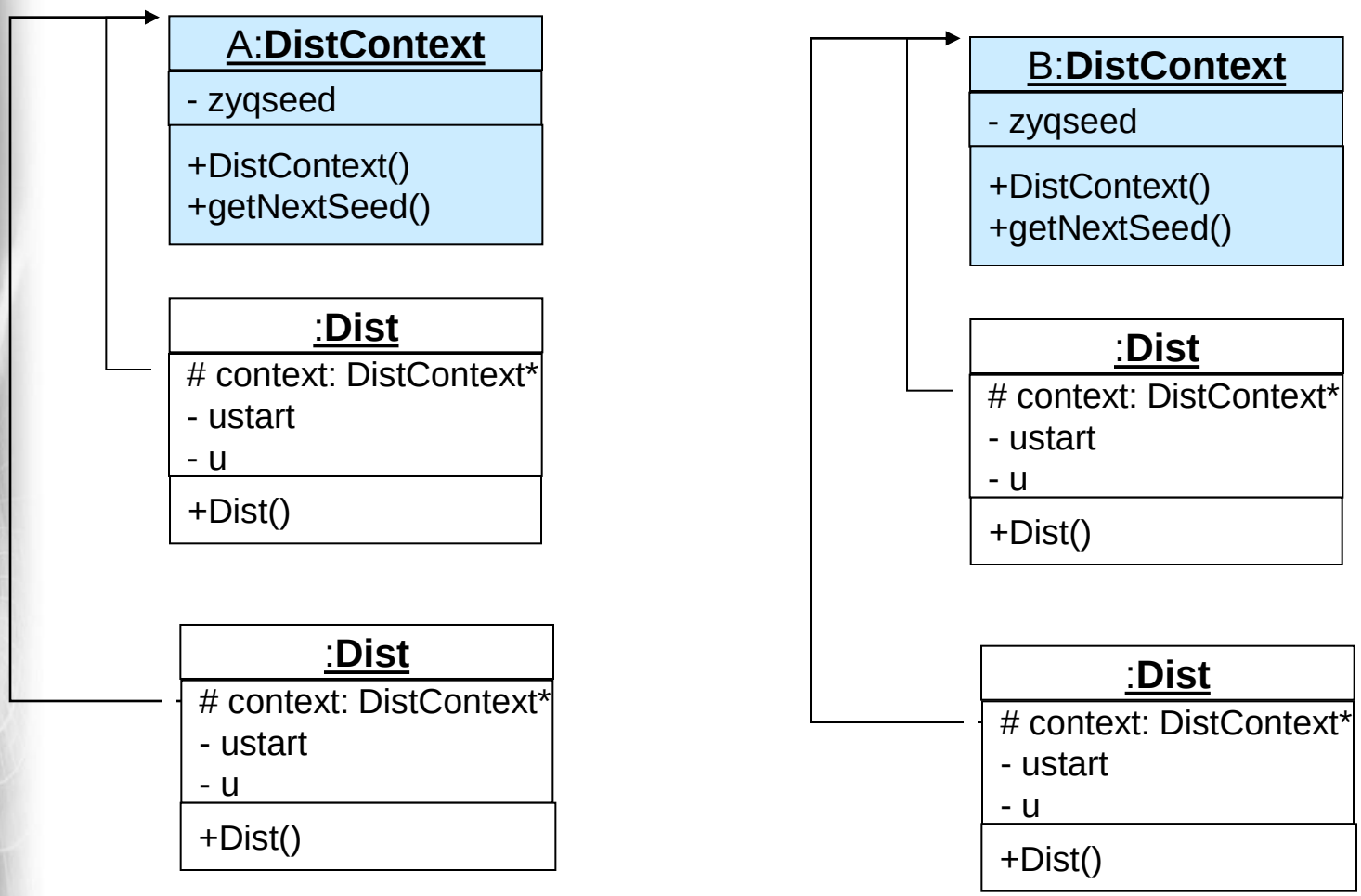
DistContext- Objekt mit zugeordneten Generator-Objekten



Zufallszahlengeneratoren sind Objekte von Dist-Ableitungen

Mehrere DistContext-Objekte

DistContext- Objekte A und B mit ihren Generator-Objekten



Eigene Kontextklassen statt Default-Simulation-Kontext

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

Report-File

Zeitpunkt der Reporterstellung

Bezug zum jeweiligen Simulations-kontext

alle erzeugten ZZ-Generatoren des zugehörigen Kontextes

Simulation: DefaultSimulation | **SimTime: 1318.53**

HtmlReport | **ODEMx Version: 1.0**

Random Number Generators

Name	Reset at	Type	Uses	Seed	Parameter 1	Parameter 2	Parameter 3
mainland	0	Negexp	95	33427485	0.1	0	0
island	0	Negexp	94	22276755	0.1	0	0
crossing	0	Normal	72	46847980	7.5	0.5	0

Queue Statistics

Name	Reset at	Min queue length	Max queue length	Now queue length	Avg queue length
mainland_queue	0	0	1	0	0
island_queue	0	0	1	0	0
ferry load_queue	0	0	1	0	0

Bin Statistics

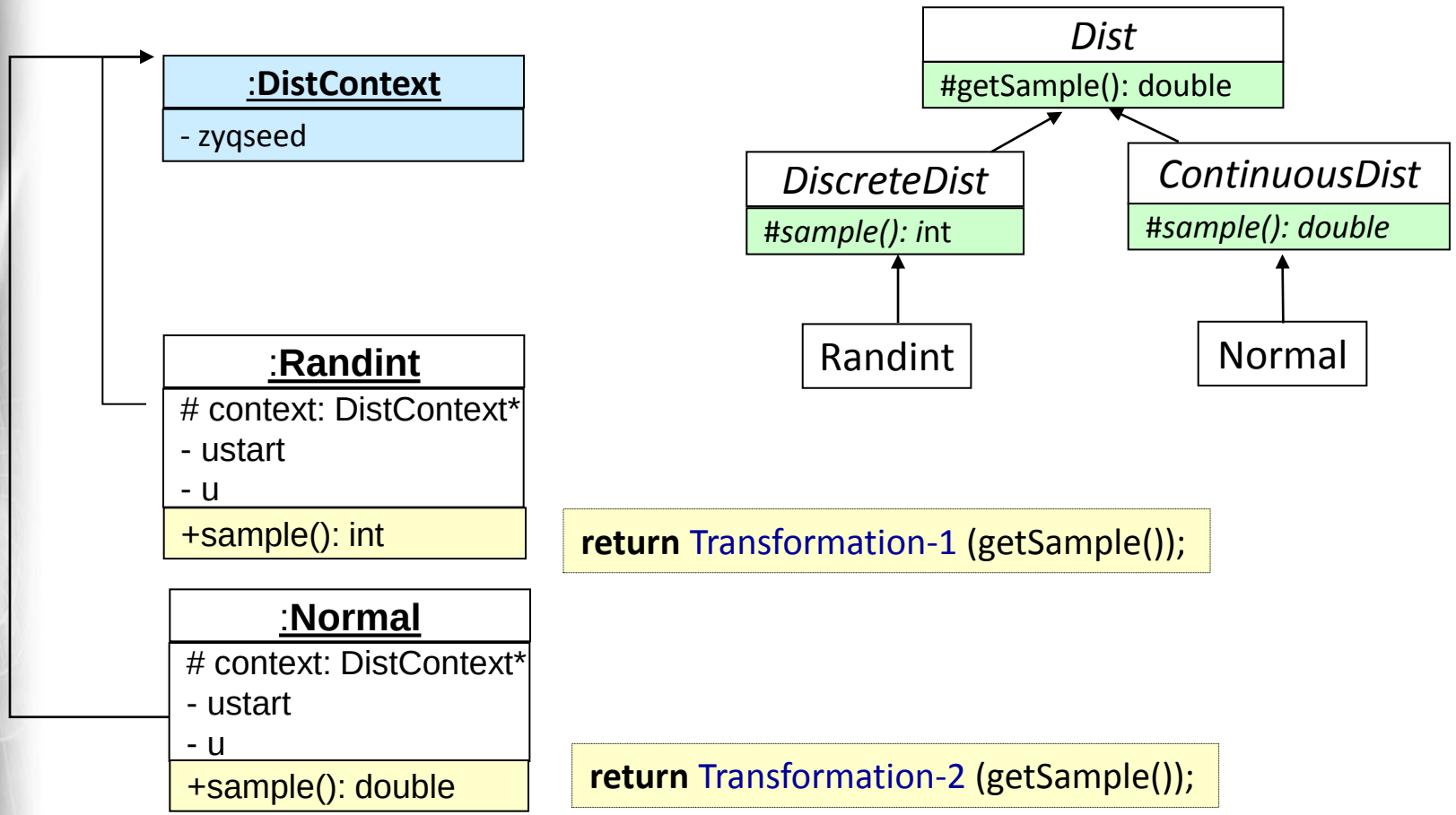
Name	Reset at	Queue	Users	Provider	Init number of token	Min number of token	Max number of token	Now number of token	Avg number of token	Avg waiting time
mainland_1	0	mainland_queue	92	94	3	0	7	5	1.99533	0
island_1	0	island_queue	93	93	1	0	8	1	1.4375	0

5. ODEMx-Modul Random

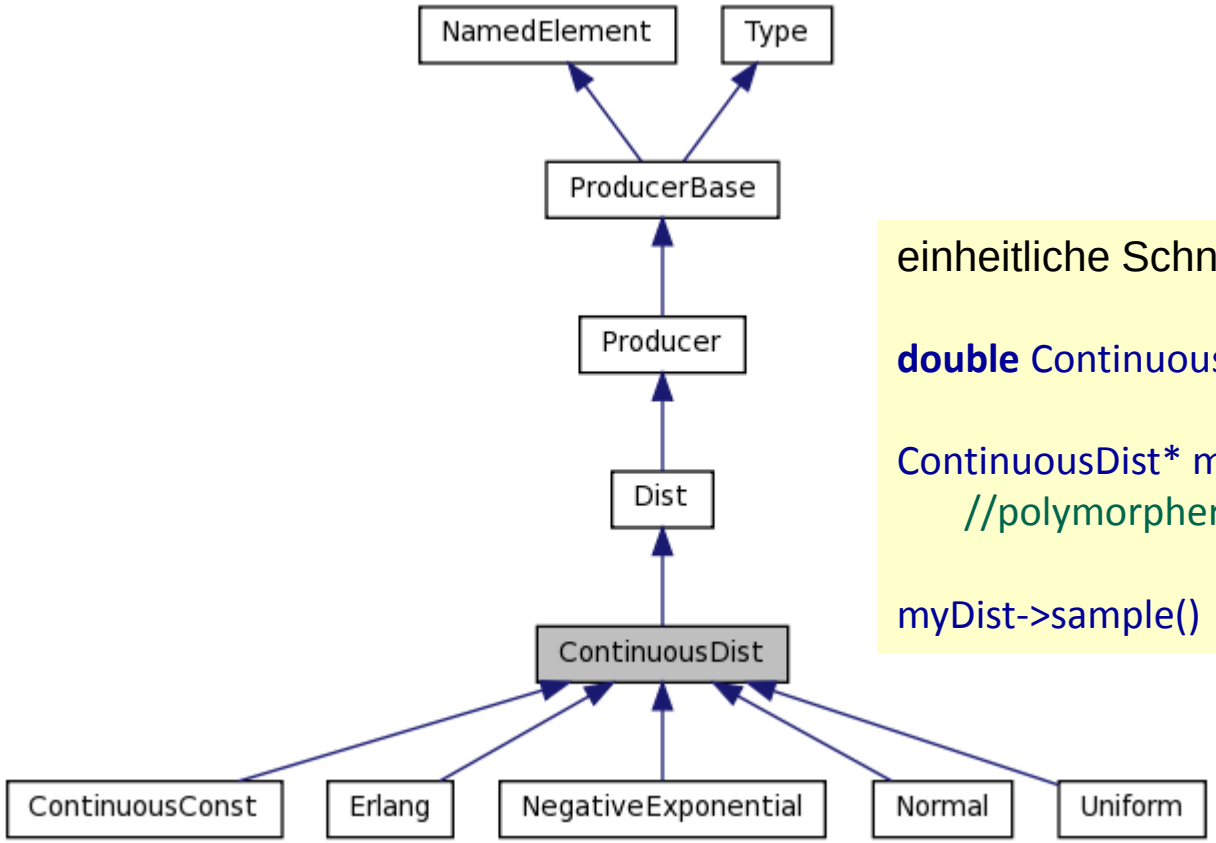
1. Charakterisierung von Zufallsgrößen
2. Approximation von Zufallszahlen
3. ODEMx- Zufallszahlengeneratoren (Übersicht)
4. Einstellung von Startwerten
5. Protokollierung
6. Berechnung von Zufallszahlen ausgewählter Verteilungen

Schema zur Berechnung von Zufallszahlen

redefinierte Member-Funktion sample() einer Dist-Ableitung
transformiert gleichverteilte (0,1)-Folge von Dist



Stetige Verteilungsfunktionen



einheitliche Schnittstelle

double ContinuousDist::sample()

```
ContinuousDist* myDist;
    //polymorpher Zeiger

myDist->sample()
```

Generator für exponentialverteilte Pseudo-Zufallszahlen

Transformationsgenerator für **exponential- verteilte Zufallswerte** mit Erwartungswert a

(Dichtefunktion: $f(x) = a\, e^{-ax}$)

$\{y_i\}$ sei (0, 1)- verteilte Zufallszahlenfolge von Dist

$x_i = (-1/a) * \ln (1 - y_i) \quad (i = 0, 1, 2, 3, \ldots)$

$x_i = (-1/a) * \ln (y_i) \quad (i = 0, 1, 2, 3, \ldots)$

➔ $x_0, x_1, x_2, \ldots, x_{p-1}, x_p = x_0, x_1, x_2, \ldots$
exponential-verteilte Zufallswerte mit Erwartungswert a

Generator für normalverteilte Pseudo-Zufallszahlen (2)

ODEMx-Lösung:

seien y_1 **und** y_2 zwei aufeinander folgende Werte einer (0, 1)-verteilten Zufallszahlenfolge

$$x_i = \sqrt{-2 \ln (y_i)} \sin(2\pi y_{i+1})$$

$$z_i = \mu + \sigma x_i$$

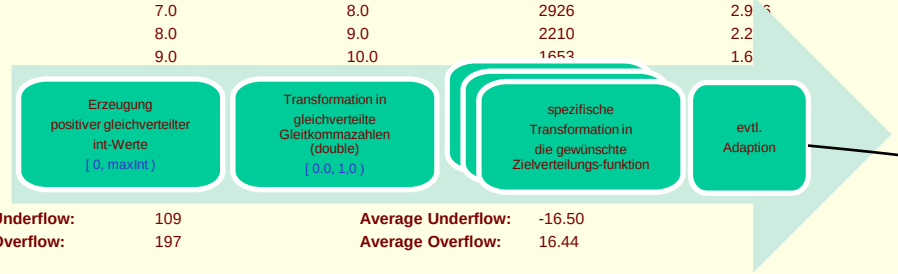
➔ $x_0, x_1, x_2, \ldots, x_{p-1}, x_p = x_0, x_1, x_2, \ldots$

normal-verteilte Zufallswerte
mit Erwartungswert $\mu = 0.0$ und
Standardabweichung $\sigma = 1.0$

normal-verteilte Zufallswerte
mit Erwartungswert μ
und
Standardabweichung σ

Random Variable	#Observed	Mean or ~Value	Std Dev or ~Error	Sig. Digits	Minimum	Maximum
norm_full	100000	0.483	5.014		-21.79	22.82

Lower	Upper	Frequency	Percent	
-15.0	-14.0	76	0.076	|
-14.0	-13.0	160	0.160	|
-13.0	-12.0	255	0.255	| *
-12.0	-11.0	469	0.469	| **
-11.0	-10.0	735	0.735	| ****
-10.0	-9.0	1099	1.099	| ******
-9.0	-8.0	1619	1.619	| **********
-8.0	-7.0	2275	2.275	| **********
-7.0	-6.0	3092	3.092	| **********
-6.0	-5.0	3874	3.874	| **********
-5.0	-4.0	4800	4.800	| **********
-4.0	-3.0	5829	5.829	| **********
-3.0	-2.0	6635	6.635	| **********
-2.0	-1.0	7313	7.313	| **********
-1.0	0.0	7748	7.748	| **********
0.0	1.0	8087	8.087	| **********
1.0	2.0	7802	7.802	| **********
2.0	3.0	7342	7.342	| **********
3.0	4.0	6552	6.552	| **********
4.0	5.0	5885	5.885	| **********
5.0	6.0	4759	4.759	| **********
6.0	7.0	3785	3.785	| **********
7.0	8.0	2926	2.9	| **********
8.0	9.0	2210	2.2	| **********
9.0	10.0	1653	1.6	| **********
				| ******
				| ****
				| **
				| *
				|



falls Zufallsgröße negativ, dann verwerfen und neuen Wert generieren solange bis positives Resultat

Underflow:	109	**Average Underflow:**	-16.50
Overflow:	197	**Average Overflow:**	16.44

Random Variable	#Observed	Mean or ~Value	Std Dev or ~Error	Sig. Digits	Minimum	Maximum
norm_half	100000	3.732	2.551		0.00	10.00

Lower	Upper	Frequency	Percent	
0.0	1.0	15859	15.859	| **********
1.0	2.0	15387	15.387	| **********
2.0	3.0	14290	14.290	| **********
3.0	4.0	12936	12.936	| **********
4.0	5.0	11246	11.246	| **********
5.0	6.0	9397	9.397	| **********
6.0	7.0	7514	7.514	| **********
7.0	8.0	5805	5.805	| **********
8.0	9.0	4389	4.389	| **********
9.0	10.0	3177	3.177	| **********

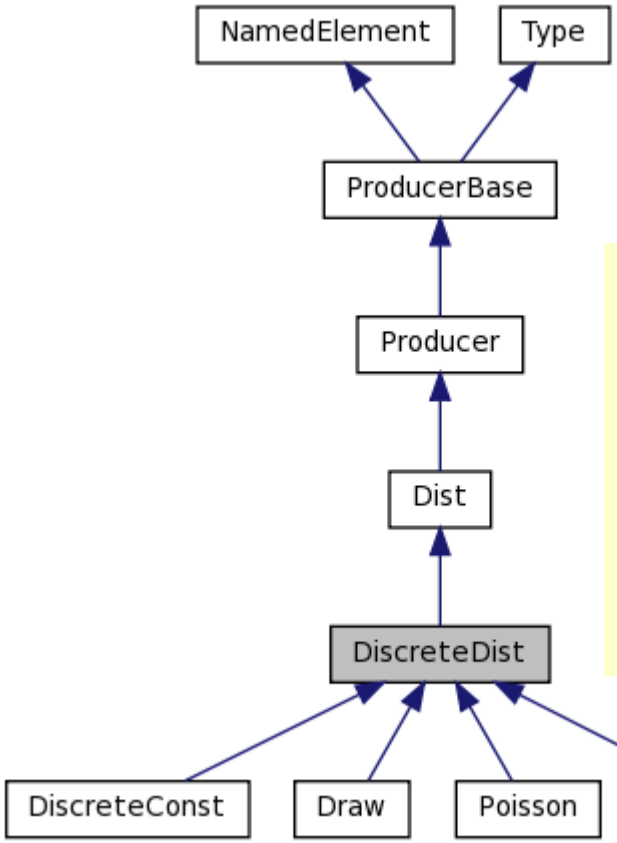
Generator für gleichverteilte reelle Pseudo-Zufallszahlen

Transformationsgenerator für **gleich-verteilte stetige Zufallswerte**
aus dem Intervall [a, b)

$\{y_i\}$ sei (0, 1)- verteilte Zufallszahlenfolge
(erzeugt durch bekannten Generator)

$$x_i = a + (b-a) * y_i$$

Diskrete Zufallszahlengeneratoren



einheitliche Schnittstelle

```
int DiscreteDist::sample()

DiscreteDist* myDist;
      //polymorpher Zeiger

myDist->sample()
```

Generator *für gleichverteilte diskrete Pseudo-Zufallszahlen*

- Konstruktor

```
Randint::Randint(DistContext* c, Label title, int na, nb);
// trägt Objekt in Liste des Kontextes ein
```

rechter Rand

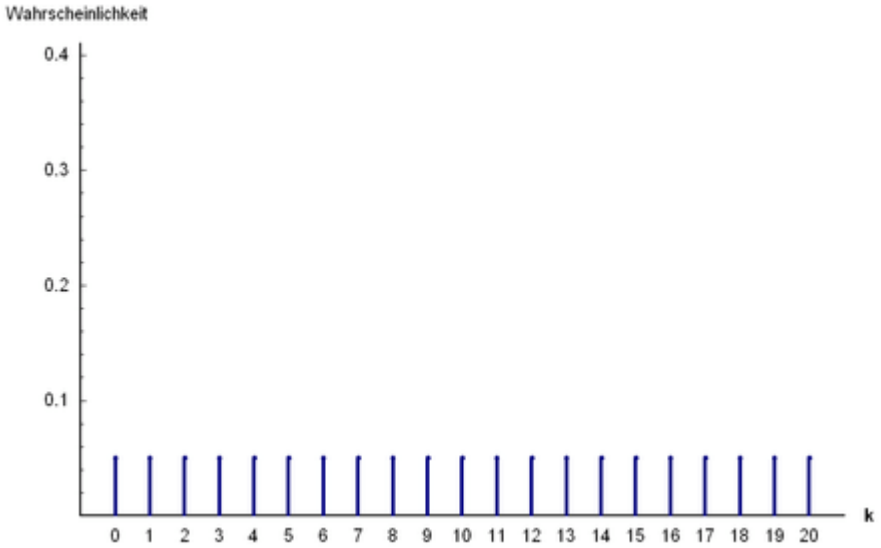
linker Rand

- Generator-Objekt

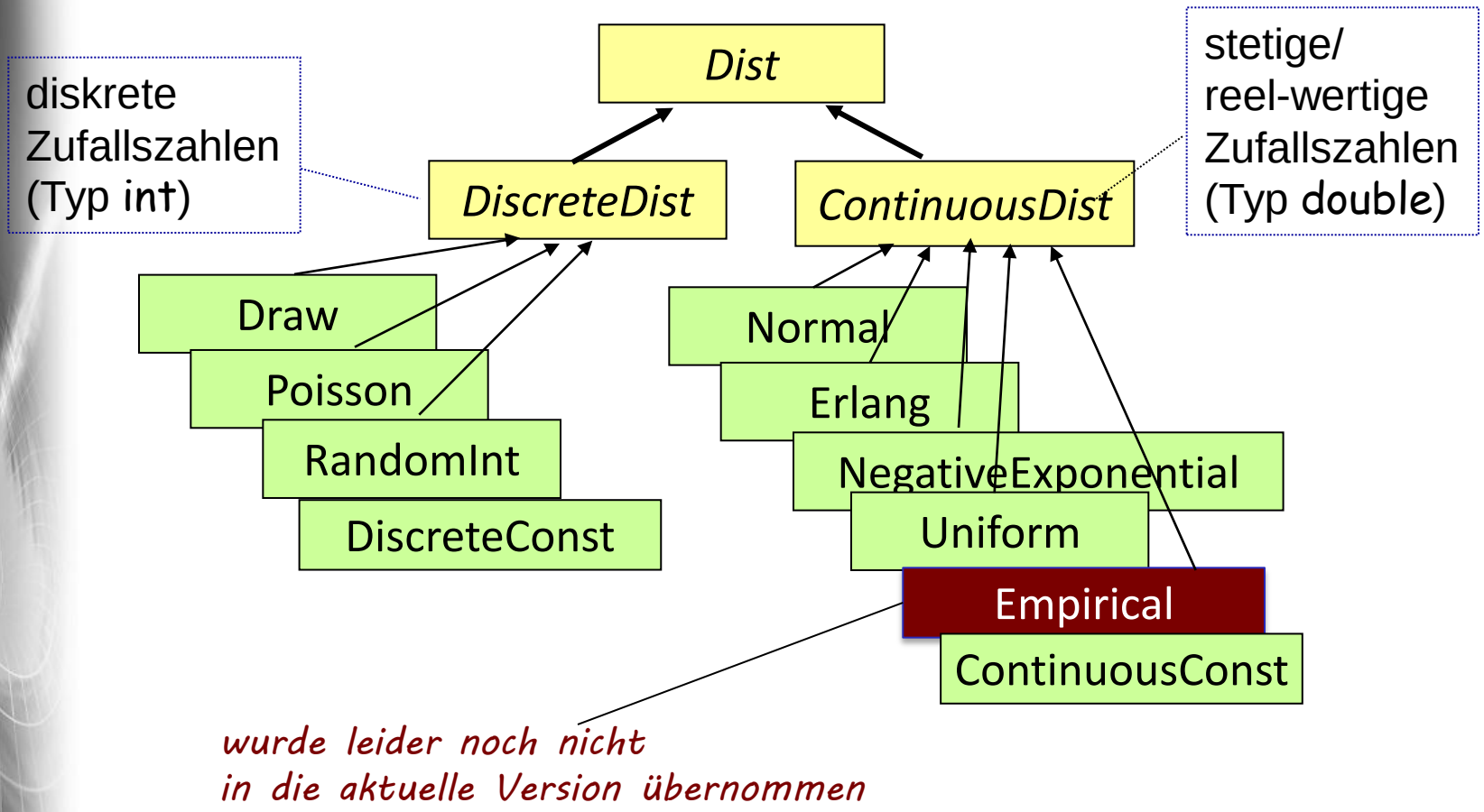
```
Dist* wuerfel; Simulation* sim;
wuerfel= new RandomInt (sim,“Los", 0, 20);
```

- Anwendung

```
wuerfel->sample();
```



Alle ZZ-Generatoren von ODEMx

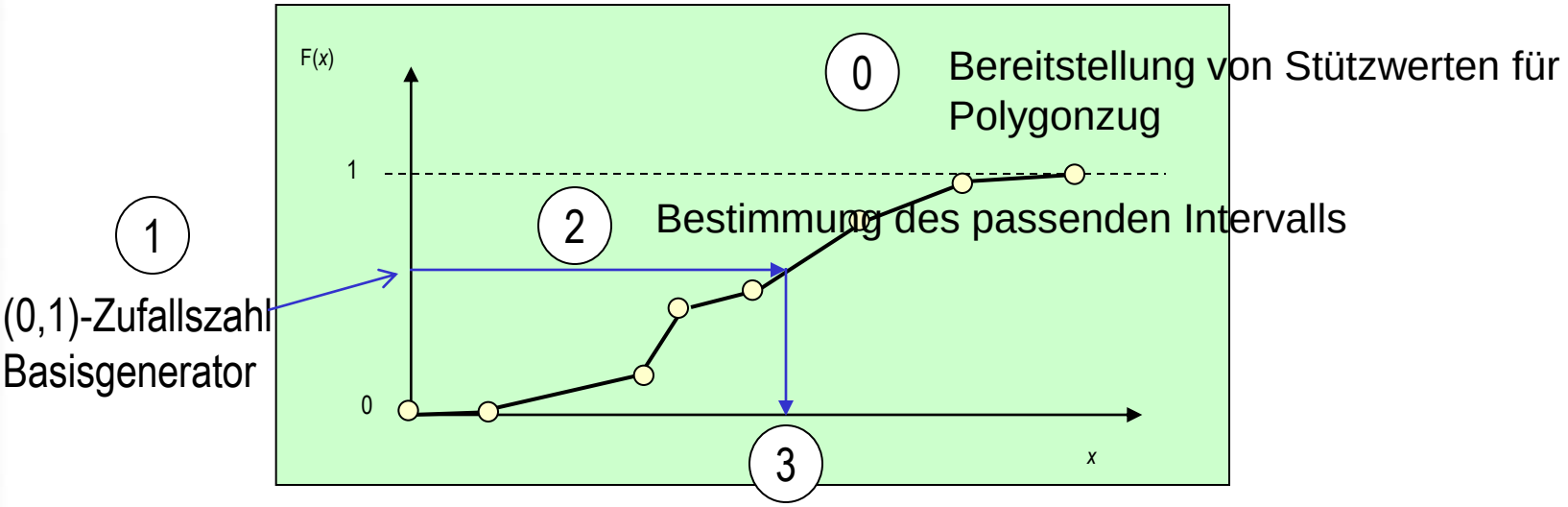


Generator für empirischverteilte Pseudo-Zufallszahlen

Vor.: aufgezeichnetes Histogramm einer beobachteten Größe
Häufigkeit für Werte in Werteklassen, daraus: kumulative Häufigkeit F(x)

→ Polygonzug über (x,F(x))-Stützwerte als Verteilungsfunktion:

Methode zur Ermittlung einer Zufallszahl entsprechend einer empirischen Verteilung F(x): Schritte 0 bis 3

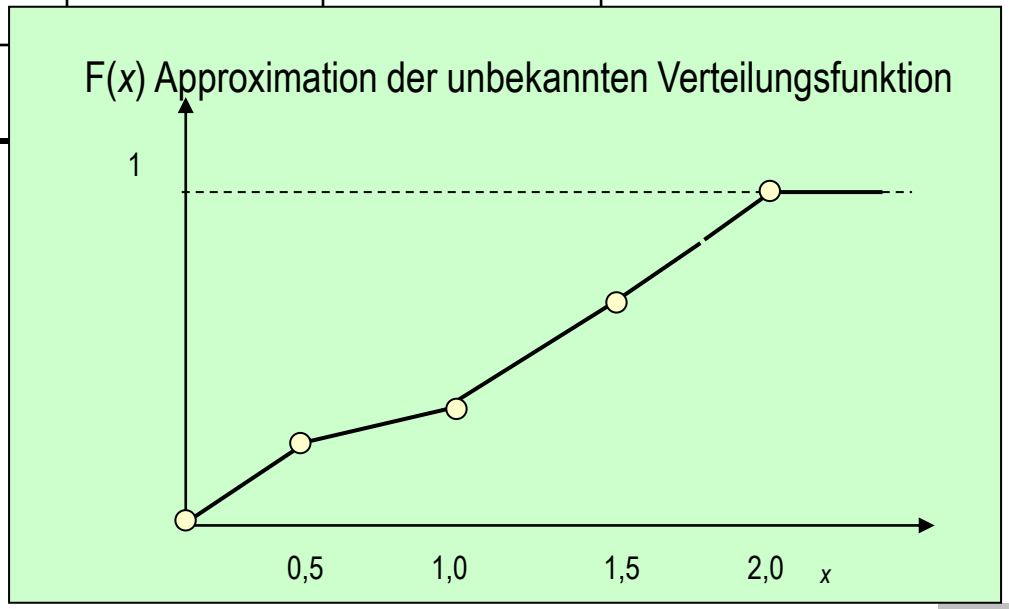


Bestimmung des x-Wertes (Zufallsgröße, die der empirischen F(x) genügt)

Empirische Verteilungen

Beispiel: Aufzeichnung von 100 Reparaturzeiten x

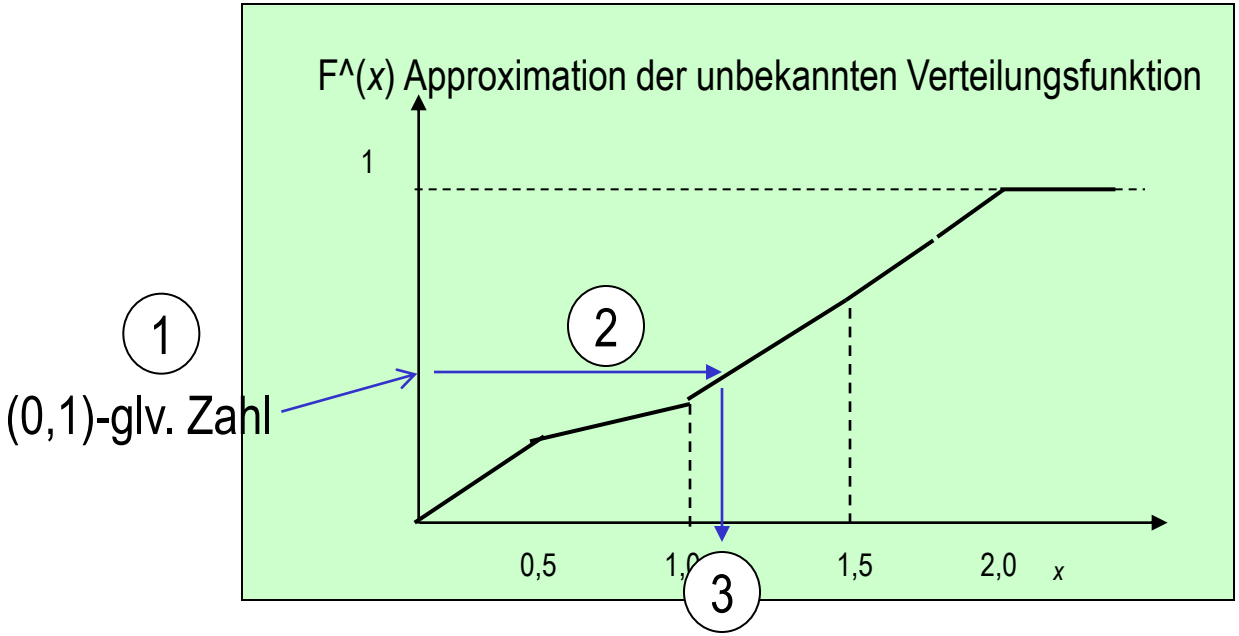
Intervall(h)	Häufigkeit	relative Häufigkeit	kummulative Häufigkeit
0 <=x<=0.5	31	0.31	0.31
0.5 < x<=1.0	10	0.10	0.41
1.0 < x<=1.5	25	0.25	0.66
1.5 < x<=2.0	34		



Empirische Verteilungen (Forts.)

Vorgehensweise:

- Erzeugung einer (0,1)- verteilten Zufallszahl
- Bestimmung des passenden Intervalls (Funktionsgleichung)
- Bestimmung der Zufallsgröße (Reparaturzeit)



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

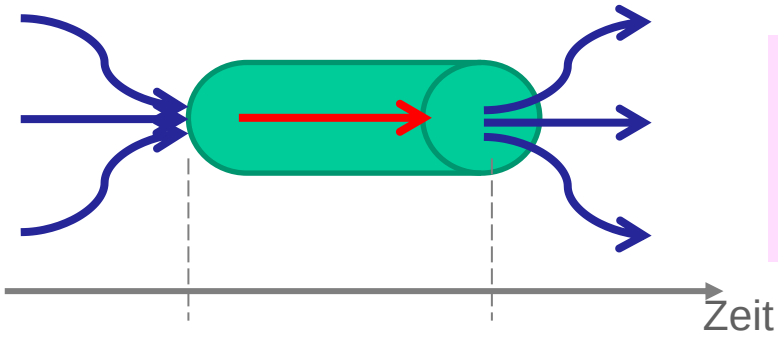
6. ODEMx-Modul Synchronisation:

WaitQ, CondQ

- Konzept WaitQ
  - Beispiel: Tankerflotte, Hafen, Raffinerie
- Konzept CondQ
  - Beispiel: Hafen, Schlepper, Gezeiten
- Weitere Anwendungsbeispiele für WaitQ u. CondQ
- Zusammenfassung/einheitliche Betrachtung

Nützliches Modellierungsmuster

- n+1 (>=2) Prozesse kooperieren ab einem Zeitpunkt für eine gewisse Dauer
- **Bed**.: (1) Zum Startzeitpunkt der Kooperation sind alle n+1 Prozesse verfügbar/für die Kooperation bereit
  (2) Zustandsänderungen der Prozesse sind voneinander abhängig)



Entschärfung der Parallelität
von synchronen Wechselwirkungen
bei Zustandsänderungen
im Simulator

- Effiziente simulative Umsetzung auf einer Ein-Prozessor-Maschine
  - einer der n+1 Prozesse übernimmt als **Master** (aktiv) die Ausführung der Zustandsänderungen sämtlicher Prozesse in Abhängigkeit der Modellzeit
  - alle anderen n Prozesse warten als **Slave** (passiv) auf die Beendigung der Kooperation durch den **Master**

ACHTUNG: Master und Slave sind Rollen, die Prozesse zeitweilig spielen

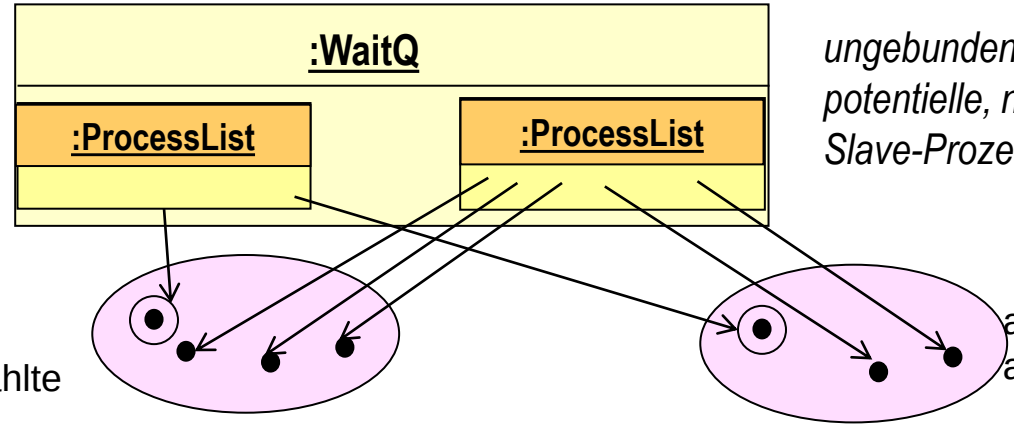
WaitQ-Konzept

Synchronisationsklasse

zur Erfassung von Prozessen und Bildung zeitweiliger Kooperationsgemeinschaften mit unterbrechbarem Warten auf das Zustandekommen der Kooperation, falls Kooperationspartner momentan nicht zur Verfügung stehen

- jeweils **einem** Master lassen sich beliebig **viele** Slave-Prozesse zuordnen
- **Master** bestimmt **allein** die Dauer der Kooperationsleistung (und gibt danach die Slaves, i. allg. gleichzeitig, wieder frei)
- **Master** realisiert **allein** die entsprechenden Zustandsänderungen, die mit der Kooperation aller Partner verbunden sind (benötigt entsprechende Zugriffsrechte auf seine Slaves)

ungebundene potentielle, noch blockierte Master-Prozesse



ungebundene potentielle, noch blockierte Slave-Prozesse

aktiver Master verwaltet temporär ausgewählte Slaves

anderer aktiver Master, andere Slaves

Weitere Anforderungen an WaitQ

1. über ein WaitQ-Objekt sollen sich

   gleichzeitig / nacheinander

   beliebig viele temporäre Master-Slave-Ensemble bilden können

2. folgende Teilaktivitäten bei Nutzung eines WaitQ-Objektes sollen extern (z.B. Timer) vorzeitig **unterbrechbar** sein:
   - Warten eines Prozesses als Master auf die Verfügbarkeit eines Slaves
   - Warten eines Prozesses als Slave auf die Verfügbarkeit eines Masters
   - Erbringung der laufenden Kooperationsleistung (Zustandsänderungen) des Masters

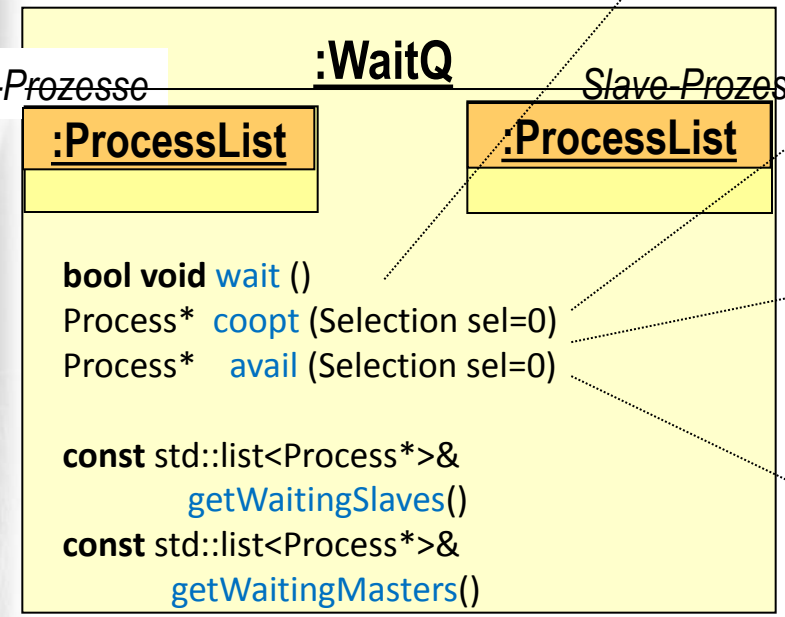
3. ein Master sollte über ein waitQ-Objekt die Verfügbarkeit eines Slaves mit bestimmten Eigenschaften fordern können
   - bestimmter Prozesstyp
   - bestimmte Attribut-Belegungen (Zustand)

WaitQ- Member-Funktionen

```
                        WaitQ (base::Simulation &sim, const data::Label &label, WaitQObserver *obs=0)
                                        // Construction for user-defined Simulation.
                        ~WaitQ ()
                                        // Destruction.
const base::ProcessList &   getWaitingSlaves () const
                                        // List of blocked slaves.
const base::ProcessList &   getWaitingMasters () const
                                        // List of blocked masters.
// Master-slave synchronisation
                    bool    wait ()
                                        // Wait for activation by a 'master' process.
                    bool    wait (base::Weight weightFct)
                                        // Wait for activation by a 'master' process.
          base::Process *   coopt (base::Selection sel=0)
                                        // Get a 'slave' process.
          base::Process *   coopt (base::Weight weightFct)
                                        // Get a 'slave' process by evaluating a weight function.
          base::Process *   avail (base::Selection sel=0)
                                        // Get available slaves without blocking (optional: select slave)
                    void    signal ()
                        //reactivate all master for rechecking of modified selection or weight conditions
```

WaitQ-Synchronisation

Achtung !
keine spezielle Funktion zur Slave-Reaktivierung
→ Verwendung von: activate(), …



Aufrufer-Prozess wird zum Slave,
- wartet auf Master-Prozess, falls keiner momentan verfügbar
- aktiviert den ersten wartenden Master-Prozess
- Rückgabewert liefert Info, ob Aktivierung vom Master (true) oder per Unterbrechung der Wartephase (false)

Aufrufer-Prozess wird zum Master,
- wartet auf Slave-Prozess (ohne Bedingung), blockiert falls keiner momentan verfügbar
- liefert den ersten wartenden Slave-Prozess per Rückgabewert, wenn verfügbar

Aufrufer-Prozess wird zum Master,
- wartet auf Slave-Prozess (der Bedingung erfüllt), blockiert falls kein solcher momentan verfügbar
- liefert den ersten wartenden Slave-Prozess per Rückgabewert, wenn verfügbar und Bedingung erfüllt

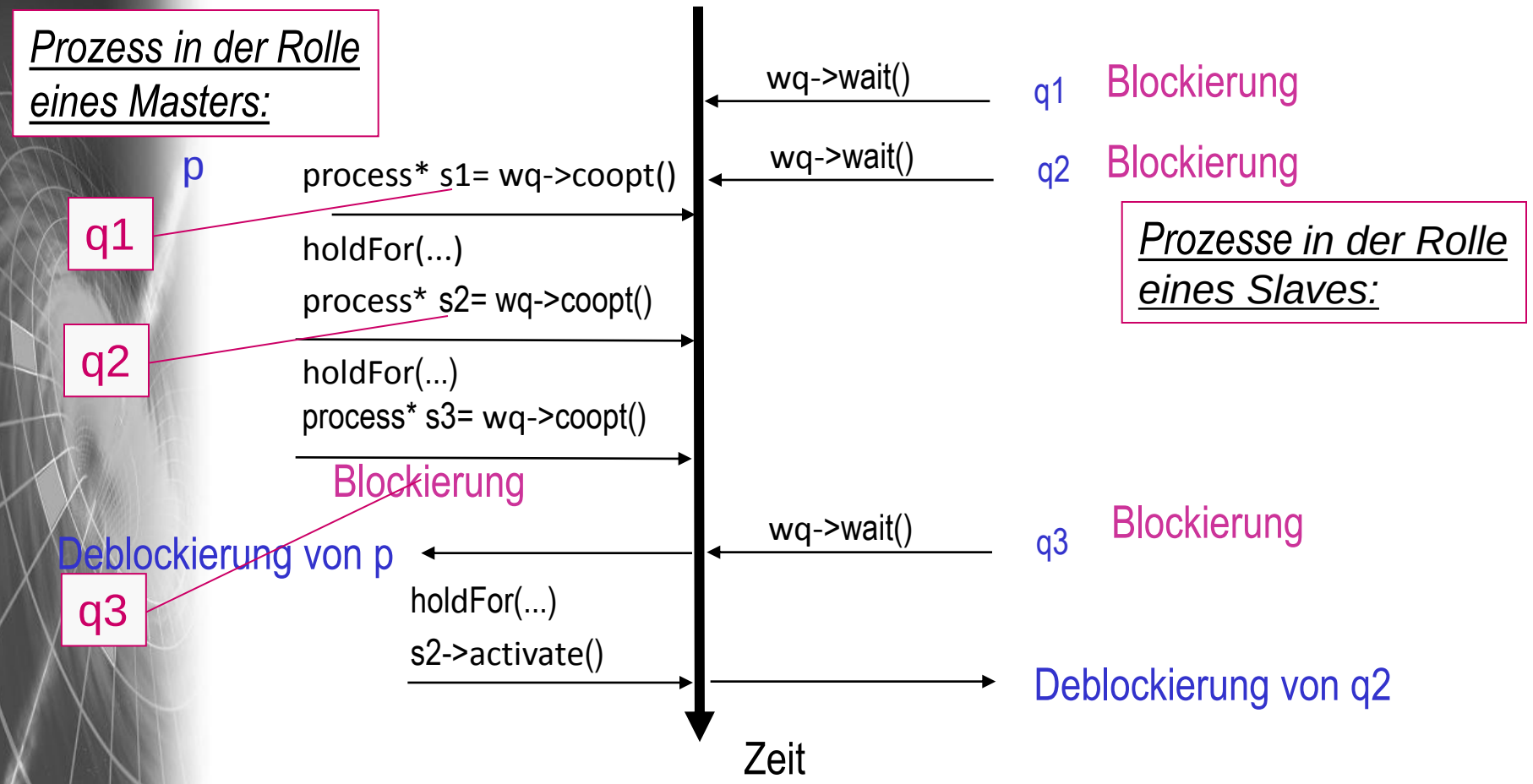
Aufrufer-Prozess wird weder Master noch Slave
- liefert ersten wartenden Slave, für den die eventuell angegebene Bedingung gilt (sonst Null-Pointer)

```
typedef bool (*Selection)(Process*);
```

bereitzustellen als Member-Funktion einer Process-Ableitung, von der Master-Objekte gebildet werden

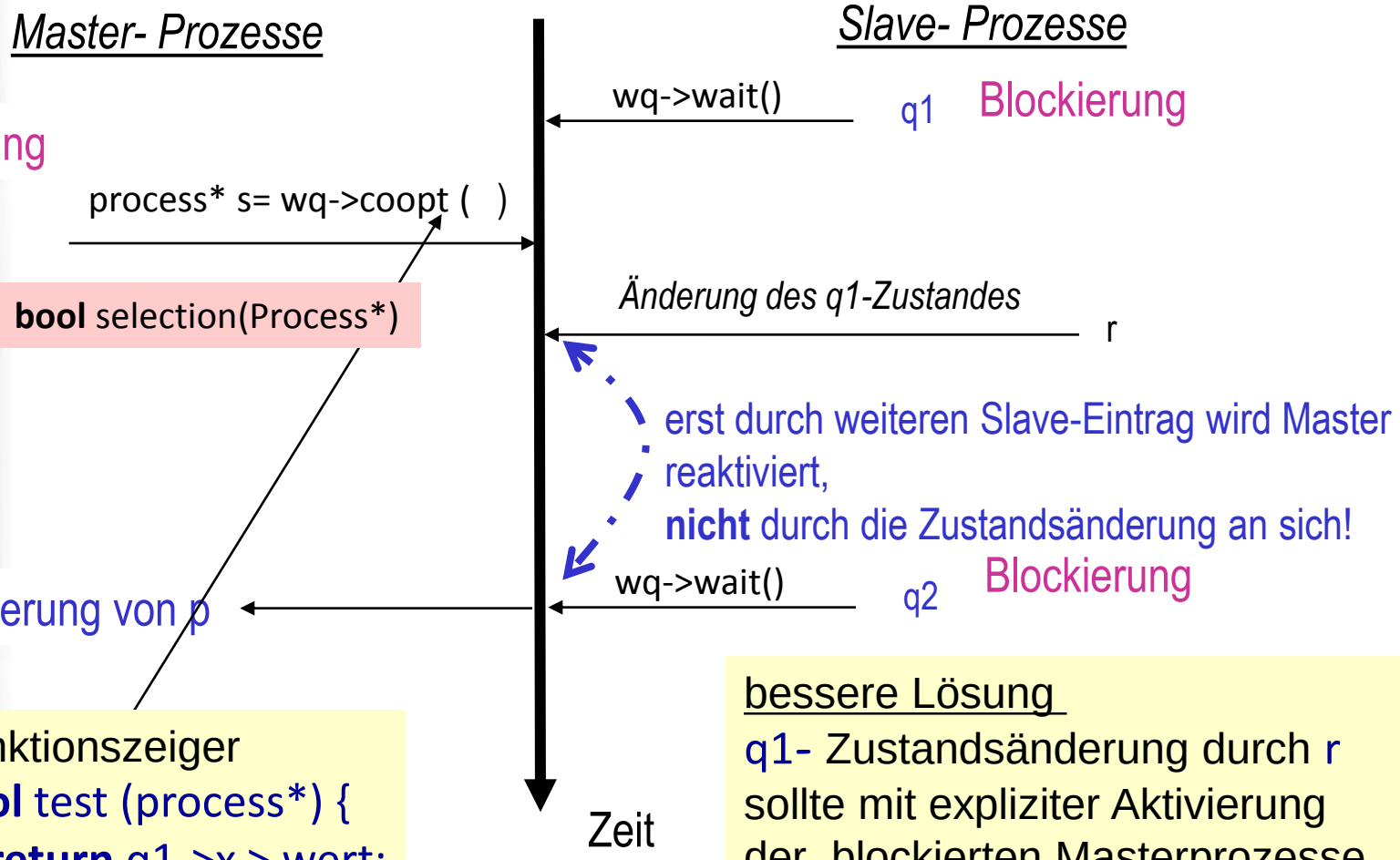
WaitQ-Synchronisation

```
Process *p, *q1, *q2, *q3;      // Zeiger auf Prozessobjekte
WaitQ *wq;
```



WaitQ-Synchronisation

```
Process *p, *q1, *q2, *r;      // Zeiger auf Prozessobjekte
WaitQ *wq;
```



Funktionszeiger

```
bool test (process*) {
    return q1->x > wert;
}
```

bessere Lösung

q1- Zustandsänderung durch r sollte mit expliziter Aktivierung der blockierten Masterprozesse in wq verbunden sein: wq->signal()