Kurs OMSI
im WiSe 2010/11

Objektorientierte Simulation mit ODEMx

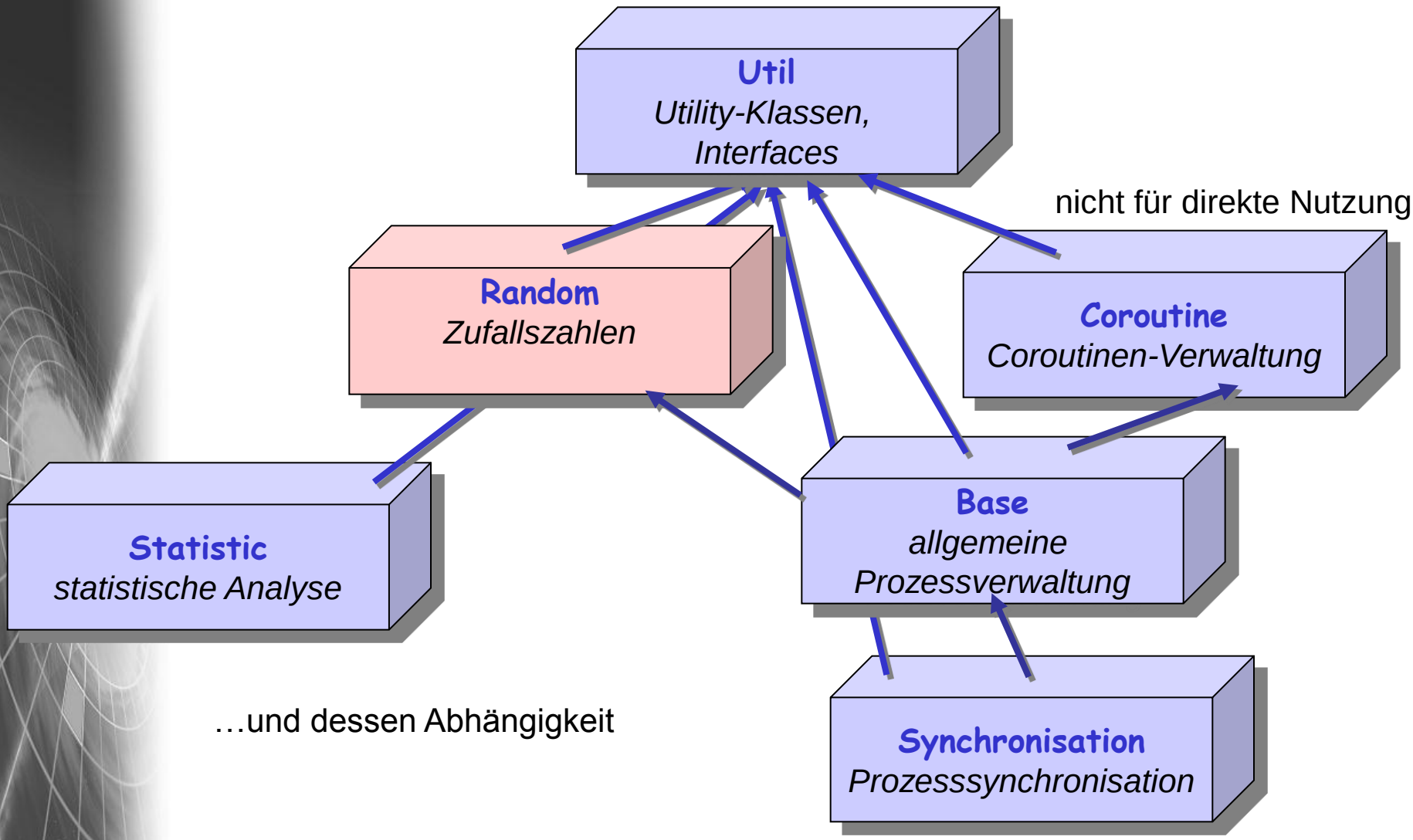
Prof. Dr. Joachim Fischer
Dr. Klaus Ahrens
Dipl.-Inf. Ingmar Eveslage

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

Der ODEMx- Modul Random



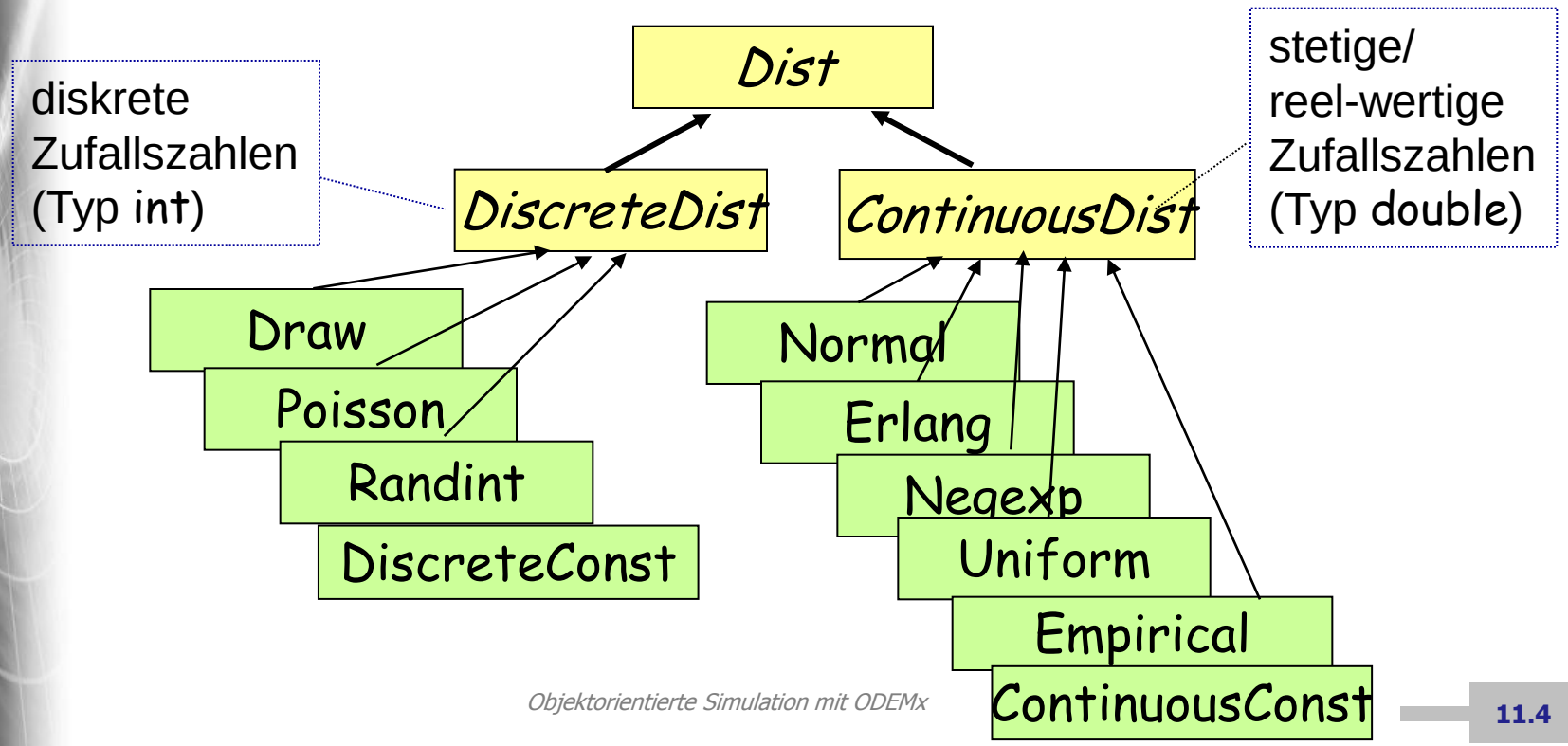
…und dessen Abhängigkeit

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

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

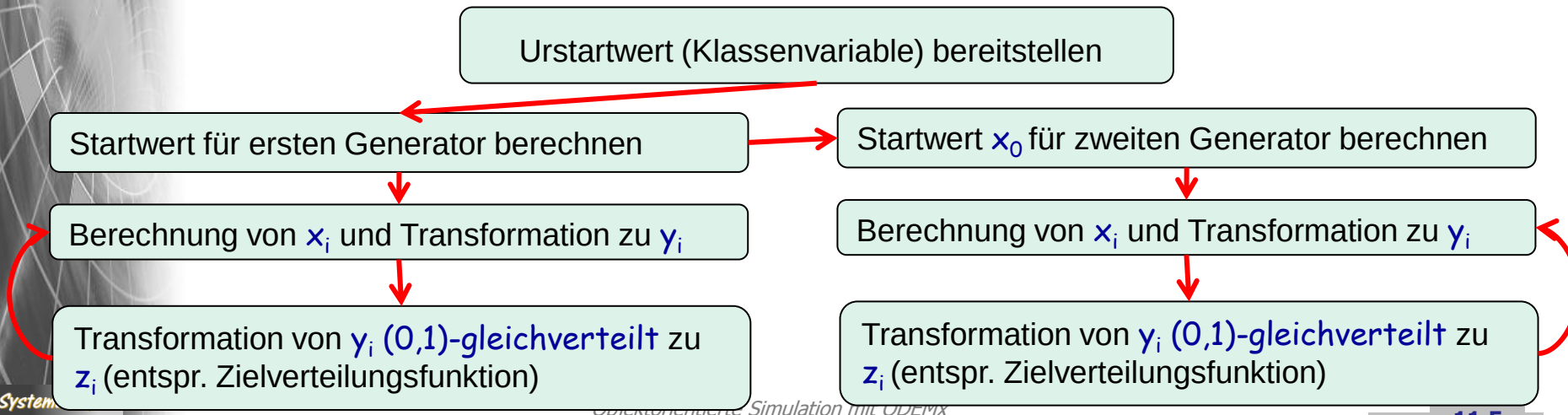
liefert Zahlenfolge mit Periode p:

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

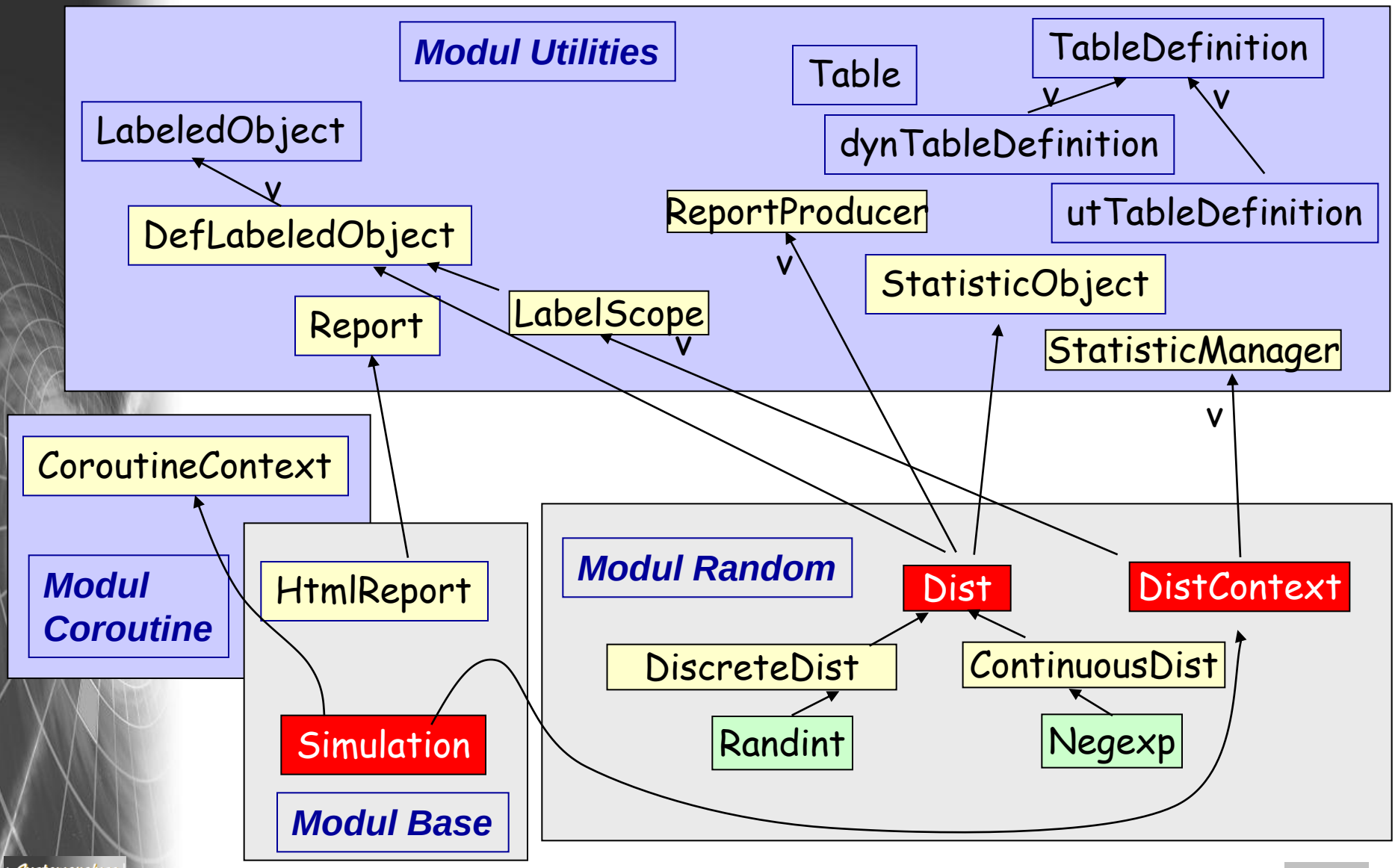
Generator für **gleichverteilte** (31 Bit-) Zufallswerte mit **maximaler Periode**

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

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

DistContext

- Ein DistContext-Objekt stellt einen gemeinsamen Kontext für verschiedene Zufallszahlengeneratoren dar (Simulation hat DistContext-Funktionalität geerbt)

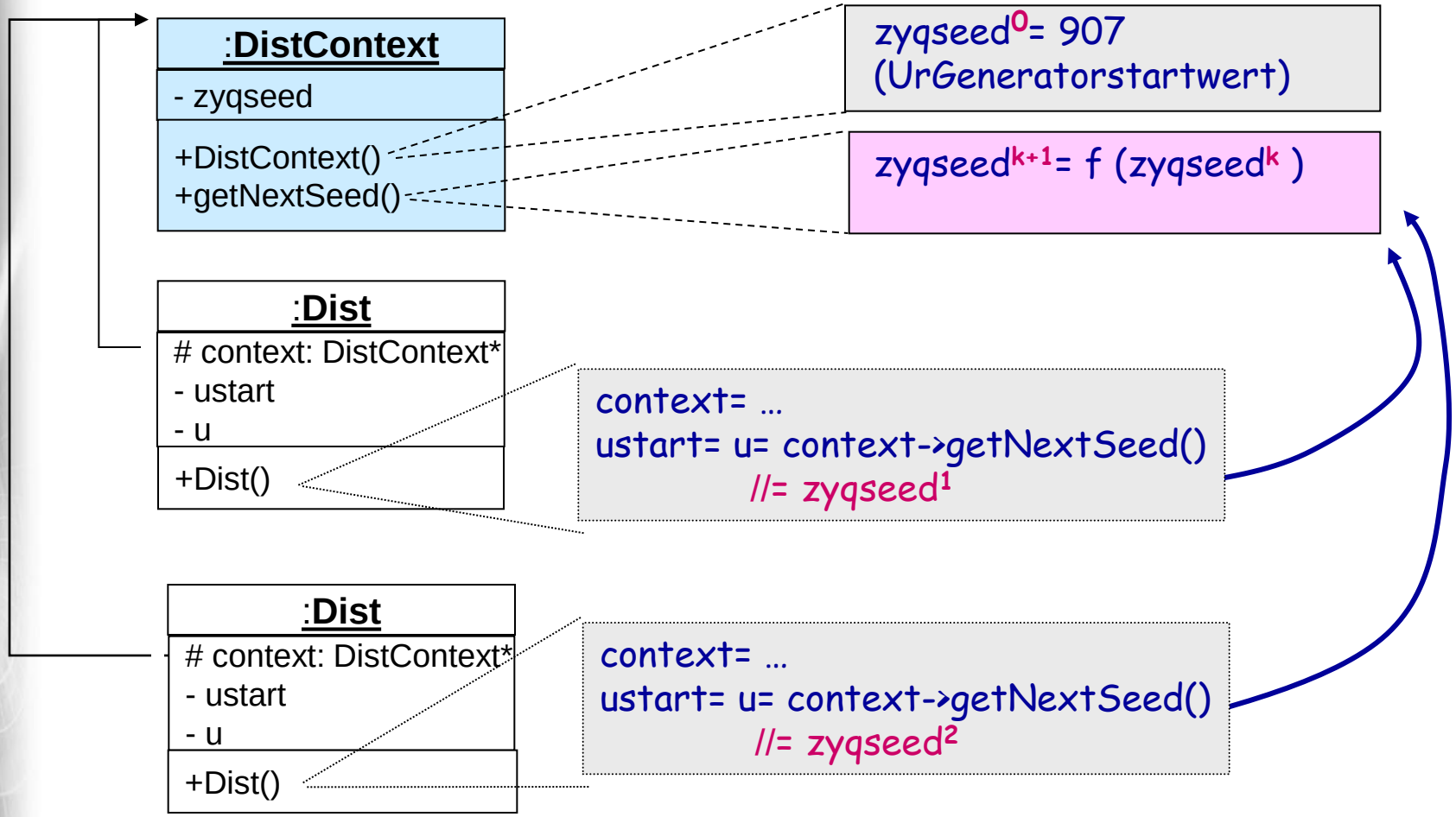
```
class DistContext : public virtual LabelScope,
                    public virtual StatisticManager {
  public:
        DistContext ();
        virtual ~DistContext();
        unsigned long getNextSeed();     // berechnet jeweils neuen
                                         // Startwert im Kontext
  protected:
        friend class Dist;
        unsigned long 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

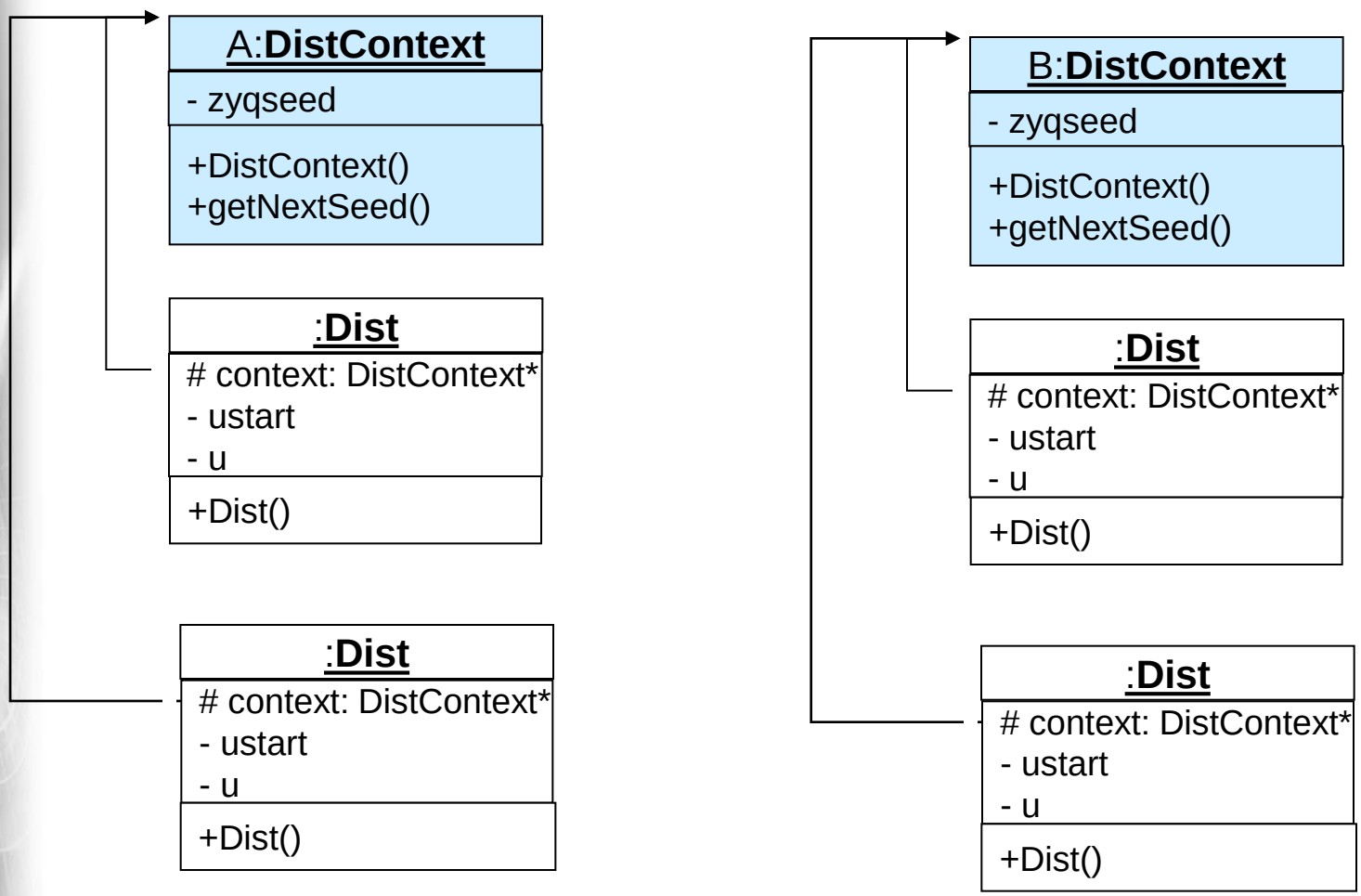
DistContext-Objekt mit zugeordneten Generator-Objekten



Zufallszahlengeneratoren sind
Objekte von Dist-Ableitungen

Mehrere DistContext-Objekte

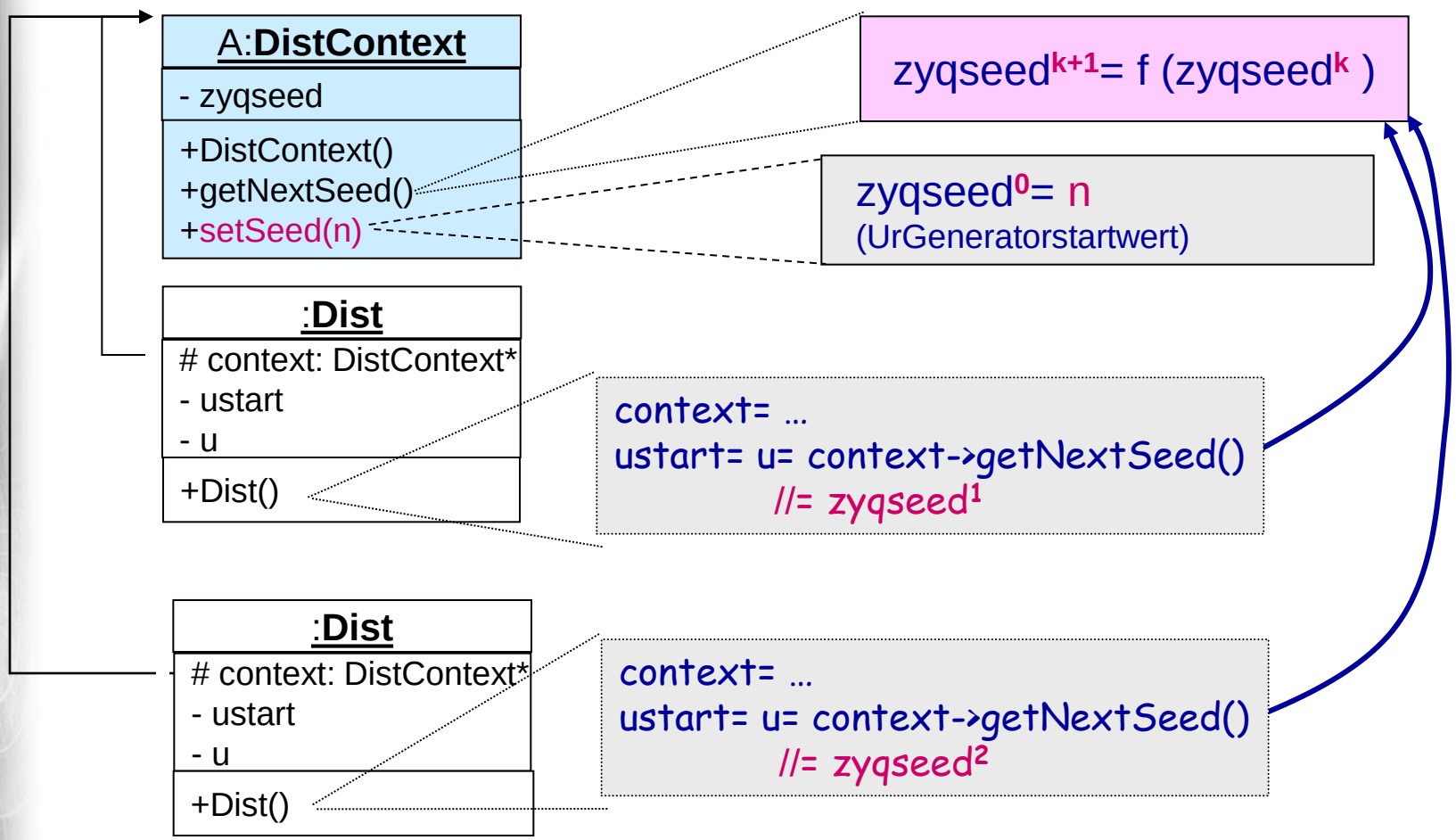
DistContext-Objekte A und B mit ihren Generator-Objekten



Eigene Simulationskontextklassen statt Defaultsimulation-Kontext

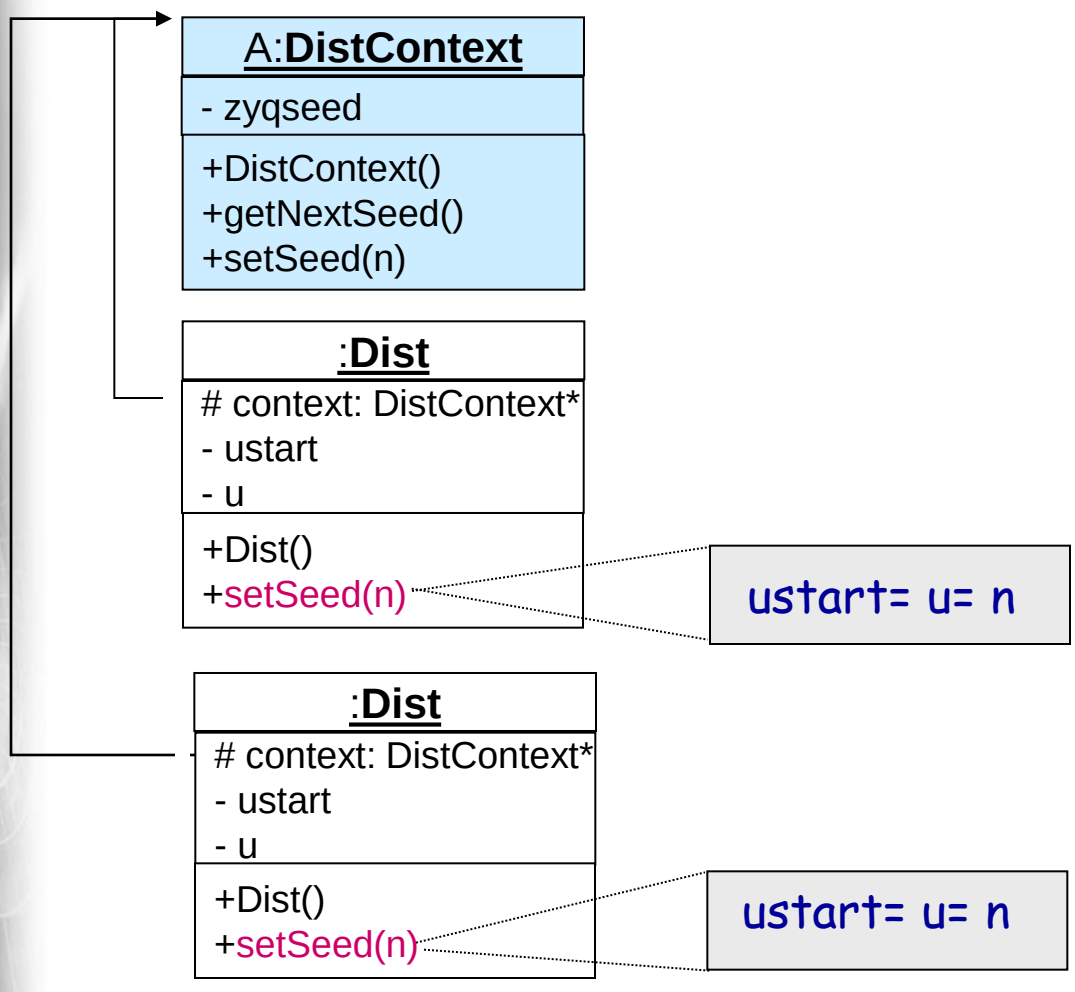
Nutzereigene Änderung von Startwerten

Variante A) explizite Bestimmung des Urwertes (DistContext)



Nutzereigene Änderung von Startwerten (Forts.)

Variante B) explizite Bestimmung individueller Dist-Startwerte



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

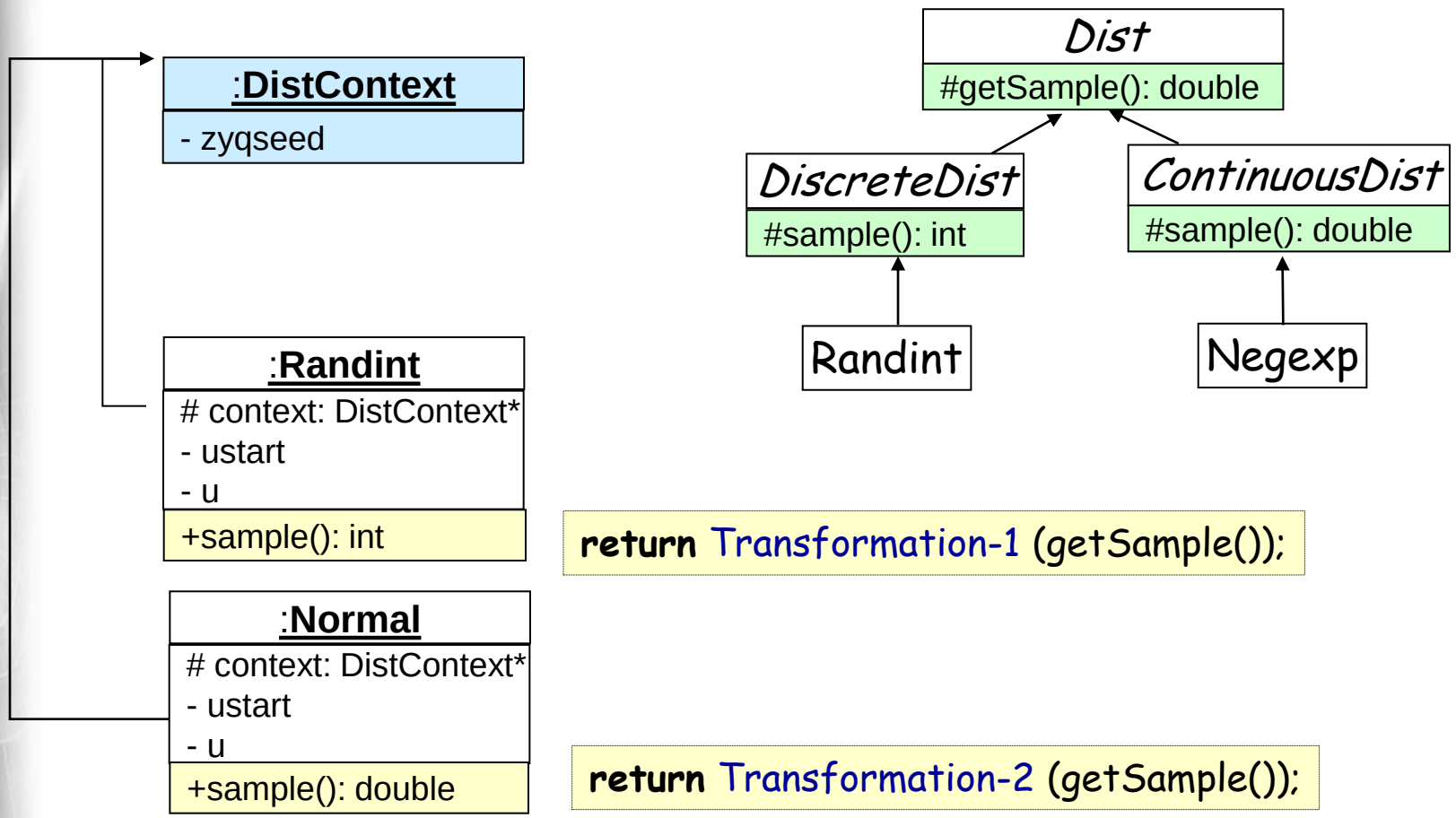
Name	Reset at	Queue	Users	Provider	Init number of token	Min number of token	Max number of token	Now number of token	Avg number of token	Avg waiting time
mainland_1	0	mainland_queue	92	94	3	0	7	5	1.99533	0
island_1	0	island_queue	93	93	1	0	8	1	1.4375	0

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



ODEMx-Generatoren stetiger Zufallsgrößen

Ableitungen von ContinuousDist

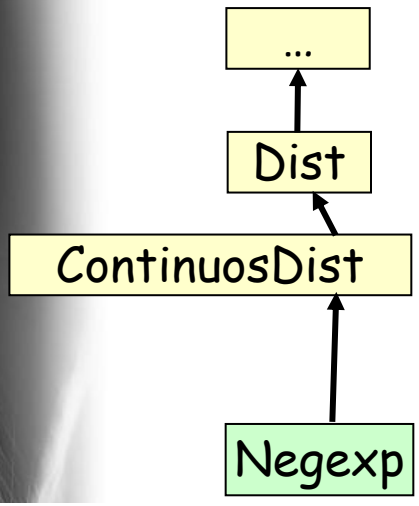
- Negexp für Exponentialverteilung
- Normal für Normalverteilung
- Uniform für Gleichverteilung
- Erlang für Erlang-Verteilung
- Empirical für Empirische Verteilung
- ContinuousConst als konstanter Wert (für einfachen Test)

einheitliche Schnittstelle
double ContinuousDist::sample()

```
ContinuousDist* myDist;
      //polymorpher Zeiger

myDist->sample()
```

Generator für exponentialverteilte Pseudo-Zufallszahlen



- Konstruktor:

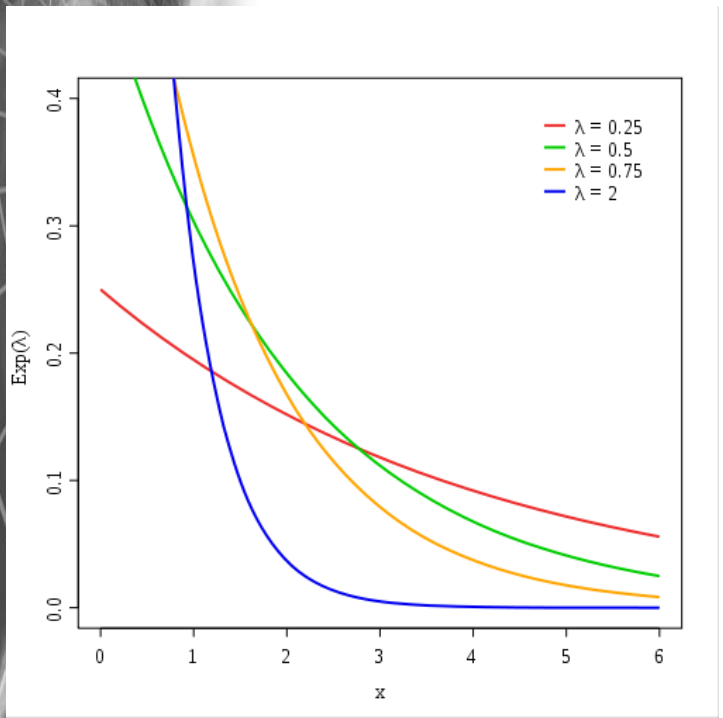
  Negexp::Negexp(DistContext* c, Label title, double na);

  // trägt Objekt in Liste des Kontextes ein

  Mittelwert= 1/na

- Konstruktor-Aufruf

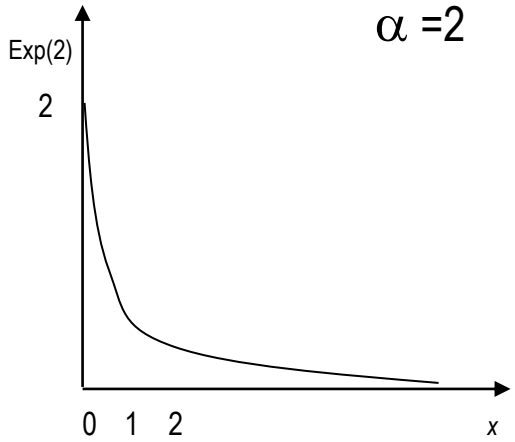
```
Dist* arrival; Simulation* sim;
arrival= new Negexp(sim, "Ankunft", 2.0);
```



Generator-Objekt-Anwendung:

```
arrival->sample(); //liefert double-Wert
```

...



$\mu = 1/\lambda$

$\sigma^2 = 1/\lambda^2$

Generator für exponentialverteilte Pseudo-Zufallszahlen

Transformationsgenerator für **exponential- verteilte Zufallswerte** mit Erwartungswert 1/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 1/a

Verwendung im Prozessablauf

Wichtige Anwendung

- Bestimmung von Verzögerungszeiten von Prozessen

```
class factory: public process {
    negexp *delay;
    int main();
    factory(negexp* d): process (defaultSimulation(), "Ingot"),
                                delay(d) {}
}
```

```
int factory::main() {
    for (;;) {
        new ingot(...)->activate();
        holdFor (delay->sample());
    }
    return 0;
}
```

Generator für normalverteilte Pseudo-Zufallszahlen

ODEMx-Transformationsgenerator für **exponential- verteilte Zufallswerte**
mit Erwartungswert μ und Standardabweichung σ

Dichtefunktion: $f(x) = \frac{1}{\sigma\sqrt{2\pi}} a * exp \left\{ \frac{1}{2} \left[\frac{x - \mu}{\sigma} \right]^2 \right\}$

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 $\sigma = 1.0$

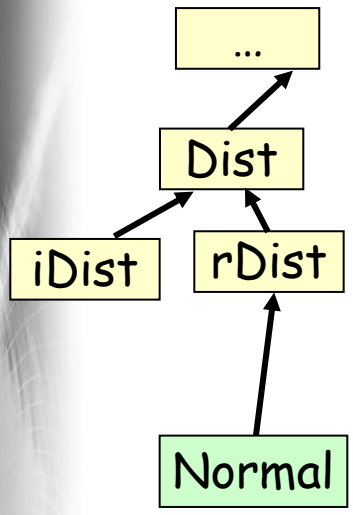
normal-verteilte Zufallswerte
mit Erwartungswert μ
und
Standardabweichung σ

Generator für normalverteilte Pseudo-Zufallszahlen (3)

Standardabweichung
Mittelwert

- Konstruktor:

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

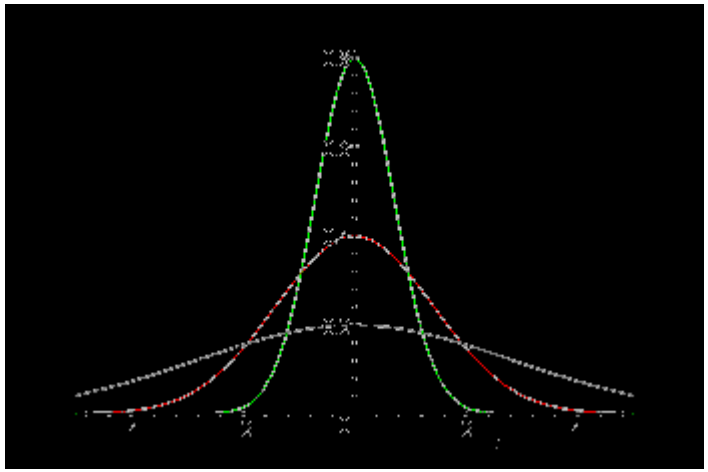


- Konstruktor-Aufruf

```
Dist* duration; Simulation* sim; ...
duration= new Normal(sim, "Dauer", 1.0, 5.2);
```

- Generator-Objekt-Anwendung:

```
duration->sample(); //liefert double-Wert
```

- 

verschiedene
Standardabweichungen
0.5
1.0
2.0
bei Mittelwert 0.0

Generator für gleichverteilte reelle Pseudo-Zufallszahlen

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

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

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

Generator *für gleichverteilte reelle Pseudo-Zufallszahlen*

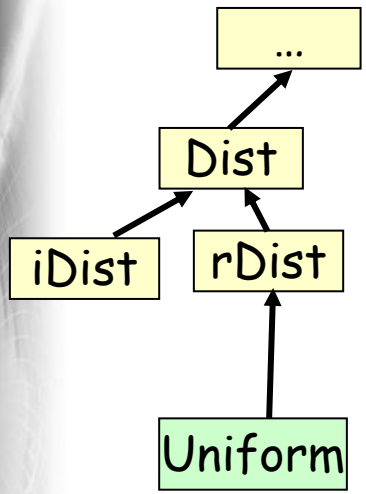
rechter Rand

linker Rand

- Konstruktor:

  Uniform::Uniform(DistContext* c, Label title, double na, nb);

  trägt Objekt in Liste des Kontextes ein

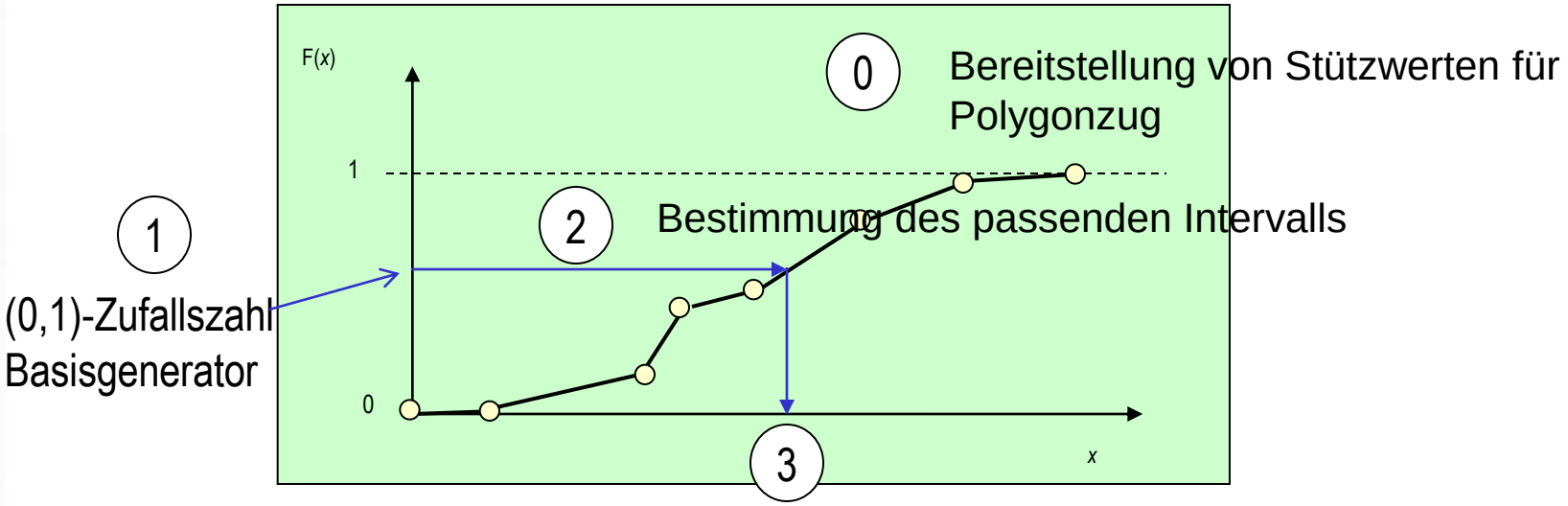


Generator für empirischverteilte Pseudo-Zufallszahlen

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

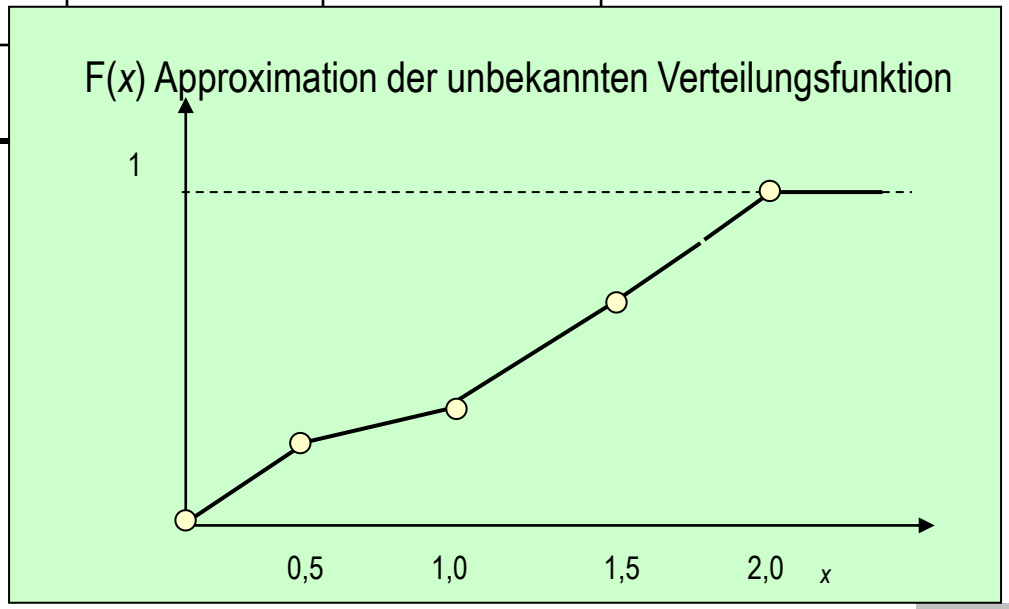


Bestimmung des x-Wertes (Zufallsgröße, die der empirischen F(x) genügt)

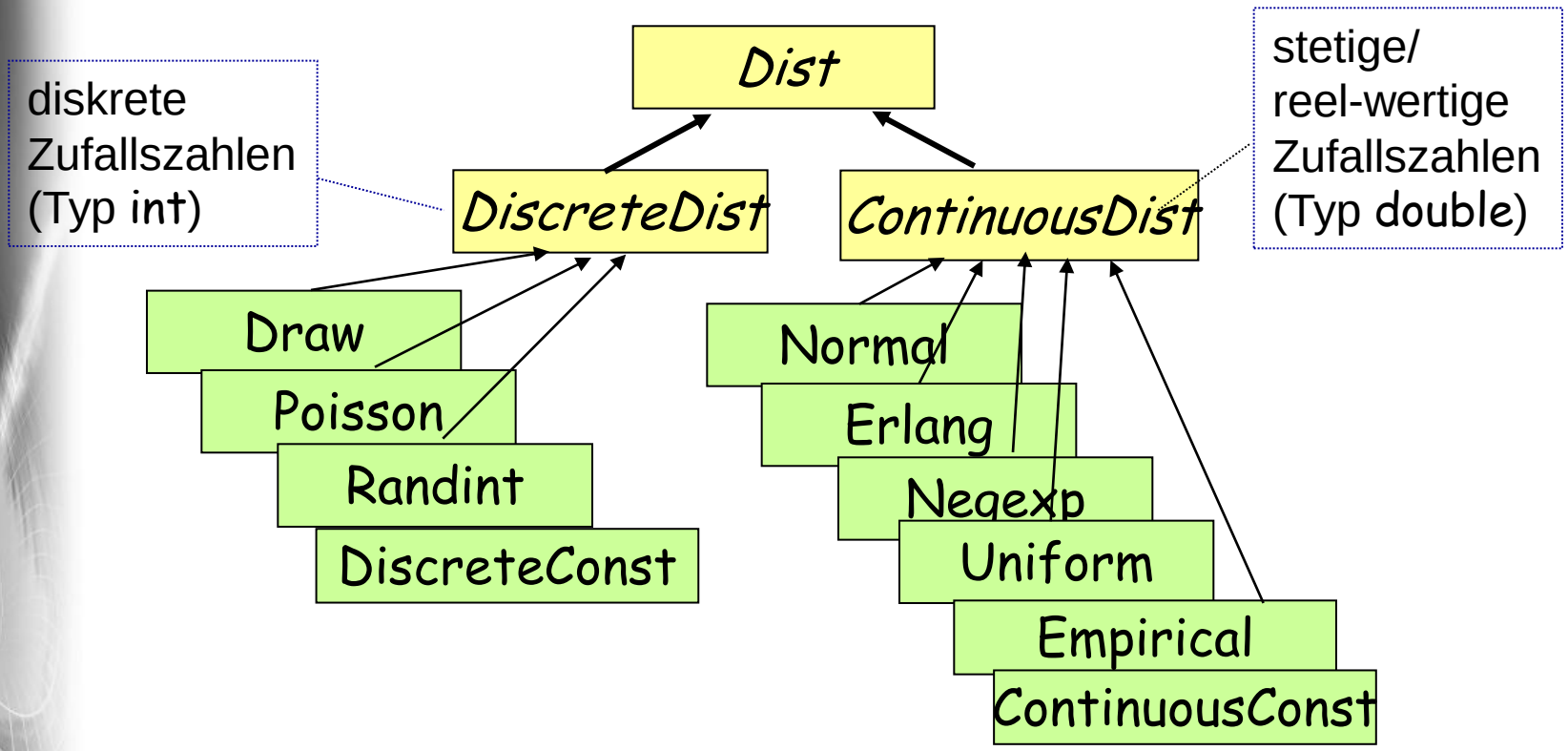
Empirische Verteilungen

Beispiel: Aufzeichnung von 100 Reparaturzeiten

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		



Alle ZZ-Generatoren von ODEMx



ODEMx-Generatoren diskreter Zufallsgrößen

Ableitungen von iDist

- Randint für Gleichverteilung
- Draw für (0,1)-Entscheidungen
- Poisson für Poisson-Verteilung
- iConst für einen konstanten Wert

einheitliche Schnittstelle
int iDist::sample()

```
iDist* myDist;
//polymorpher Zeiger
myDist->sample()
```

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

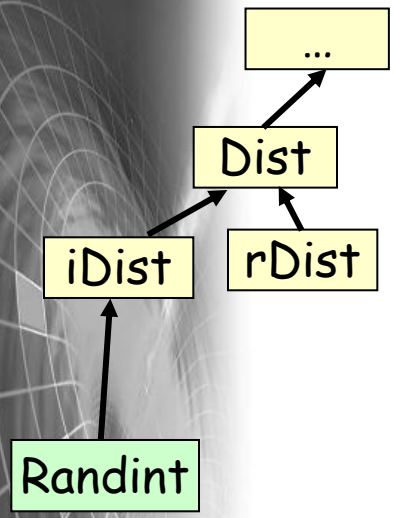
Vor.: sei $\{ y_i \}$ eine (0, 1)- verteilte Zufallszahlenfolge

Generator *für gleichverteilte diskrete Pseudo-Zufallszahlen*

rechter Rand
linker Rand

- Konstruktor:

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

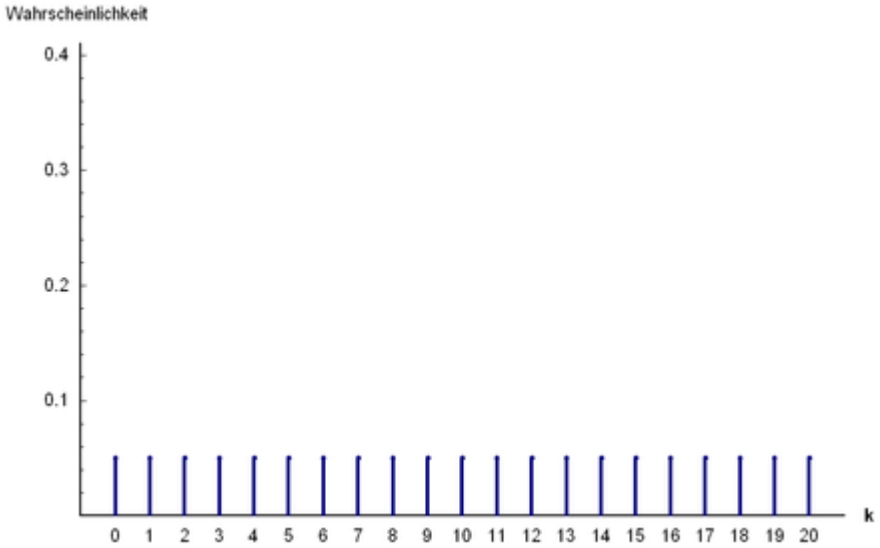


- Konstruktor-Aufruf

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

- Anwendung:

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

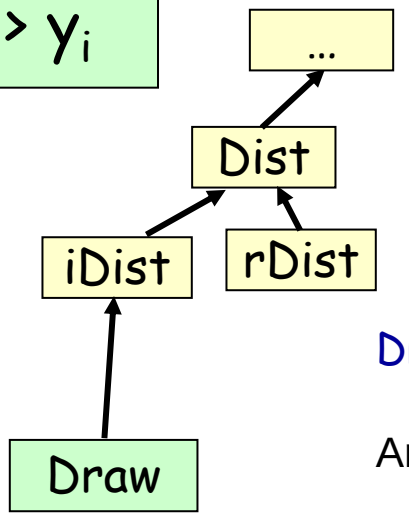


Generator für 0/1-Entscheidung

Transformationsgenerator für zufällige **0-1-Entscheidungen** bei Vorgabe einer Wahrscheinlichkeit p für den Wert 1 (true)

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

$x_i = p > y_i$



Wahrscheinlichkeit für **true**

```
Draw::Draw (DistContext* c, Label title, double p);
```

Anwendung

```
Dist* muenze; Simulation* sim;

muenze= new Draw (sim,"Muenze", 0.5);

muenze ->sample(); //liefert int-Wert
```

6. ODEMx-Modul Synchronisation:

WaitQ, CondQ

- Konzept WaitQ
  - Beispiel: Tankerflotte, Hafen, Raffinerie
- Konzept CondQ
  - Beispiel: Hafen, Schlepper, Gezeiten
- Weitere Anwendungsbeispiele

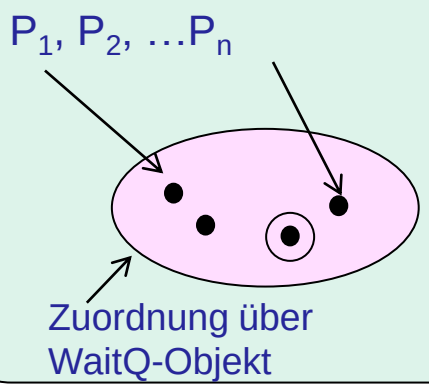
für WaitQ und CondQ

Motivation von WaitQ

- allgemeines Synchronisationsproblem zwischen Prozessen
  bei Auszeichnung dynamischer **Rollen** in der Erbringung einer gemeinsamen Kooperationsleistung

  – **Rolle** Slave :
  sind Ressourcen/Kooperationspartner zugeordneter Master-Prozesse zur Realisierung der gemeinsamen (zeitlich befristeten) Kooperationsleistung

  – **Rolle** Master :
  sind **alleinige** Erbringer dieser Kooperationsleistung, Slave-Partner bleiben während der Kooperation passiv, müssen nach Abschluss der Kooperation vom Master aktiviert werden



◉ Master-Rolle, ● Slave-Rolle

temporäres Prozessensemble
bei Auszeichnung eines Prozesses als
Stellvertreter (Master) zur Realisierung wechselseitig
abhängiger Zustandsänderungen

Entschärfung der Parallelität von Wechselwirkungen bei Zustandsänderungen im Simulator

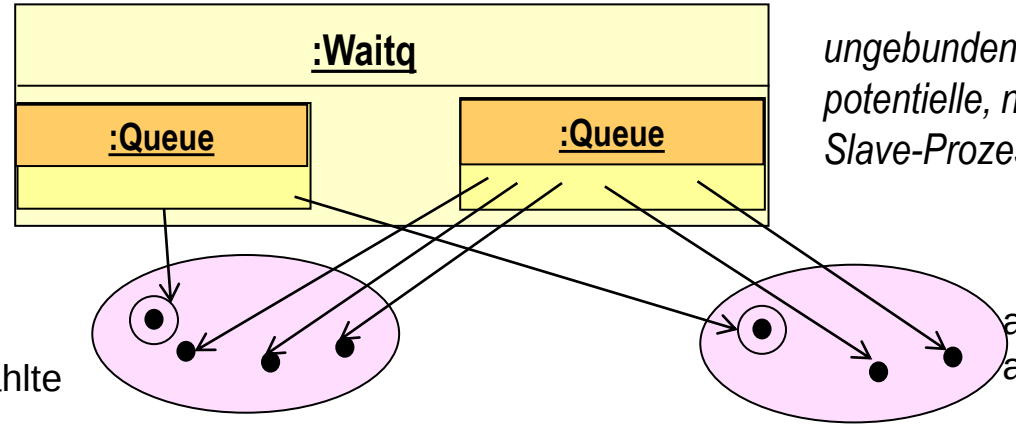
WaitQ-Konzept

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 wieder frei)
- **Master** realisiert **allein** die entsprechenden Zustandsänderungen, die mit der Kooperation aller Partner verbunden sind
  (benötigt entsprechende Zugriffsrechte auf seine Klienten)

ungebundene potentielle, noch blockierte Master-Prozesse



ungebundene potentielle, noch blockierte Slave-Prozesse

aktiver Master verwaltet temporär ausgewählte Slaves

anderer aktiver Master, andere Slaves

WaitQ-Anwendungsmuster

Variation der Autofähre

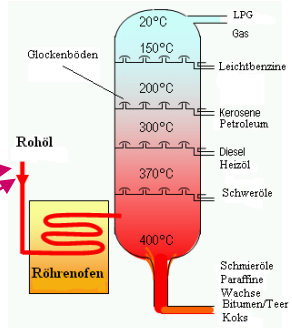
Annahme: Modellierung der Autos ebenfalls als Prozesse notwendig

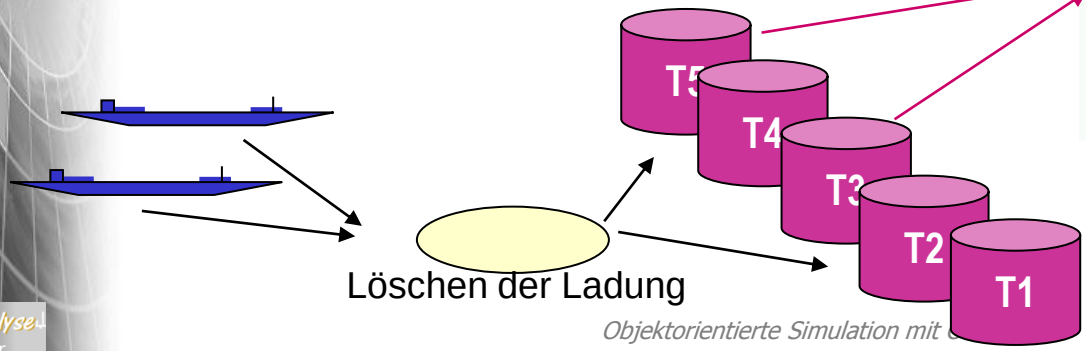
- für den Transport ordnen sie sich als Slave-Prozesse
- einem Master-Objekt (Fähre)
- an einer Anlegestelle (WaitQ-Objekt) unter,

vor und nach dem Transport sind die Autos in ihrer Rolle nicht festgelegt

Entladung von Öl-Tankern

Modellierung der Tank-Schiffe als Master
Modellierung der Tank-Behälter (Zwischenlager für eine Raffinerie) als Slave-Prozesse



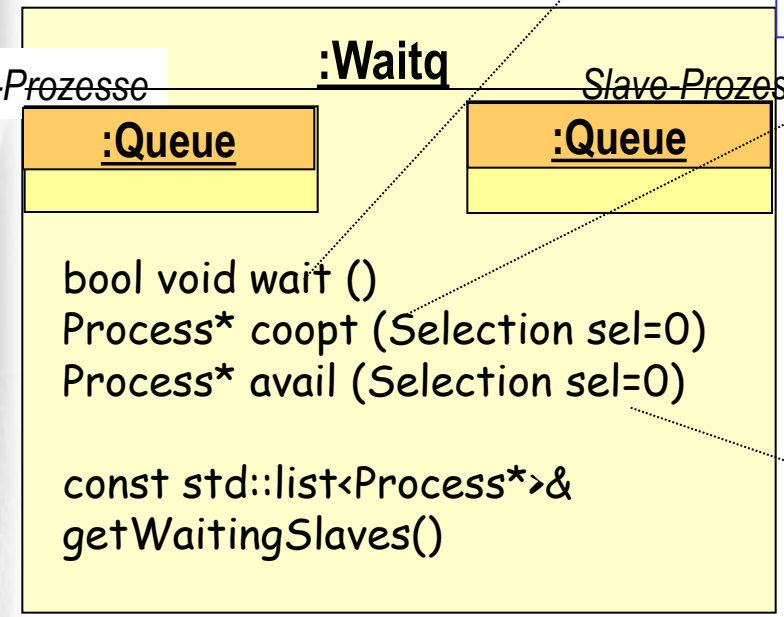


Auswahlbedingung:
- Tank kann komplette Tankerladung übernehmen (→ Tanker muss nicht umgesetzt werden)

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, blockiert falls keiner momentan verfügbar
- liefert den ersten wartenden Slave-Prozess per Rückgabewert, wenn verfügbar

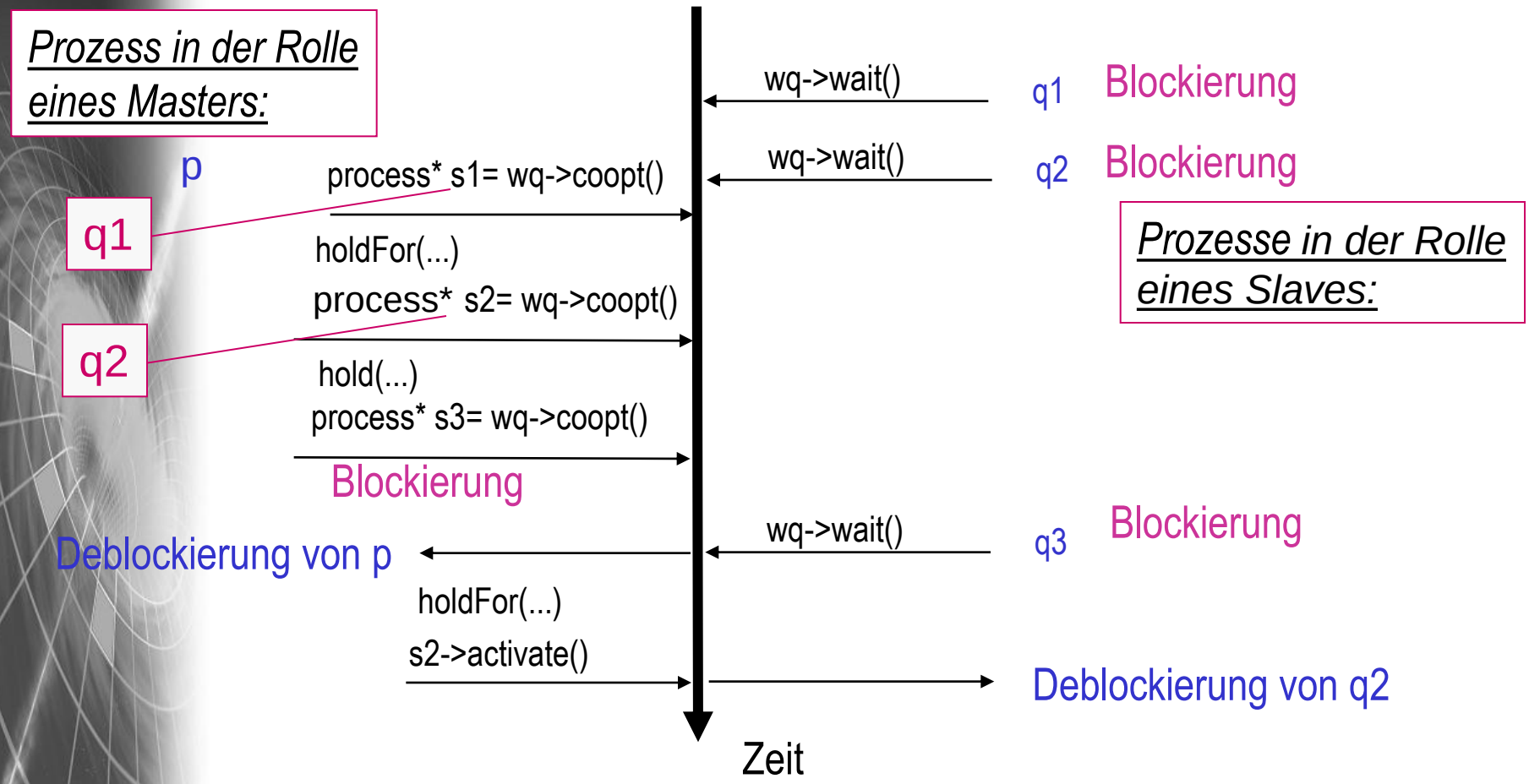
Aufrufer-Prozess wird weder Master noch Slave
- liefert ersten wartenden Slave, für den Bedingung gilt (sonst Null-Pointer)

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

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

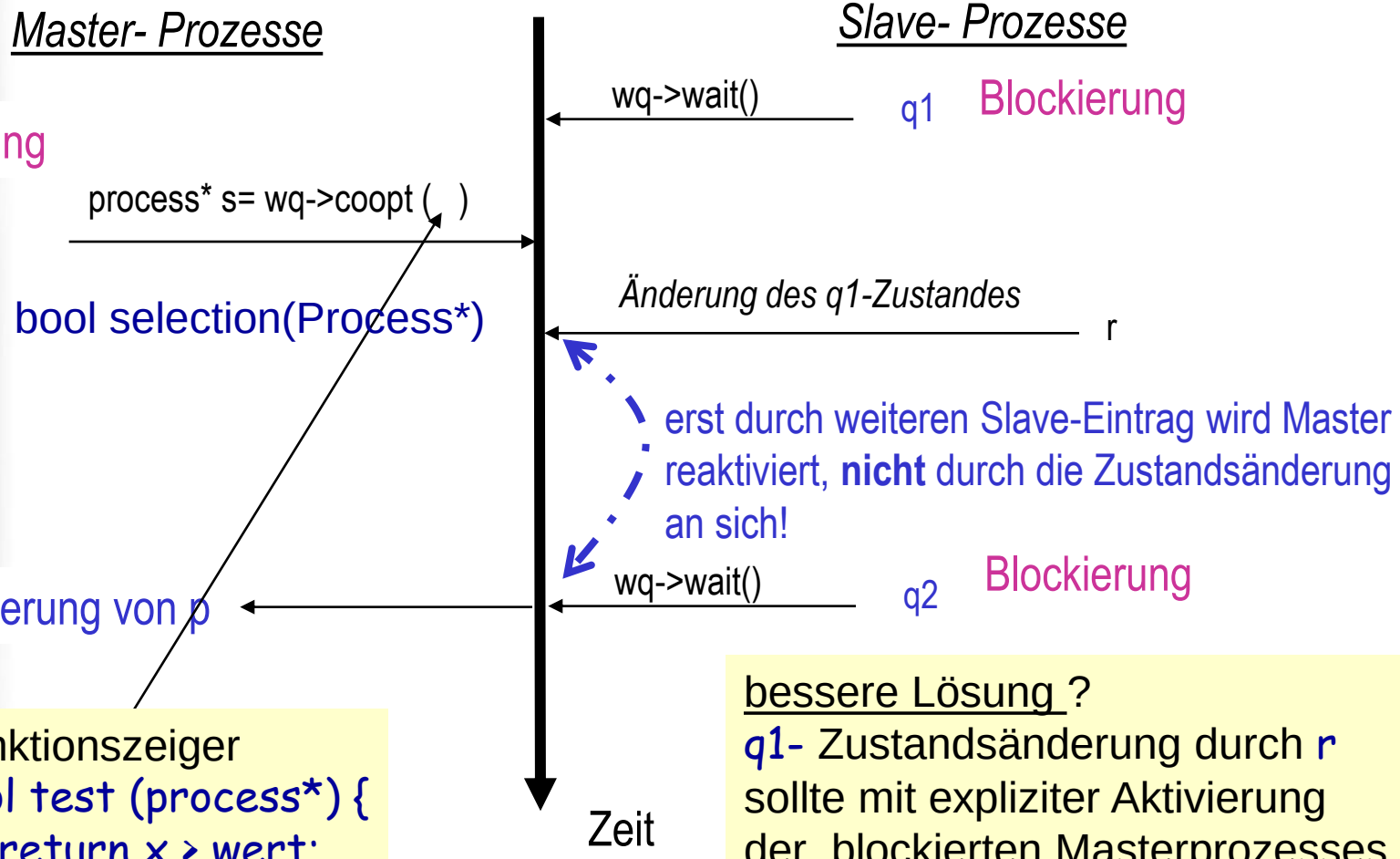
WaitQ-Synchronisation

```
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 x > wert;
}
```

bessere Lösung ?

q1- Zustandsänderung durch r sollte mit expliziter Aktivierung der blockierten Masterprozesses in wq verbunden sein

Unterbrechung wartender Master- u. Slave-Prozesse

- das evtl. Warten auf den Partner-Prozess
  kann sowohl beim Master- als auch beim Slave-Prozess mittels
  interrupt abgebrochen werden:

  in diesem Fall liefert

  - coopt() einen 0-Zeiger

  - wait() den Wert false

  ACHTUNG:
  ein Master sollte jedoch seinen erwählten Slave nicht per interrupt()
  aktivieren !!! (sondern per activate())

  nur dann liefert wait() den Wert true