Kurs OMSI
im WiSe 2013/14

Objektorientierte Simulation mit ODEMx

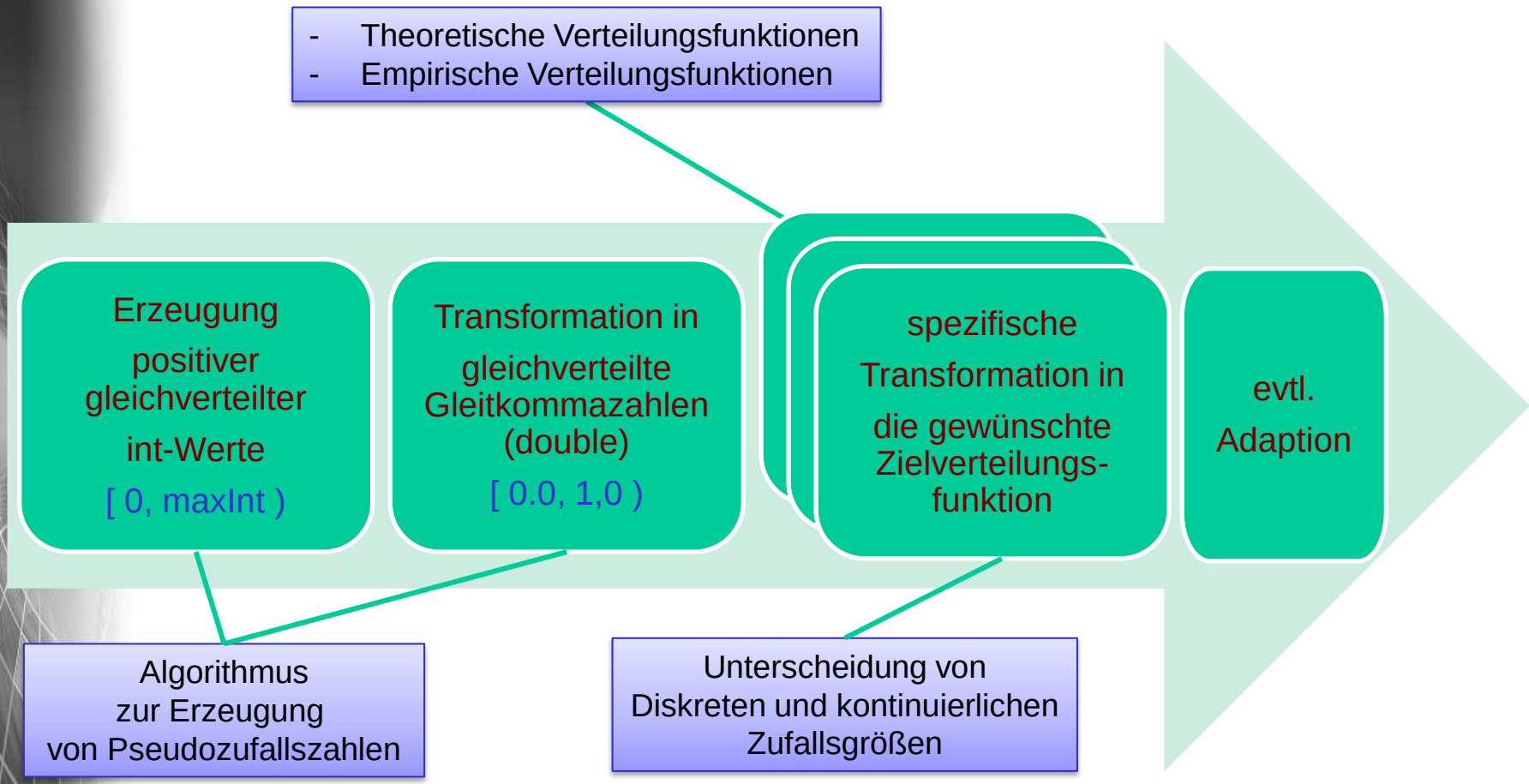
Prof. Dr. Joachim Fischer
Dr. Klaus Ahrens
Dipl.-Inf. Ingmar Eveslage

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

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
7. Lösung: Autofähre

Schema zur Berechnung von Zufallszahlen



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
7. Lösung: Autofähre

(0,1)- Pseudo-Zufallszahlen in ODEMx

Iterationsverfahren mit Startwert x_0

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

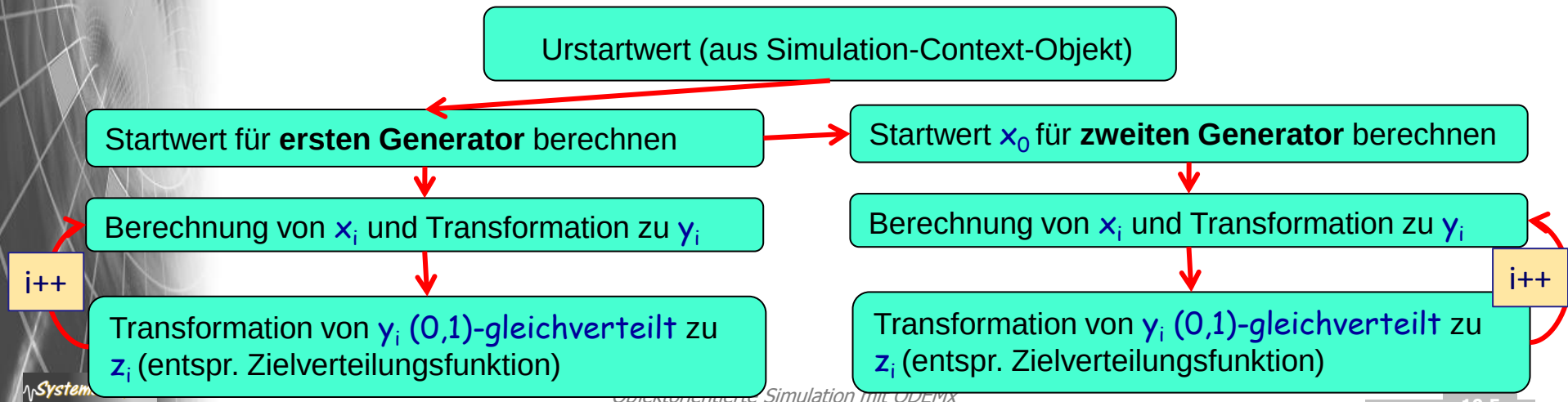
liefert Zahlenfolge mit Periode p:

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

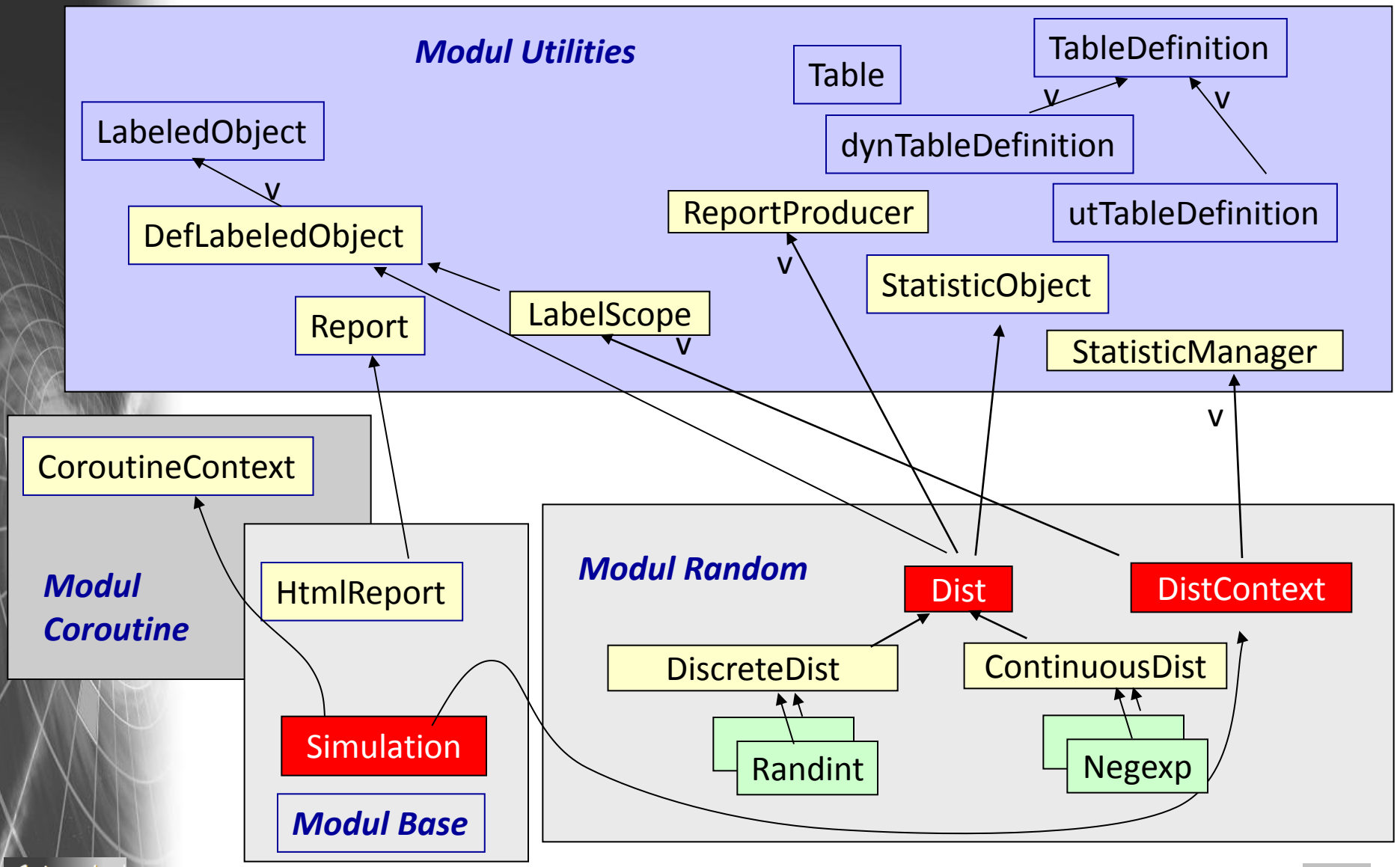
Generator für **gleichverteilte** (31 Bit-) Zufallswerte mit **maximaler Periode**

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

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



ZZ-Generatoren in der ODEMx-Klassenhierarchie



Abstrakte Klasse: Dist

... liefert Funktionalität zur Erzeugung einer (0,1)-gleichverteilten Zahlenfolge

ODEMx-ZZ-Generatoren sind Objekte von Dist- Ableitungen

```
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;
```

Frage:
woher bezieht
ein Dist-Objekt
seinen individuellen
Startwert ?

Antwort: i.d.R.
iterativ von seinem
(Simulations-)kontext
über seinen Konstruktor
aus einem Ur-Startwert

oder
nutzerspezifisch

u wird mit ustart initialisiert und dann iterativ verändert

DistContext *(abstrakte Basisklasse von Simulation)*

- Ein Simulation-Objekt stellt damit **auch** einen gemeinsamen Kontext für all seine (verschiedenen) ZZ-Generatoren dar

  die Zuordnung vollzieht sich bei Generierung der ZZ-Generator-Objekte

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

$zyqseed^{k+1} = f(zyqseed^{k})$

$zyqseed^{0} = 907$ (UrGeneratorstartwert)

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
7. Lösung: Autofähre

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

Anzahl der Aufrufe: ~ 95-1 Autos auf Mainland

Anzahl der Aufrufe: ~ 72 Überfahrten

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

Es gibt ja nur eine Fähre

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

bislang keine Wartezeit auf Autos

Anzahl der Beladungen: 92

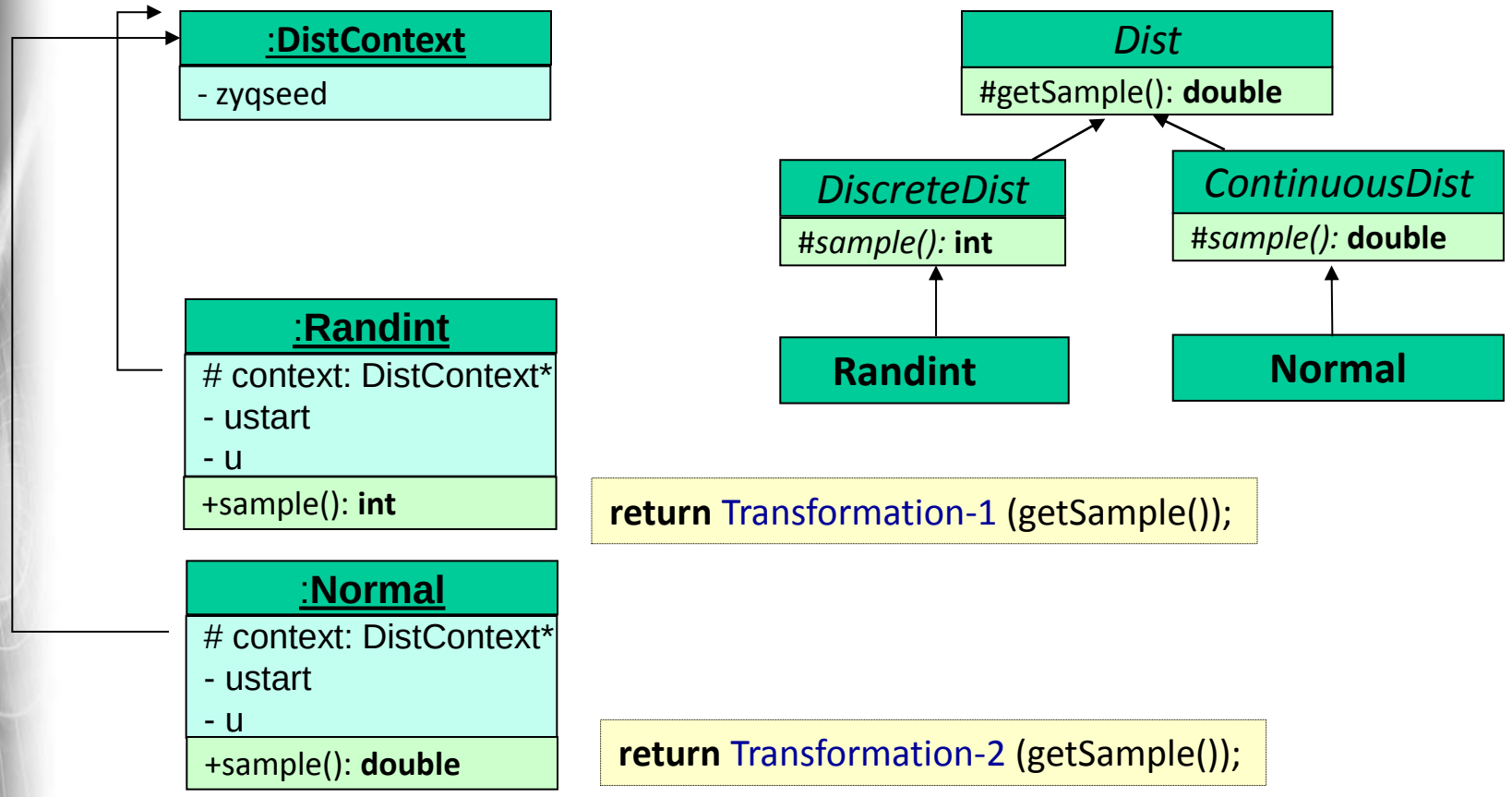
5. ODEMx-Modul Random

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

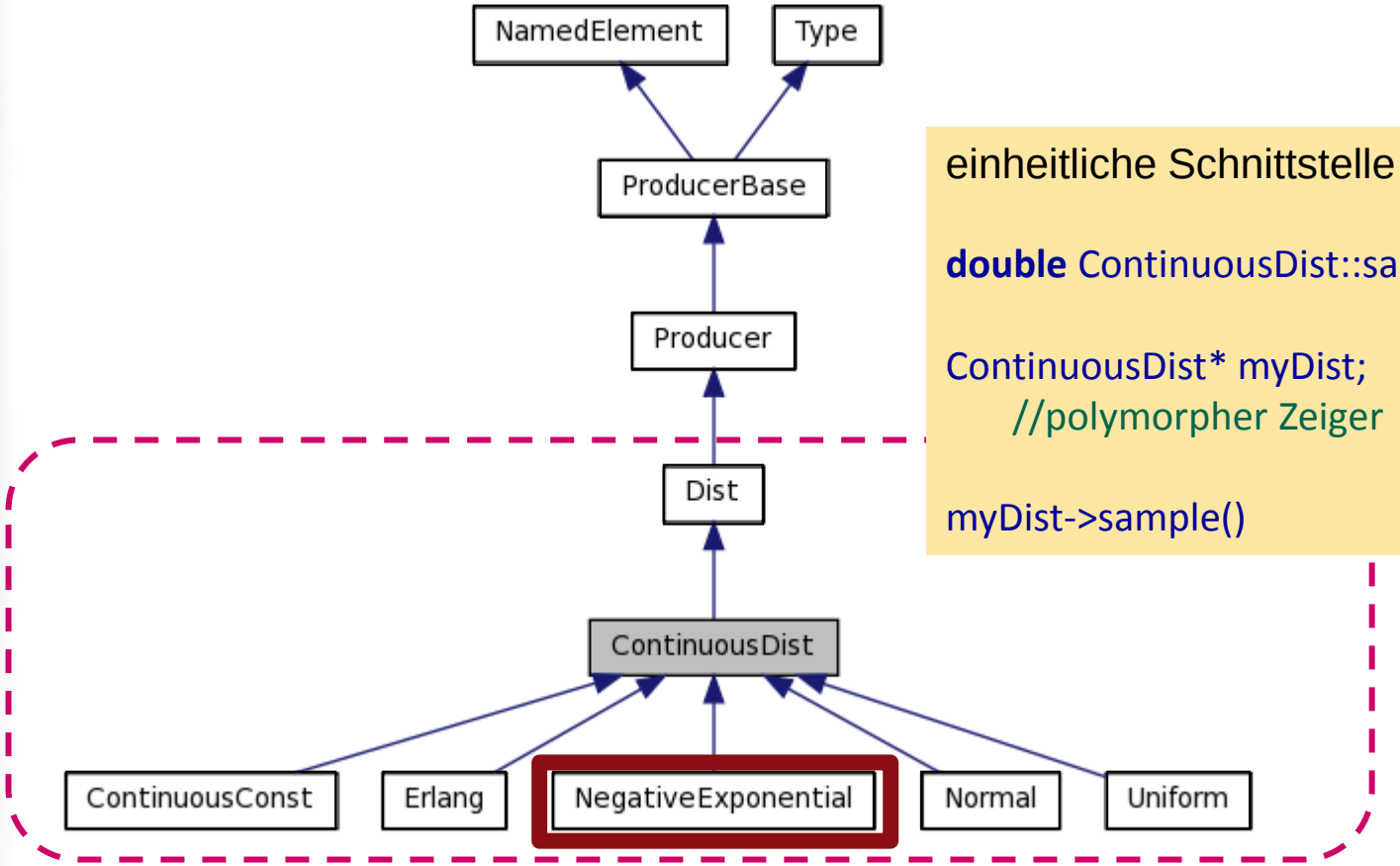
Schema zur Berechnung von Zufallszahlen

redefinierte Member-Funktion sample() einer
- DiscreteDist- bzw.
- ContinuousDist-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 α

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

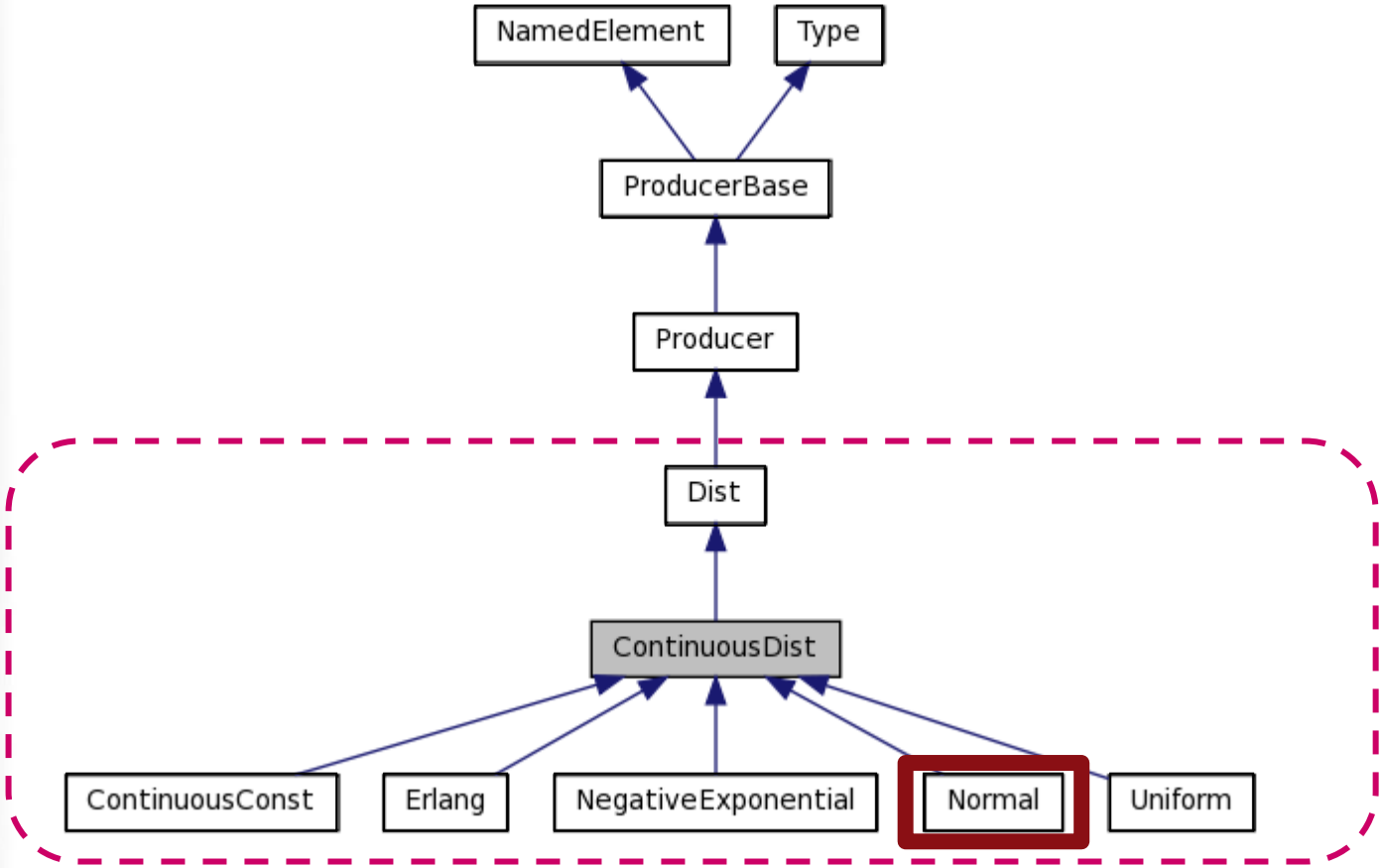
$\{y_i\}$ sei (0, 1)- verteilte Zufallszahlenfolge von Dist

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

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

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

Stetige Verteilungsfunktionen



Generator für normalverteilte Pseudo-Zufallszahlen

Transformation erfolgt über mehrere Schritte

ODEMx-Lösung:

seien y_i **und** y_{i+1} zwei aufeinander folgende Werte einer (0, 1)-verteilten Zufallszahlenfolge, dann entsteht nach folg. Alg. eine neue Folge

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

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

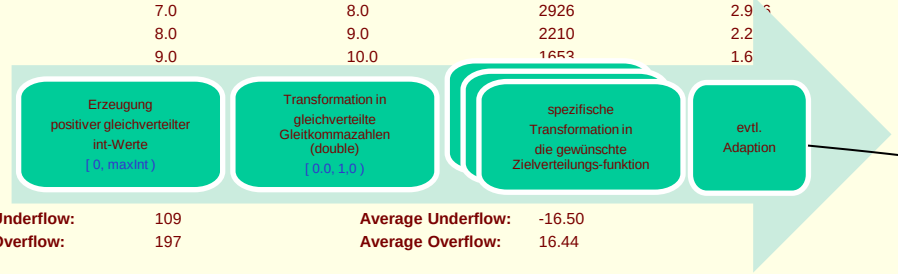
➔ $x_0, x_1, x_2, \ldots, x_{p-1}, x_p = x_0, x_1, x_2, \ldots$

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	| **********
				| ******
				| ****
				| **
				| *
				|



leistet ODEMx noch nicht

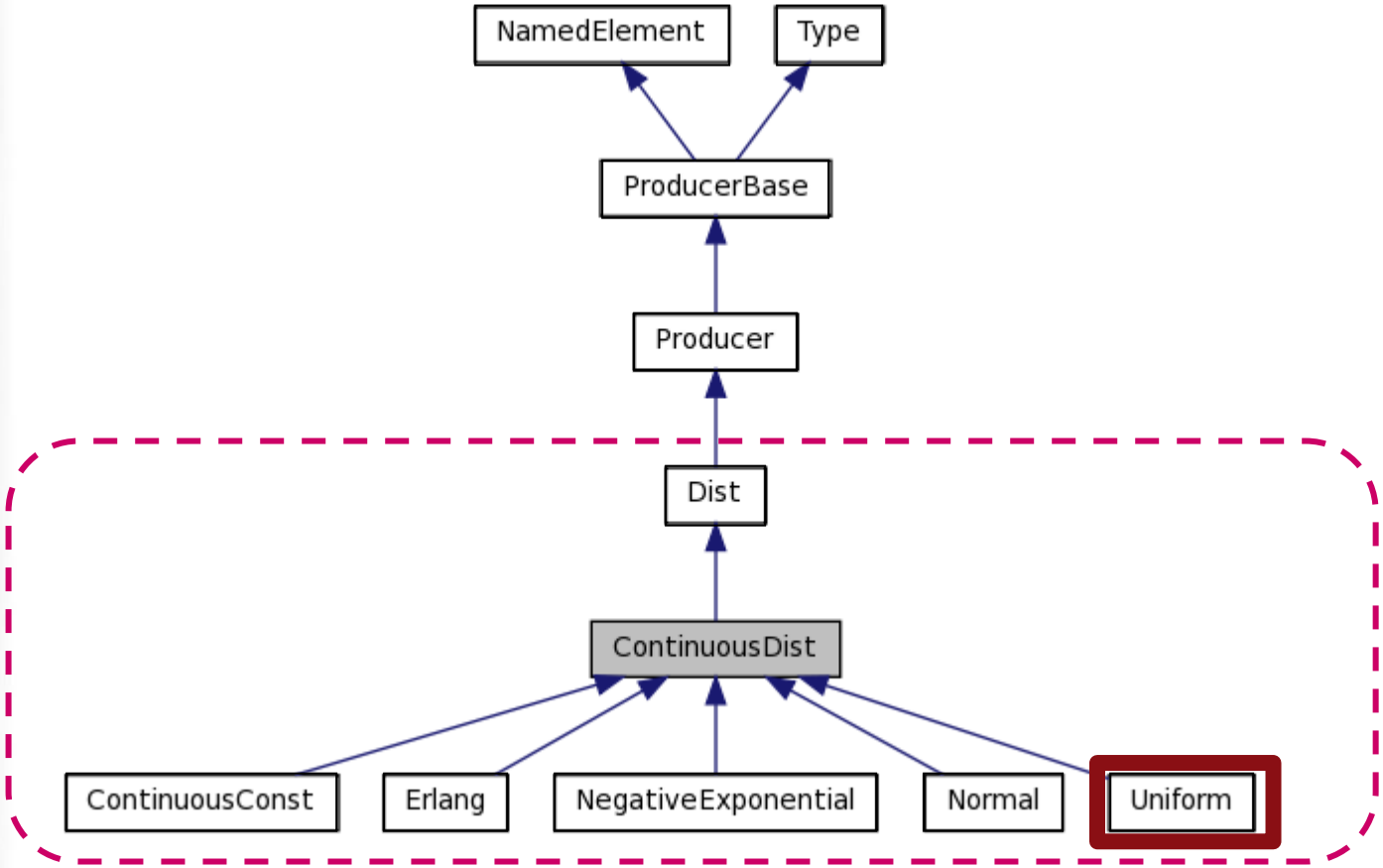
falls Zufallsgröße negativ, dann verwerfen und neuen Wert generieren solange bis positives Resultat

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	| **********

Stetige Verteilungsfunktionen



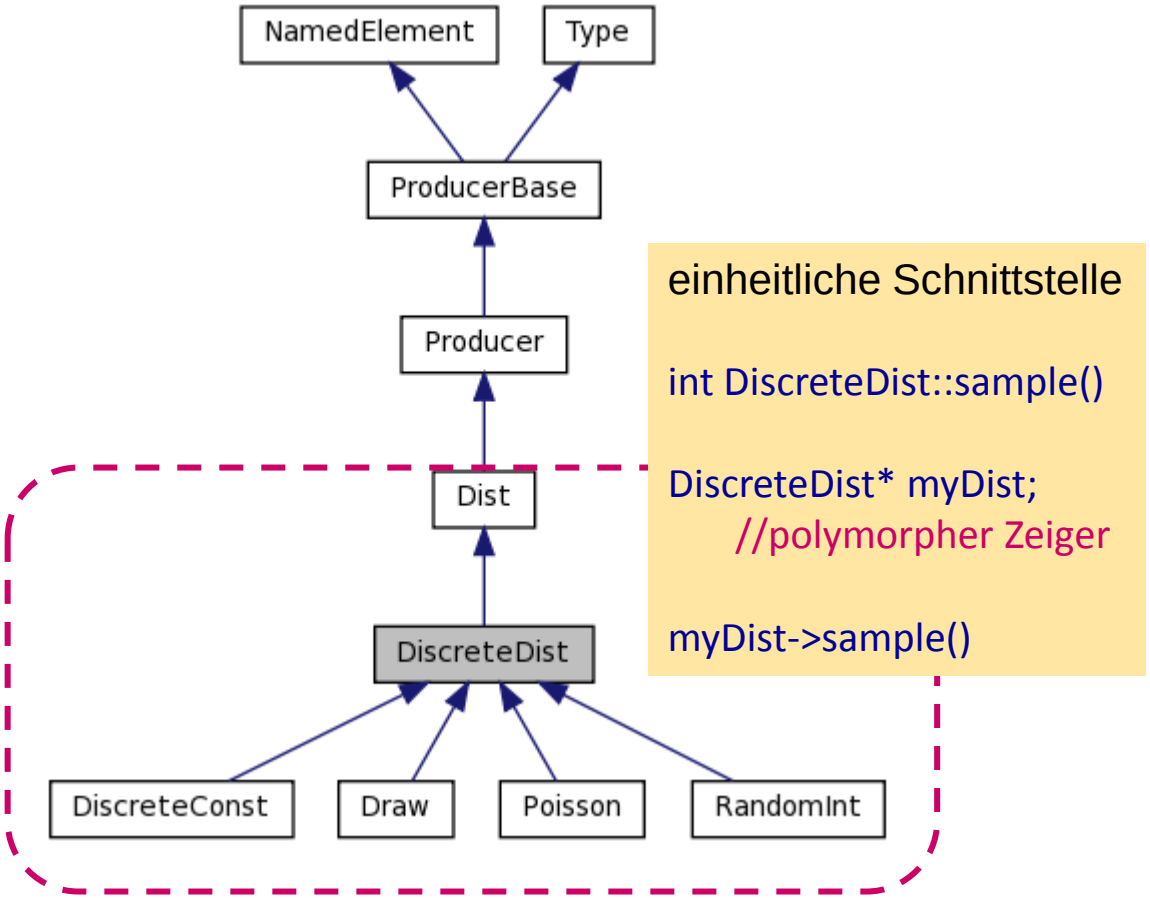
Generator für gleichverteilte reelle Pseudo-Zufallszahlen

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



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

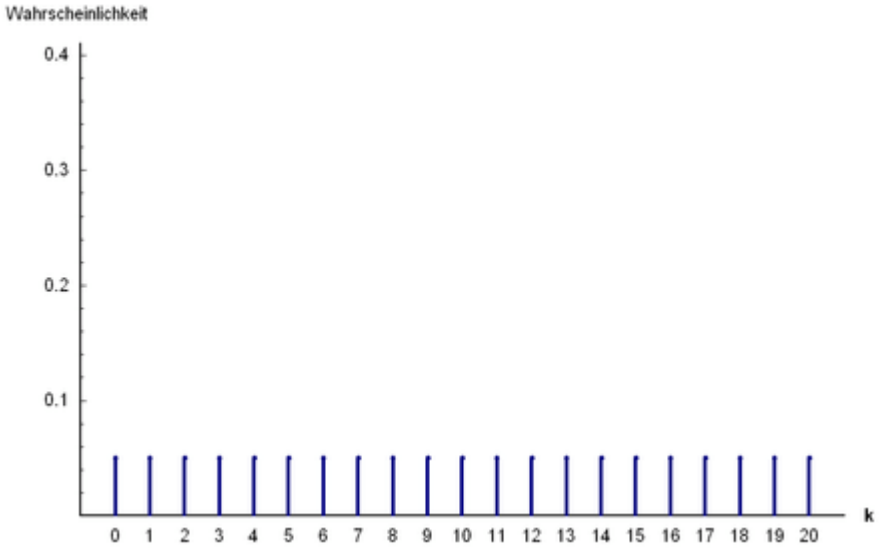
- Generator-Objekt

  Dist* wuerfel; Simulation* sim;

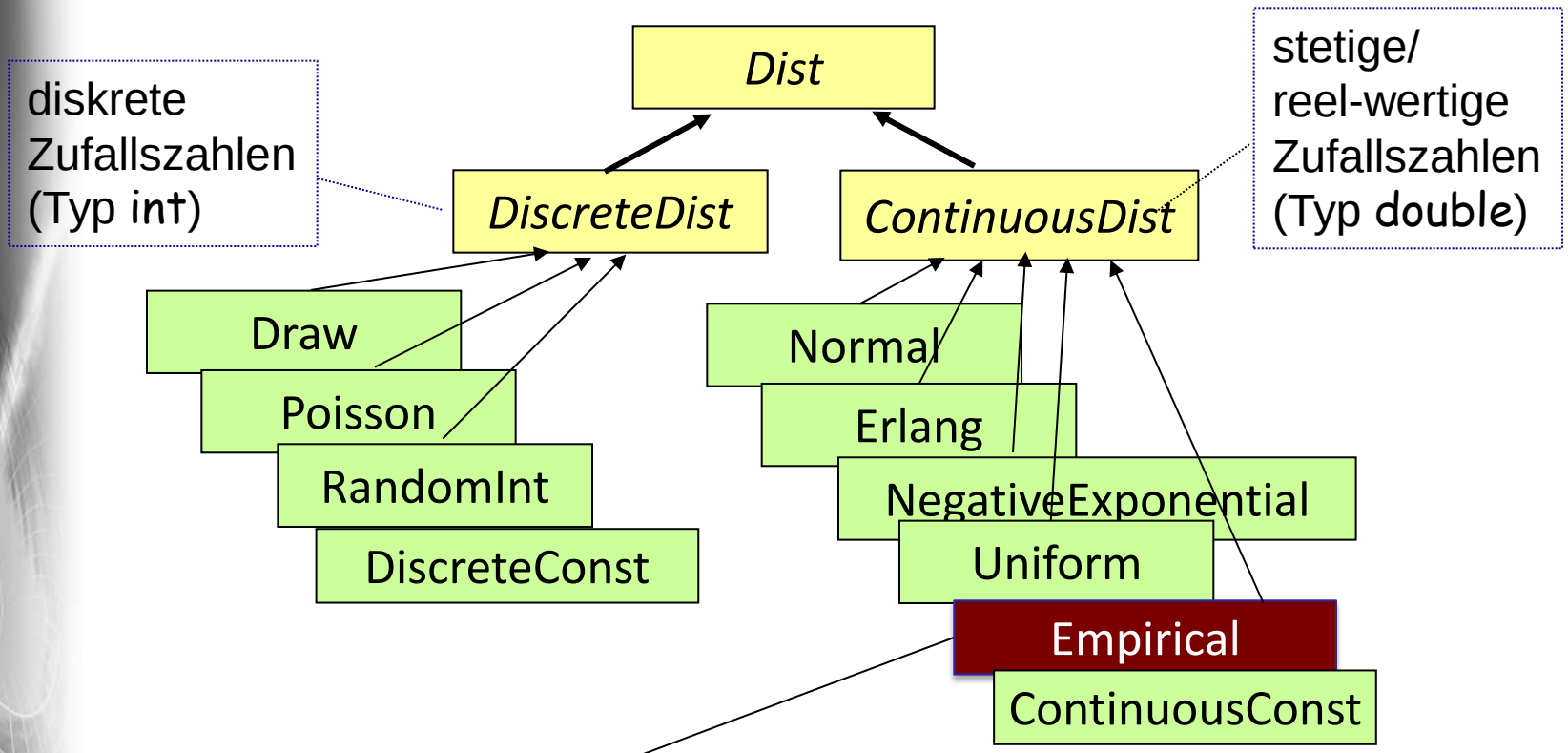
  wuerfel= **new** RandomInt (sim,“Los", 0, 20);

- Anwendung

  wuerfel->sample();



Empirische Verteilungsfunktion



Generator für empirischverteilte Pseudo-Zufallszahlen

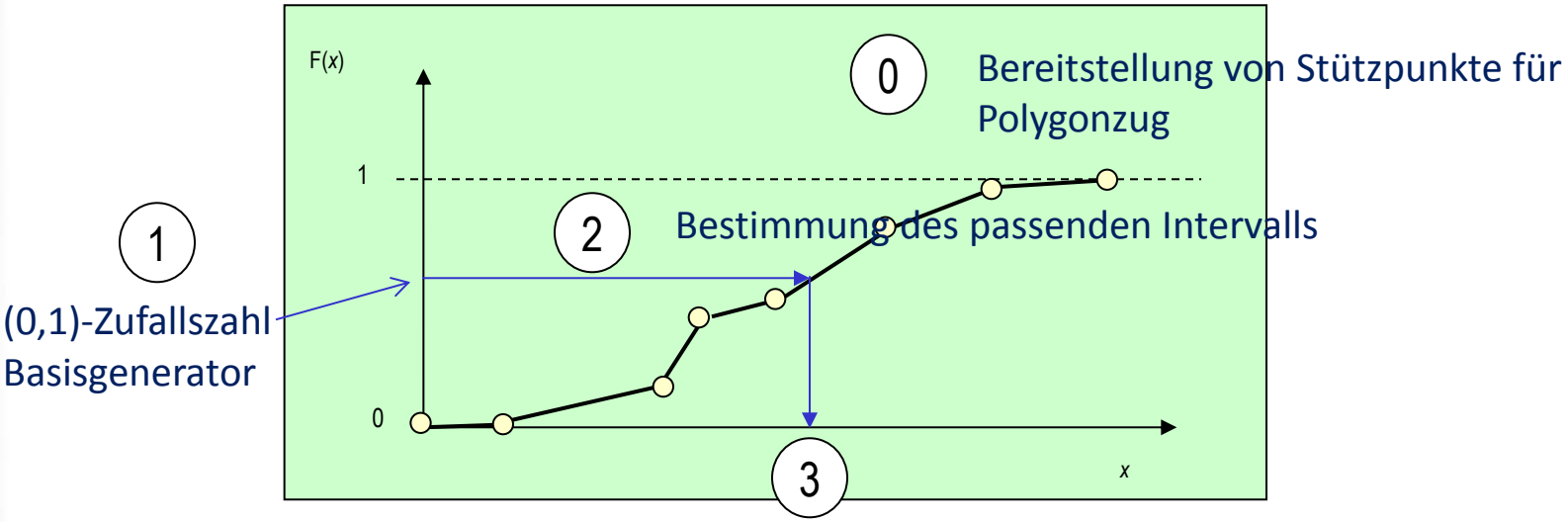
Vor.: aufgezeichnete Werte einer beobachteten Größe

a) Bestimmung der Häufigkeit der Werte in äquidistanten Intervallen

b) daraus: kumulative Häufigkeit F(x)

→ erhalten **Verteilungsfunktion**: Polygonzug über (x, F(x))-Stützpunkte

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

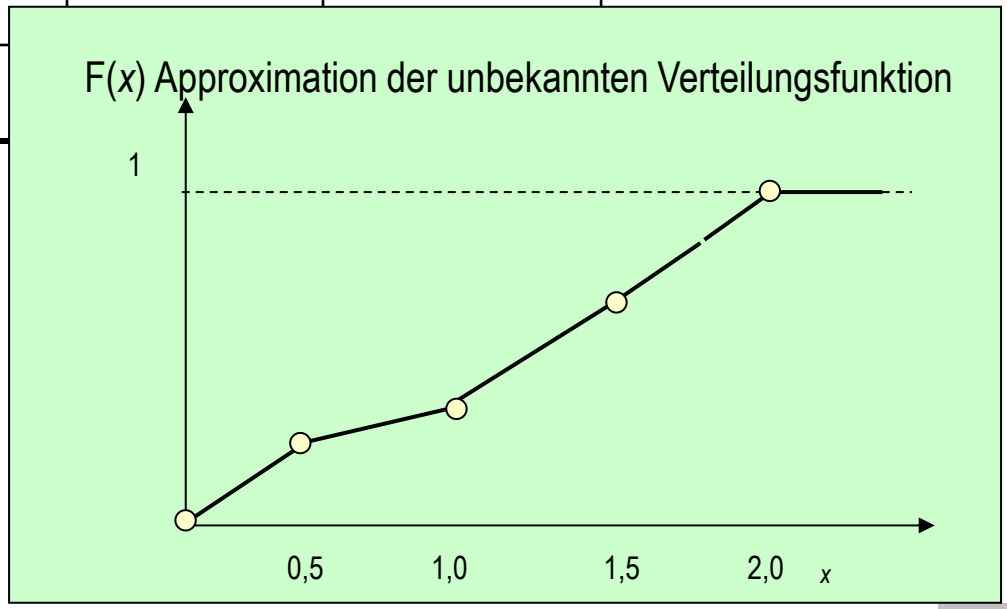


Bestimmung des x-Wertes (Zufallsgröße, die der empirischen F(x) genügt)

Empirische Verteilungen

Beispiel: Aufzeichnung von 100 Reparaturzeiten x

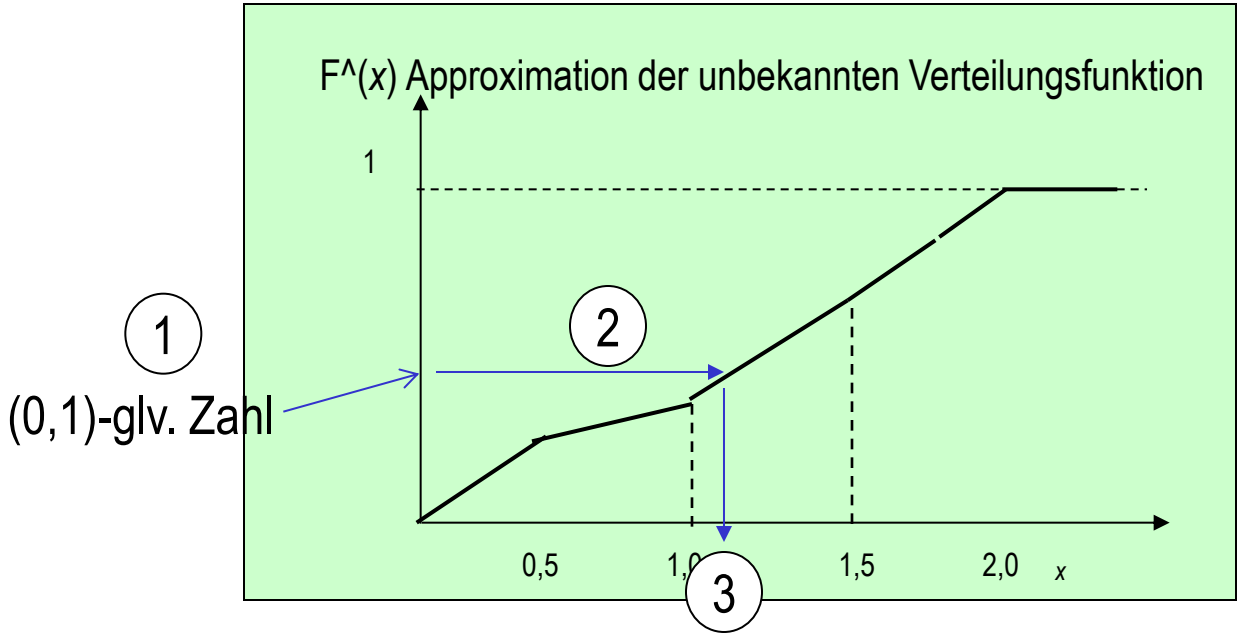
Intervall(h)	Häufigkeit	relative Häufigkeit	kummulative Häufigkeit
0 <=x<=0.5	31	0.31	0.31
0.5 < x<=1.0	10	0.10	0.41
1.0 < x<=1.5	25	0.25	0.66
1.5 < x<=2.0	34		



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
7. Lösung: Autofähre
   - Bin, Res,
   - Negexp, Normal,
   - Statistikprofile beobachteter Modellgrößen (später)

Globale Größen

Hauptprogramm wird Zeiger mit Objekten verbinden

```
/* example Ferry.cpp Ferry simulates ... */
#include <odemx/odemx.h>

odemx::base::Simulation& sim = odemx::getDefaultSimulation();

odemx::synchronization::Bin* q[2];

odemx::synchronization::Res* fload;

odemx::random::ContinuousDist* next [2];

odemx::random::ContinuousDist* crossing;

odemx::statistics::Tally *av_load;
odemx::statistics::Count *trips, *empties;

double minStayTime;
```

zwei Warte-Listen von Fahrzeugen(Token): Anlegestellen Festland und Insel

eine Warte-Liste von Fahrzeugen(Token): Ladebereich der Fähre

zwei Generatoren von Pseudo-Zufallszahlen (polymorph für stetige Vfktionen): hier später: Exponential-Verteilung

ein Generator von Pseudo-Zufallszahlen (polymorph für stetige Vfktionen): hier später: Normal-Verteilung

Statistisches Beladungsprofil (min, max, EW, SA...)

zwei Zähler

Hauptprogramm

```
main( int argc, const char* argv[]) {
        q[0] = new odemx::synchronization::Bin(sim, "mainland", 3);
        q[1] = new odemx::synchronization::Bin(sim, "island", 1);

        fload = new odemx::synchronization::Res(sim, "ferry load", 4, 4);

        next[0] = new odemx::random::NegativeExponential(sim, "mainland", 0.1);
        next[1] = new odemx::random::NegativeExponential(sim, "island", 0.1);
        crossing = new odemx::random::Normal(sim, "crossing", 7.5, 0.5);

        trips = new odemx::statistics::Count(sim, "trips");
        empties = new odemx::statistics::Count(sim, "empty trips");
        av_load = new odemx::statistics::Tally(sim, "av.load");
        minStayTime = 5.0;

        Arrival *a1 = new Arrival(0);
        a1->activateAt(380);
        Arrival *a2 = new Arrival(1);
        a2->activateAt(380);
        Ferry *f = new Ferry(fload);
        f->activateAt(7 * 60); //Start um 7.00 Uhr
        sim.run(); //Abbruch durch Ferry
}
```

Festland: 3 Fahrzeuge zur Zeit 0·0

Insel: 1 Fahrzeug zur Zeit 0·0

Maximale und aktuelle Kapazität der Fähre: 4

Pseudo-Zufallszahlengeneratoren (Exponential-, Normalverteilung)

Zähler für Fähre-Reisen (Hin und zurück)

Zähler für Leerfahrten

Beladungsprofil (min, max, EW, SA)

Ankunftsprozess-Klasse

```
class Arrival : public odemx::base::Process {
        public: int side;
        Arrival (int s): odemx::base::Process(sim, "Arrival"), side(s) {}
        int main ();
};

int Arrival::main() {
        for (;;) {
                holdFor(next[side]->sample());
                ::q[side]->give(1);
        }
        return 0;
}
```

Pseudo-Zufallszahl (exponential-verteilt) wird ermittelt

Fahrzeug-Instanz, repräsentiert durch ein Token, wird „erzeugt“ und in Bin-Objekt abgelegt

Hauptprogramm wird zwei Objekte dieser Klasse generieren, diese sorgen nach Aktivierung für zyklische (zeitversetzte) Erzeugung von Fahrzeugen auf der Festland- bzw. Inselseite

Hauptprogramm

```
main( int argc, const char* argv[]) {
        q[0] = new odemx::synchronization::Bin(sim, "mainland", 3);
        q[1] = new odemx::synchronization::Bin(sim, "island", 1);

        fload = new odemx::synchronization::Res(sim, "ferry load", 4, 4);

        next[0] = new odemx::random::NegativeExponential(sim, "mainland", 0.1);
        next[1] = new odemx::random::NegativeExponential(sim, "island", 0.1);
        crossing = new odemx::random::Normal(sim, "crossing", 7.5, 0.5);

        trips = new odemx::statistics::Count(sim, "trips");
        empties = new odemx::statistics::Count(sim, "empty trips");
        av_load = new odemx::statistics::Tally(sim, "av.load");
        minStayTime = 5.0;

        Arrival *a1 = new Arrival(0);
        a1->activateAt(6*60+20);
        Arrival *a2 = new Arrival(1);
        a2->activateAt(6*60+20);
        Ferry *f = new Ferry(fload);
        f->activateAt(7 * 60); //Start um 7.00 Uhr
        sim.run(); //Abbruch durch Ferry
}
```

zu benutzender Index von q: 0

Beginn der Fahrzeuggenerierung: 6:20 Uhr

Ferry-Klasse

```
class Ferry : public odemx::base::Process {
            odemx::synchronization::Res *myload;
    public: int main ();
            Ferry(odemx::synchronization::Res *r) : odemx::base::Process(sim, "Ferry"),
                                                    myload(r) {}
};
```

Hauptprogramm wird ein Objekt generieren, dieses sorgt für zyklische Aktionsfolge - Fahrzeugentladung, Beladung, Überfahrt

Hauptprogramm

```
main( int argc, const char* argv[]) {
        q[0] = new odemx::synchronization::Bin(sim, "mainland", 3);
        q[1] = new odemx::synchronization::Bin(sim, "island", 1);

        fload = new odemx::synchronization::Res(sim, "ferry load", 4, 4);

        next[0] = new odemx::random::NegativeExponential(sim, "mainland", 0.1);
        next[1] = new odemx::random::NegativeExponential(sim, "island", 0.1);
        crossing = new odemx::random::Normal(sim, "crossing", 7.5, 0.5);

        trips = new odemx::statistics::Count(sim, "trips");
        empties = new odemx::statistics::Count(sim, "empty trips");
        av_load = new odemx::statistics::Tally(sim, "av.load");
        minStayTime = 5.0;

        Arrival *a1 = new Arrival(0);
        a1->activateAt(6*60+20);
        Arrival *a2 = new Arrival(1);
        a2->activateAt(6*60+20);
        Ferry *f = new Ferry(fload);
        f->activateAt(7 * 60); //Start um 7.00 Uhr
        sim.run(); //Abbruch durch Ferry
}
```

Ferry-main

```
int Ferry::main() {
    MyTimer *timeout = new MyTimer (this);
    do {
        for (int side=0; side<=1; side++) {
            double entryTime = sim.getTime();
            // Entladung
                ...
            // Beladung
                ...
            // Mindestaufenthalt
                ...
            // Beladungsstatistiken
                ...
             // Fahrt
                ...
        }
        trips->update(1);  // Reise= Hin- und Rueckfahrt
     }
     while (sim.getTime() < 21*60 + 45); // nur tagsueber
    // zum Schluss noch einmal entladen
    int currentLoad = myload->getTokenLimit() - myload->getTokenNumber();
    holdFor(currentLoad*0.5);
    myload->release(currentLoad);
    sim.exitSimulation();
return 0; }
```

Ferry-main

```
int Ferry::main() {
    MyTimer *timeout =
    do {
        for (int side=0; s
            double er
            // Entladu
                ...
            // Beladu
                ...
            // Mindes
                ...
            // Beladu
                ...
             // Fahrt
                ...
        }
        trips->update(1
     }
     while (sim.getTime() < 21*60 + 45); // nur tagsueber
    // zum Schluss noch einmal entladen
    int currentLoad = myload->getTokenLimit() - myload->getTokenNumber();
    holdFor(currentLoad*0.5);
    myload->release(currentLoad);
    sim.exitSimulation();
return 0; }
```

```
for (int side=0; side<=1; side++) {
            double entryTime = sim.getTime();
            // Entladung
            int currentLoad = myload->getTokenLimit() - myload->getTokenNumber();
            holdFor(currentLoad*0.5);
            myload->release(currentLoad);
            // Beladung
            while ( myload->getTokenNumber()>0 && ::q[side]->getTokenNumber()>0 ) {

                ::q[side]->take(1);
                holdFor(0.5);
                myload->acquire(1);
            }
            // Mindestaufenthalt
            ...
```

Ferry-main

```
int Ferry::main() {
    MyTimer *timeout = new MyTimer (this);
    do {
        for (int side=0; side<=1; side++) {
            double entryTime = sim.getTime();
            // Entladung
                ...
            // Beladung
                ...
            // Mindestau
                ...
            // Beladungss
                ...
             // Fahrt
                ...
        }
        trips->update(1);  //
     }
     while (sim.getTime() < 2
    // zum Schluss noch einr
    int currentLoad = myloac
    holdFor(currentLoad*0.5
    myload->release(currentLoad);
    sim.exitSimulation();
return 0; }
```

```
// Mindestaufenthalt
            double loadTime = sim.getTime() - entryTime;
            while (loadTime < minStayTime && myload->getTokenNumber()>0) {
                timeout->setTimeout (minStayTime - loadTime);
                timeout->activate();
                if (::q[side]->take(1) != 0) { // Auto in Wartezeit angekommen
                    timeout->interrupt(); // timer reset
                    holdFor(0.5);
                    myload->acquire(1);
                }
                loadTime = sim.getTime() - entryTime;
            }
```

Ferry-main

```
int Ferry::main() {
    MyTimer *timeout = new MyTimer (this);
    do {
        for (int side=0; side<=1; side++) {
            double entryTime = sim.getTime();
            // Entladung
                ...
            // Beladung
                ...
            // Mindestaufenthalt
                ...
            // Beladungsstatistiken
                ...
             // Fahrt
                ...
        }
        trips->update(1);  // Reise= Hin- und Rueckfahrt
     }
     while (sim.getTime() < 21*60 + 45); // nur tagsueber
    // zum Schluss noch einmal entladen
    int currentLoad = myload->getTokenLimit() - myload->getTokenNumber();
    holdFor(currentLoad*0.5);
    myload->release(currentLoad);
    sim.exitSimulation();
return 0; }
```

```
// Beladungsstatistiken
   currentLoad = myload->getTokenLimit() - myload->getTokenNumber();
   av_load->update(currentLoad);
   if (currentLoad == 0) empties->update(1);
// Fahrt
   holdFor(crossing->sample());
```

MyTimer-Klasse

```
class MyTimer : public odemx::base::Process {
        odemx::base::Process *interruptible;
        double howLong;
      public:
        MyTimer(odemx::base::Process *i):
            odemx::base::Process(sim, "Timer Interrupt"), interruptible(i), howLong(0) { }
        void setTimeout(double h) { howLong = h; }

        int main() {
            for(;;) {
                holdFor(howLong);
                if (!isInterrupted()) // timer reset
                interruptible->interrupt();
                sleep();
            }
            return 0;
        }
};
```

6. ODEMx-Modul Synchronisation:

WaitQ, CondQ

- Konzept WaitQ

  Beispiel: Tankerflotte / Hafen / Raffinerie

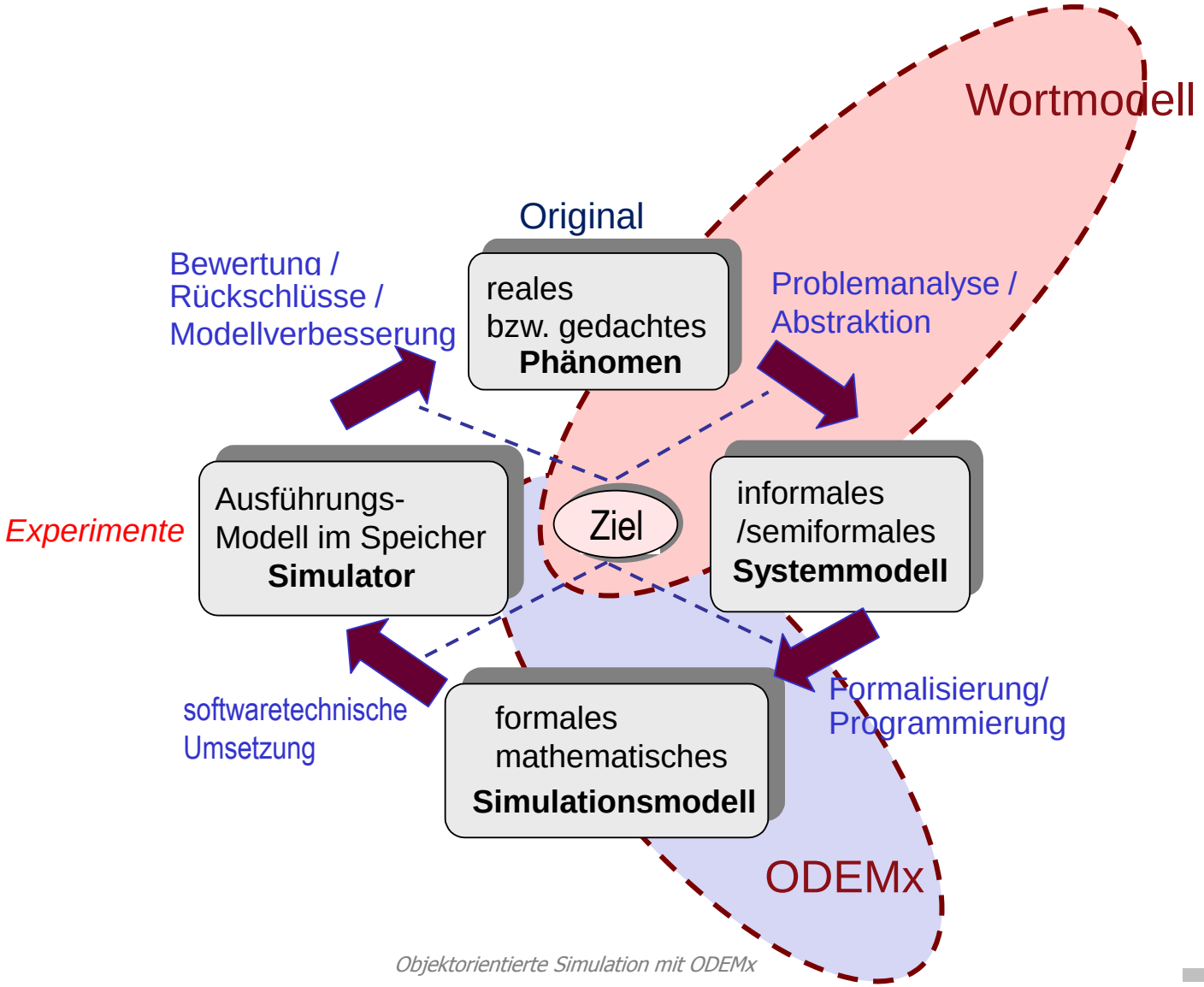
- Konzept CondQ

  Beispiel: Hafen / Schlepper / Gezeiten

- Weitere Anwendungsbeispiele für WaitQ u. CondQ

- Zusammenfassung/einheitliche Betrachtung

Erinnerung: Vorgehensweise bei der Systemsimulation



Tanker – Tank – Raffinerie: Wortmodell

1. Durchführung einer 1000h-Simulation logistischer Abläufe eines **Ölhafen**

2. **Tanker** unterschiedlichen Fassungsvermögens (gleichverteilt 15 tb, 20 tb, 25 tb)
   - treffen zufällig im Öl-Hafen ein und
   - können unter best. Bedingungen parallel entladen werden

   1 b ≈ 159 l

3. die **Zwischenankunftszeit** der Tanker ist neg.exponential verteilt
   - im Mittel soll alle 8h ein weiterer Tanker eintreffen

4. zur Entladung stehen maximal **5 Tankbehälter** bereit,
   von denen pro Tanker jeweils immer nur einer zugeordnet wird, und zwar der
   - der die längste Zeit zur Befüllung bereit war und
   - dessen freie Kapazität die Schiffsladung komplett übernehmen kann
   - steht kein solcher Tank zur Verfügung, muss der Tanker **warten**

5. die **Befüllung** des Tankbehälters (Entladung des Tankers) erfolgt
   mit einer konstanten Pumprate
   - von 1 tb/h;
   - zum Anschluss eines Tankers an einen Tank werden
   - 0,5 h als **Vorbereitungszeit**
   - benötigt.

Wortmodell (Forts.)

5. das maximale Fassungsvermögen eines jeden Tanks beträgt 70 tb

6. während der Befüllung des Tankbehälters erfolgt keine Entnahme durch die angeschlossene Raffinerie

7. die Entnahme von Öl durch die Raffinerie erfolgt
   - (nach Abschluss der Befüllung durch einen Tanker),
     sobald der Tank nur noch 20 tb oder weniger aufnehmen kann
   - mit einer konstanten Pumprate von 4 tb/h;

8. es liegt eine besondere Ausgangskonfiguration vor
   a) zwei Tanks sind leer

   b) einer ist an der Raffinerie angeschlossen und wird in 8h leer

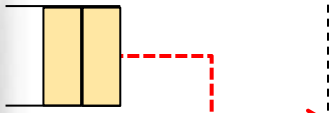
   c) zwei werden gefüllt (d.h. zwei Tanker haben angelegt), wobei
      - ein Tank in 3,5h fertig wird
        mit verbleibender Aufnahmekapazität von 25tb und
      - der andere in 12h mit verbleibender Aufnahmekapazität von 45tb

   d) der nächste Tanker wird zur Zeit 0.0 erwartet

Beispiel: Tanker – Tank – Raffinerie

Ladung: Gleichverteilung
15tb, 20tb, 25tb

Zwischenankunftszeit:
n.-exponential verteilt



(Kap 70 tb)

T5
T4
T3
T2
T1

Pumprate
1 tb/h

4 tb/h

Raffinerie

Bed.2: Ende der Tank-Befüllung
- ist Tank nahezu voll (≈70%), wird Öl zur Raffinerie abgepumpt
- Beladung wird gestoppt

Bed. 1: Tankauswahl
(1) Tank,
der die längste Zeit leer war
und
(2) dessen frei gebliebene
Kapazität die Schiffsladung
komplett übernehmen kann
(3) konstante Vorbereitungszeit: 0,5h

Ziel:
1000 h Simulation, bei besonderer Ausgangssituation:
(1) Füllstand der Tankbehälter
- zwei sind leer (70tb frei)
- einer wird in 8h leer (70tb frei)
- einer wird in 12 h mit Befüllung fertig (45tb frei)
- einer wird in 3.5h mit Befüllung fertig (25tb)

(2) erste Tankerankunft: 0.0

Beispiel zeigt typisches Problem

- Prozess-Objekte kooperieren zeitweilig miteinander
  (sogar starke Kopplung, da Rückkopplung)

  – hier: Tanker ←→ Tankbehälter
    Tankbehälter ← → Raffinerie
    (falls Raffinerie als eigenständiges Objekt)

  – Zustandsänderung des einen Objektes (Tanker-Inhalt) ist abhängig von Zustandsänderung des anderen Objektes (Tank-Inhalt) und umgekehrt

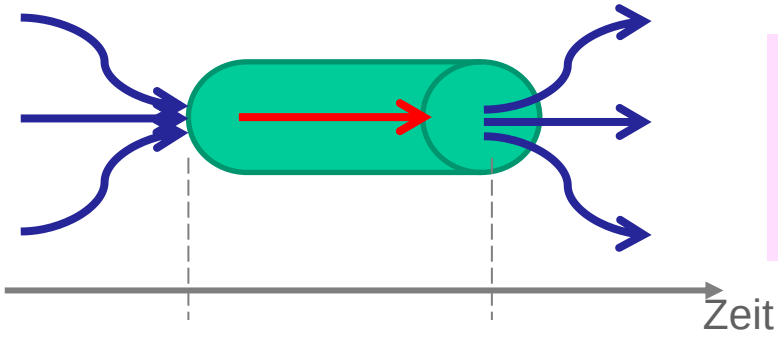
- Daraus entsteht ein allg. Synchronisationsproblem:
  bei quasiparalleler Ausführung von n Prozessen in ihrer Kooperationsphase

- Ein Prozess-Objekt kann in seiner Lebenszeit verschiedene zeitweilige Kooperationsbeziehungen eingehen

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
  falls nicht, müssen die bereits verfügbaren auf die anderen warten

  (2) Zustandsänderungen der Prozesse sind voneinander abhängig)



Entschärfung der Parallelität
der 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**

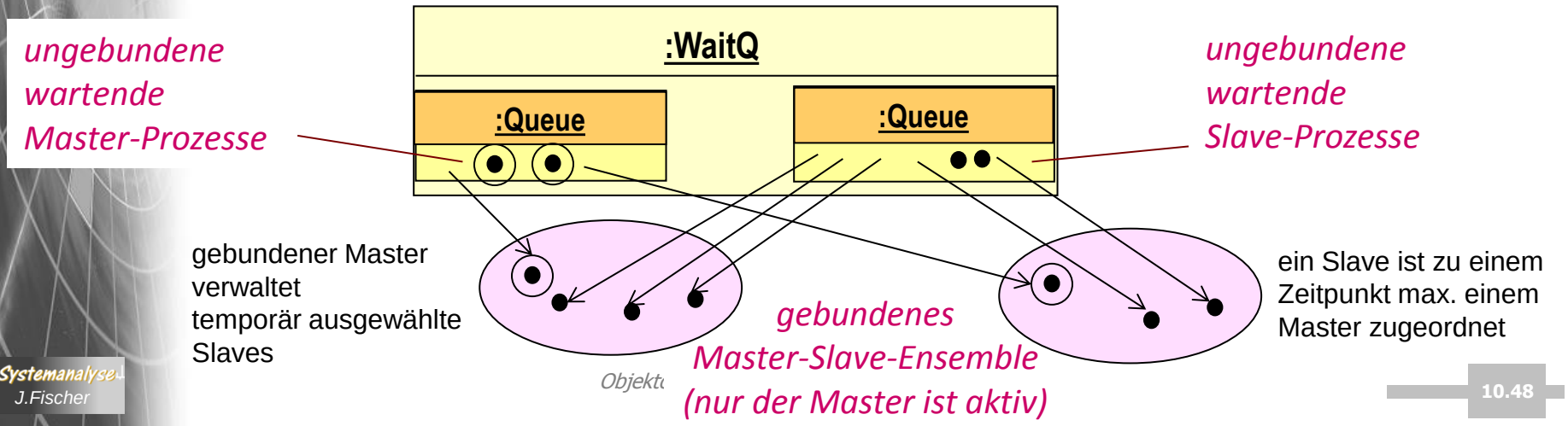
ACHTUNG: Master und Slave sind nur Rollen, die Prozesse zeitweilig spielen

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 seiner temporären Partner verbunden sind
  (benötigt entsprechende Zugriffsrechte auf seine 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 und seiner Slaves)

3. ein Master sollte über ein waitQ-Objekt die Verfügbarkeit eines Slaves mit bestimmten Eigenschaften fordern können
   - bestimmter Prozesstyp (abgeleitete Klasse)
   - 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.
                            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
```