

```

                                PSS_orientierte_DES_WSS.cpp
// Simulationsprogramm fuer ein Warteschlangensystem (WSS) mit
// Poisson'schem Ankunftsprozess und
// gleichverteilter Bedienzeit,
// prozessorientiert
//
//          ! Vorversion !
//

#include <stdio.h>           // Bibliothek fuer die Standard-Ein/Ausgabe
#include <iostream.h>       // Bibliothek fuer Stream-Ein/Ausgabe
#include <fstream.h>        // Bibliothek fuer Datei-Ein/Ausgabe
#include <math.h>           // Bibliothek mit mathematischen Standardfunktionen
#include <string>           // Bibliothek
#include <vector>           // Bibliothek
using namespace std;

const double riesig = 1E30; // Sehr große Gleitpunktkonstante

double Inverse_der_Normalverteilung(double p)
// p: Quantil der Normalverteilung, Fehler < 4.5*10E-4
{
    double q, t, x;
    if (p<=0.0)
    { cout << "Falscher Parameter in InverseDerNormalverteilung"
      << endl;
      return -1E30;
    };

    if (p>=1.0)
    { cout << "Falscher Parameter in InverseDerNormalverteilung"
      << endl;
      return 1E30;
    };

    if (p > 0.5) q = 1-p ;
    else q=p;
    t=sqrt(-log(q*q));
    x= t - ( 2.515517 + 0.802853*t + 0.010328*t*t) /
        ( 1.0 + 1.432788*t + 0.189269*t*t + 0.001308*t*t*t);
    if (p == q)
    { x= -x;
    };
    return x;
} // Ende Inverse_der_Normalverteilung

double Inverse_der_Studentverteilung(double q, int f)
// q:Quantil, f: Anzahl der Freiheitsgrade
// Naehertung, Fehler für kleine f betraechtlich,
// besonders fuer große und kleine q.
// f=1 schlecht bis unbrauchbar
// f=2, q=0.975: rel. Fehler ca. 1/4%
//      q=0.995: rel. Fehler ca. 5%
// f=3, q=0.995: rel. Fehler ca. 1%

{
    double p, t, x, x2, x3, x5, x7, x9, g1, g2, g3, g4, rf;
    int i;

    if ( q>0.5 )
        p=1.0-q;
    else p=q;
    x=Inverse_der_Normalverteilung(1.0-p);
    x2=x*x; x3 = x2*x; x5=x3*x2; x7=x5*x2; x9=x7*x2;
    g1=(x3 + x)/4.0;

```

```

PSS_orientierte_DES_WSS.cpp
g2=(5.0*x5 + 16.0*x3 + 3.0*x)/96.0;
g3=(3.0*x7 + 19.0*x5 + 17.0*x3 - 15.0*x)/384.0;
g4=(79.0*x9 + 776.0*x7 + 1482.0*x5 - 1920.0*x3 - 945.0*x)/92160.0;
rf=f;
t=x + (((g4/rf+g3)/rf)+g2)/rf+g1)/rf;
if (q<0.5) t=-t;
return t;
} // Ende Inverse_der_Studentverteilung

class Zufalls_Zahlen // Linearer Kongruenzgenerator
// für (0,1)-gleichverteilte Zufallszahlen
// und solche mit anderen Verteilungen
{ private:
double modul, faktor, Z_Zahl, y0, ZweiHoch31, nix,
EinsDurchZweiHoch31, Verhaeltnis, InvVerhaeltnis, y, ys, fy, k;
long h, i, j, g;
public:
Zufalls_Zahlen() // Konstruktor fuer Initialisierungen
{
ZweiHoch31= 1.0;
faktor = 16807.0;
for (i=1; i<=31; i++)
ZweiHoch31 = 2.0 * ZweiHoch31;
EinsDurchZweiHoch31 = 1.0 / ZweiHoch31;
modul = ZweiHoch31 - 1.0;
Verhaeltnis = modul / ZweiHoch31;
InvVerhaeltnis = ZweiHoch31 / modul;
y = 1.0 / ZweiHoch31;
Z_Zahl = 1.0 / modul;
y0 = y;
}
double gleichverteilt_0_1() // berechnet zwischen 0 und 1
// gleichverteilte Zufallszahlen
{
fy = faktor * y;
nix = modf(fy, &k);
ys = fy - k;
y = ys + k*EinsDurchZweiHoch31; // "simulierte Division"
if (y >= Verhaeltnis)
y = y - Verhaeltnis;
Z_Zahl = y * InvVerhaeltnis;
return Z_Zahl;
}
double exponentiell_mit_Parameter(double par)
{
return ( -log( gleichverteilt_0_1() ) / par );
}
double gleichverteilt(double a, double b) //berechnet zwischen a und b
// gleichverteilte Zufallszahlen
{ return (b-a)* gleichverteilt_0_1() + a;
}
};

class summierender_Zaehler // statistische Zaehler fuer diskrete Zeit
{
private:

```

```

                                PSS_orientierte_DES_WSS.cpp
double Sum, SumQuadrate, Min, Max;
int Anz;

public:

summierender_Zaehler()           // Konstruktor, Initialisierung
{  Sum = 0;
  Anz = 0;
  SumQuadrate = 0;
  Min = riesig;
  Max = -riesig;
}

void initialisiere(double s, int a)
{  Sum = s; Anz = a;
}

void addiere(double Summand)
{  Sum = Sum + Summand;
  Anz = Anz + 1;
  SumQuadrate = SumQuadrate + Summand * Summand;
  if (Summand < Min) Min =Summand;
  if (Summand > Max) Max =Summand;
}

double Summe()
{  return Sum;
}

double Minimum()
{  return Min;
}

double Maximum()
{  return Max;
}

double Anzahl()
{  return Anz;
}

double Mittelwert()
{  if (Anz>0) return Sum / Anz;
  else return 0;
}

double VI_halbe_Laenge(double Niveau)
{  double q= 1 - (1-Niveau)/2;
  return Inverse_der_Studentverteilung(q,Anz-1)
    * sqrt( (SumQuadrate - Sum*Sum/Anz)/(Anz-1)/Anz );
}

double MCI_Niveau() // Vertrauensniveau des
                  // Median-Konfidenzintervalles [Min,Max]
{
  return 1- pow(2,Anz-1);
}

};

class Haeufigkeitszaehler // statistische Zaehler fuer natuerlich-zahlige
                        // Werte, diskrete Zeit
{
private:

double Sum;
double SumQuadrate;
int Anz, Max;

```

```

int Vektorlaenge;
vector<int> H;
string Name;

public:

Haeufigkeitszaehler() // Konstruktor, Initialisierung
{
    Sum = 0;
    Anz = 0;
    SumQuadrate = 0;
}

void initialisiere(double s, int a, string name)
{
    int i;
    Sum = s; Anz = a;
    Name = name;
    Vektorlaenge = 100;
    H.reserve(Vektorlaenge+1);
    for (i=0; i<=Vektorlaenge; i++) H[i] = 0;
    Max = 0;
}

void addiere(int w)
{
    double s = double(w);
    Sum = Sum + s;
    Anz = Anz + 1;
    SumQuadrate = SumQuadrate + s * s;
    if (w > Max) Max = w;
    if (w > Vektorlaenge) {
        Vektorlaenge = w;
        H.reserve(Vektorlaenge+1);
    };
    if (w >= 0) {
        H[w]++;
    };
}

double Summe()
{
    return Sum;
}

double Anzahl()
{
    return Anz;
}

int Maximum()
{
    return Max;
}

double Mittelwert()
{
    if (Anz>0) return Sum / Anz;
    else return 0;
}

void speichern()
{
    int i;
    ofstream aus("Haeufigkeiten", ios::app);
    aus << Name << " " << Anz << " " << Max << " " ;
    for (i=0; i <= Max; i++) {
        aus << H[i] << " " ;
    };
    aus << endl;
}

};

class Entitaet

```

```

PSS_orientierte_DES_WSS.cpp
{ public:
    Entitaet(){} // Konstruktor
    double Erzeugungszeit, Zwischenzeit;
    int Art;
};
class Ereignis
{ public:
    Ereignis(){} // Konstruktor
    virtual void Ereignisroutine(Entitaet* Kunde){}
    virtual int Nummer(){return 0;}
    virtual string Name(){return " "};
    virtual void Eingang(Entitaet* Kunde){}
};
class Liste
/* Zeigerstruktur der Listen (Schlangen):

    <----->
Allgemein: Org-Element ---> Kopf ---> Element ---> ... ---> Schwanzspitze
    - - - - ->

Laenge 0:    <-----
    Org-Element
    - - ->

Laenge 1:    <-----
    Org-Element ---> Kopf    (= Schwanzspitze)
    - - - - ->

Laenge 2:    <-----
    Org-Element ---> Kopf ---> Schwanzspitze
    - - - - ->

---->: Verweis mit Naechstes; - - - ->: Verweis mit Ende
verlaengere haengt ein neues Element an die Schwanzspitze der Schlange,
auf das dann Ende zeigt.

Das Org-Element und alle anderen Listenelemente haben den Typ Element.
*/
{
private:
class Element // Typ der Elemente der Listen
{ public:
    Element *Naechstes; Element *Ende;
    double Zeit;
    Ereignis* zEreignis;
    Entitaet* zEntitaet;
};
Element * Org_Element; // erzeugt das Organisations-Element
                        // der Liste, das auch in einer
                        // leeren Liste vorhanden ist.
                        // Es ist nicht eigentlich ein
                        // Element der Liste.

```

```

public:

Liste() // Konstruktor, baut bei der Inkarnation
        // eine leere Liste zusammen
{
    Org_Element = new Element;
    Org_Element -> Naechstes = Org_Element;
    Org_Element -> Ende = Org_Element;
    Org_Element -> Zeit = riesig;
}

~Liste() // Destruktor, gibt beim Auflösen der Liste
        // den Speicher aller ihrer Element frei
{
    Element * z, * zz;
    z = Org_Element -> Naechstes;
    while ( z != Org_Element )
    {
        zz = z;
        z = z -> Naechstes;
        delete zz;
    }
    delete Org_Element;
}

void verlaengere(double t, Ereignis* zE, Entitaet* zEnt)
        // die Liste am Ende um ein Element
        // mit dem Zeiteintrag t,
        // dem Zeiger zE auf ein Ereignis,
        // dem Zeiger zEnt auf eine entitaet
{
    Element *z = new Element;
    Org_Element -> Ende -> Naechstes = z;
    z -> Naechstes = Org_Element;
    Org_Element -> Ende = z;
    z -> Zeit = t;
    z -> zEreignis = zE;
    z -> zEntitaet = zEnt;
}

void drucke (string Ueberschrift) // und die Attribute der Elemente
{
    Element *z = Org_Element -> Naechstes;
    cout << Ueberschrift << endl;
    while (z != Org_Element)
    {
        cout << "          Zeit= " << z -> Zeit ;
        if ( z -> zEreignis != NULL)
            cout <<" " << z -> zEreignis -> Name();
        if ( z -> zEntitaet != NULL)
            cout <<" Entitaet " << z -> zEntitaet -> Art;
        cout << endl;
        z = z -> Naechstes;
    }
}

int leer() // Liste leer?
{
    return Org_Element -> Naechstes == Org_Element;
}

double Zeit_am_Kopf () // Zeiteintrag des Elementes am Kopf
{
    return Org_Element -> Naechstes -> Zeit;
}

Ereignis* zEreignis_am_Kopf() // Ereignisart des Elementes am Kopf
{
    return Org_Element -> Naechstes ->zEreignis;
}

Entitaet* zEntitaet_am_Kopf() // Entitaet des Elementes am Kopf
{
    return Org_Element -> Naechstes ->zEntitaet;
}

void entferne_Kopf() // verkuerze die Liste um das Element am Kopf

```

```

                                PSS_orientierte_DES_WSS.cpp
{   Element * weg;
    if ( Org_Element -> Naechstes != Org_Element)           // nicht leer
    {   if ( Org_Element -> Naechstes == Org_Element -> Ende ) // Laenge 1
        {   Org_Element -> Ende = Org_Element;
            }
        weg = Org_Element -> Naechstes;
        Org_Element -> Naechstes = Org_Element -> Naechstes -> Naechstes;
        delete weg;
    }
}

void ordne_Ereignis_ein(double wann, Ereignis* zE, Entitaet* zEnt)
                                // ordnet neues Element
                                // gemäß seines Zeit-Attributes in die
                                // Liste ein. Weitere Attribute:
                                // Ereigniszeiger, Entitaetenzeiger.

{   Element * neu = new Element; // Speicher für das neue Element
    Element * davor, * z;
    neu -> Zeit = wann;
    neu -> zEreignis= zE;
    neu -> zEntitaet = zEnt;
    // neues Ereignis zeitlich in die E-liste einordnen:
    davor = Org_Element;
    z = davor -> Naechstes;
    while (z -> Zeit <= wann)
    {   davor = z;
        z = z -> Naechstes;
    }
    davor -> Naechstes = neu;
    neu -> Naechstes = z;
    if (z == Org_Element) // das neue Ereignis ist das derzeit Späteste
    {   Org_Element -> Ende = neu;
    }
}

void leeren()                    // Liste leeren
{
    while ( Org_Element -> Naechstes != Org_Element )
        entferne_Kopf();
}
}; // Ende der Klasse Liste

class Simulation: public Liste
                                // fuer den Grundalgorithmus der Simulation.
                                // Die Spezifikation des konkreten Modells
                                // erfolgt in einer abgeleiteten Klasse
{
public:

    Liste E_Liste;           // Ereignisliste

    double Endzeit;

    double Uhr, * zUhr;      // fuer die Modellzeit
    int Spur;                // Spur aufzeichnen und ausgeben?

    Simulation(int sp)
    {
        Spur = sp;
        zUhr = &Uhr;
    }

    void Spurtritt()
    {   cout << "Spur-----"
        << endl << "Uhr=" << Uhr << endl;
    }
}

```

```

virtual void initialisiere_Simulator()
{ E_Liste.leeren();
}

virtual void setze_Simulationsende(double Zeitpunkt)
{ Endzeit = Zeitpunkt;
}

virtual void plane_Ereignis(double wann, Ereignis* zE, Entitaet * zEnt)
{ E_Liste.ordne_Ereignis_ein(wann, zE, zEnt);
  if (Spur >0)
  { Spurtritt();
    E_Liste.drucke("  E-Liste, neuer Eintrag: ");
  }
}

virtual void simuliere() // zentraler Simulationsalgorithmus
{ Ereignis *zE;
  Entitaet *zEnt;
  double E_Zeit;
  while ( !E_Liste.leer() )
  { E_Zeit = E_Liste.Zeit_am_Kopf();
    zE = E_Liste.zEreignis_am_Kopf();
    zEnt = E_Liste.zEntitaet_am_Kopf();
    if (E_Zeit <= Endzeit)
    { E_Liste.entferne_Kopf();
      Uhr = E_Zeit;
      zE -> Ereignisroutine(zEnt);
    } else
      break;
  }
};
};

class integrierender_Zaehler // statistische Zaehler fuer stetige Zeit
{
private:
  double Ordinate, letzte_Aenderungszeit, Integral, Anfangszeit,
    * aktuelleZeit;

public:
  integrierender_Zaehler() // Konstruktor, Initialisierung
  { Ordinate = 0;
    letzte_Aenderungszeit = 0;
    Integral = 0;
    Anfangszeit = 0;
  }

  void akkumuliere(double Veraenderung)
  {
    Integral = Integral +
      ( (* aktuelleZeit) - letzte_Aenderungszeit) * Ordinate;
    Ordinate = Ordinate + Veraenderung;
    letzte_Aenderungszeit = (* aktuelleZeit);
  }

  void initialisiere(double Anfangsordinate, double Anfang, double * Z)
  { Ordinate = Anfangsordinate;
    letzte_Aenderungszeit = Anfang;
    Integral = 0;
    Anfangszeit = Anfang;
    aktuelleZeit = Z;
  }

  double Wert()

```

```

{ return Integral;
}

double Mittelwert()
{ if (letzte_Aenderungszeit - Anfangszeit > 0 )
    return Integral / (letzte_Aenderungszeit - Anfangszeit);
  else return 0;
}
};

//=====
//
//          MODELL
//
//=====

// Klassenstruktur:
//
// Liste <- Simulation
//
// Innerhalb der Klasse Modell:
//
//     Ereignis <- Quelle
//
//     Ereignis <- WSS
//
//     Ereignis <- Senke
//
//     Ereignis <- Simulationsende
//
// Experiment
//
// Instanzierung von Simulation, Modell und Experiment ausserhalb

class Modell
{ public:

    Simulation * Sim;

// Deklaration eines Zufallszahlen-Objektes
    Zufalls_Zahlen ZZahl;

// Deklarationen von Konstanten des Modells:

// Zustaende des Bedieners:
    static const int frei = 0, belegt = 1;

// Arten des Bedienungsbeginns:
    static const int sofort = 1, spaeter = 0;

// Experimentplanung:
    double Endzeitpunkt;

// Modellabhaengige Klassen fuer Stationen und Ereignisse:

class Quelle: public Ereignis
// Modellaspekt: Erzeugung von Kunden gemaess Poissonprozess
// Simulationsaspekt: Station, ein Prozess mit einer Ereignisroutine
{ public:

```

```

// Umgebung:
Modell* Mod;
Simulation* Sim;

// Attribute:
string Stationsname; // des Quellprozesses
double lambda; // Rate der Kundenerzeugung
Ereignis* Ziel; // Station sind von Ereignis abgeleitet
Ereignis* zE_Quelle; // Zeiger auf diese Station

string Name()
{ return Stationsname;
}

// Statistikvariable:
int Anzahl_erzeugter_Kunden;

// Initialisierung:
Quelle(Simulation* S, Modell* M, string Na)
{ Stationsname=Na;
  Sim = S;
  Mod = M;
  zE_Quelle = ((Ereignis*) this);
}

void Stationsattribute(double lam, Ereignis* Z)
{ lambda = lam;
  Ziel = Z;
}

void initialisiere_Stationszustand()
{ // Anfangszustand:
}

void initialisiere_Statistik()
{ Anzahl_erzeugter_Kunden = 0;
}

// Prozess:

/*do*/ void Ereignisroutine(Entitaet* zKunde) {

  Entitaet* neuer_Kunde = new Entitaet;
  neuer_Kunde -> Erzeugungszeit = Sim->Uhr;
  neuer_Kunde -> Art = Anzahl_erzeugter_Kunden +1;

  // Statistik
  Anzahl_erzeugter_Kunden ++;

  // Spuraufzeichnung:
  if (Sim->Spur>0)
  { Spurdaten();
  };

  // senden des neuen Kunden zum Nachfolgeprozess:
  Ziel -> Eingang(neuer_Kunde);

  // warten auf neue Kundenerzeugung:
  Sim->plane_Ereignis(Sim->Uhr+
  Mod->ZZahl.exponentiell_mit_Parameter(lambda),
  zE_Quelle, NULL);
  // Zeitpunkt: eine Zwischenankunftszeit von jetzt an

/*while(true)*/ }

// Fuer die Modellbeschreibung:
void beschreibe_Station()
{ cout << "Station " << Stationsname

```

```

        PSS_orientierte_DES_WSS.cpp
        << ": Poissonscher Ankunftsprozess mit Rate lambda= "
        << lambda << endl
        << "ZielstationR, ---> " << Ziel->Name() << endl;
    }

// Fuer den Ergebnisbericht:
void berichte()
{   cout << "Station " << Stationsname << ", Anzahl erzeugter Kunden= "
    << Anzahl_erzeugter_Kunden << endl ;
}

// Fuer die Spuraufzeichnung:
void Spurdaten()
{   Sim->Spurtritt();
    cout << "   Station " << Stationsname <<
        ", " << Anzahl_erzeugter_Kunden << "-ter Kunde" << endl ;
}

};

class WSS: public Ereignis
// Modellaspekt: Warteschlangensystem
//           mit einem Bediener mit zwischen a und b gleichverteilter
//           Bediendauer und FIFO-Strategie
// Simulationsaspekt: Station mit zwei Prozessen,
//           einer Ereignisroutine, einem Eingang fuer Kunden,
//           und einer weiteren spontanen Aktivitaetsroutine

{   public:

// Umgebung:
    Modell* Mod;
    Simulation* Sim;

// Attribute:
    string Stationsname;
    double a,b;           // Verteilungsparameter
    Ereignis* Ziel;       // Stationen sind von der Klasse Ereignis abgeleitet
    Ereignis* zE_WSS;     // Zeiger auf diese Station

    string Name()
    {   return Stationsname;
    }

// Zustand:
    int N;                // Anzahl der Kunden im Warteschlangensystem (WSS)
    int Q;                // Anzahl der wartenden Kunden
    int Bediener;        // frei oder belegt

    Liste Warteschlange;
        // fuer die ankommenden Kunden mit ihren Ankunftszeiten

// Statistische Zaehler:
    integrierender_Zaehler Kundenzahl,           // Anzahl im WSS
        Wartende,                               // Anzahl im Warteraum
        Bedienerbelegung;                       // Anzahl im Bediener

    summierender_Zaehler Wartezeiten,           // auf Bedienung
        Bedienungsstart;                       // sofort (1) oder spaeter (0),
        // fuer die Wahrscheinlichkeit
        // nicht warten zu muessen

    Haeufigkeitszaehler WS_Belegung;           // im Warteraum

// Initialisierung:

```

```

                                PSS_orientierte_DES_WSS.cpp
WSS(Simulation* S, Modell* M, string Na)
{
    Stationsname=Na;
    Sim = S;
    Mod = M;
    zE_WSS = ((Ereignis*) this);
}

void Stationsattribute(double a0, double b0, Ereignis* Z)
{
    a = a0; b = b0;
    Ziel = Z;
}

void initialisiere_Stationszustand(int N0, int Q0, int Bediener0)
{
    // Anfangszustand:
    Bediener = Bediener0; // Bediener anfaeniglich belegt (1) oder frei (0)
    N = N0; // anfaenliche Anzahl im WSS
    Q = Q0; // anfaenliche Anzahl in der Warteschlange (WS)
    Warteschlange.leeren(); // wenn Q0 nicht 0, dann hier Kunden einfuegen!
}

void initialisiere_Statistik()
{
    // Statistische Zähler:
    Kundenzahl.initialisiere(0,0,Sim->zUhr);
    Wartende.initialisiere(0,0,Sim->zUhr);
    Bedienerbelegung.initialisiere(0,0,Sim->zUhr);
    Wartezeiten.initialisiere(0,0);
    Bedienungsstart.initialisiere(0,0);
    WS_Belegung.initialisiere(0,0,"WSS");
}

// Prozess 1, Kundenankuenfte, eine Aktivitaetsroutine:

/*do*/          void Eingang(Entitaet* Kunde) {

    // warte auf Kunden
    Kunde -> Zwischenzeit = Sim -> Uhr;

    // Neuer Zustand:
    N++;
    Q++;
    Warteschlange.verlaengere(Sim->Uhr,NULL,Kunde);

    // Statistik:
    Kundenzahl.akkumuliere(+1);
    Wartende.akkumuliere(+1);
    WS_Belegung.addiere(Q-1);
    if (Bediener == frei)
    { Bedienungsstart.addiere(sofort);
    } else
    { Bedienungsstart.addiere(spaeter);
    };

    // Spuraufzeichnung:
    if (Sim -> Spur>0)
    { Spurdatenl();
    };

    if (Bediener == frei)
    { bediene();
    };

/*while(true)*/ }

// Prozess 2, Bedienungen, eine Aktivitaetsroutine und eine Ereignisroutine:

/*do*/          void bediene() {

```

```

PSS_orientierte_DES_WSS.cpp
Entitaet* Kunde;

// warte auf Kunden
Kunde = Warteschlange.zEntitaet_am_Kopf();

// Neuer Zustand:
Q = Q-1; // ein Kunde weniger im Warteraum
Bediener = belegt;

// Statistik:
Wartende.akkumuliere(-1);
Bedienerbelegung.akkumuliere(+1);
Wartezeiten.addiere(Sim->Uhr - Warteschlange.Zeit_am_Kopf() );

// noch Neuer Zustand:
Warteschlange.entferne_Kopf(); // Kunde aus der Warteschlange

// Spuraufzeichnung:
if (Sim->Spur>0)
{ Spurdaten2();
};

// warte auf Bedienungsende:
Sim->plane_Ereignis(Sim->Uhr+ Mod->ZZahl.gleichverteilt(a,b),
zE_WSS,Kunde);
// Bediendauer gleichverteilt zwischen a und b

}

void Ereignisroutine(Entitaet* Kunde) {

// Statistik:
Bedienerbelegung.akkumuliere(-1);
Kundenzahl.akkumuliere(-1);

// Neuer Zustand:
N = N-1;
if (N == 0)
{ Bediener = frei;
};

// Spuraufzeichnung:
if (Sim->Spur>0)
{ Spurdaten3();
};

Ziel -> Eingang(Kunde);

if (N>0)
bediene();

/*while(true)*/ }

// Fuer die Modellbeschreibung:
void beschreibe_Station()
{ cout << "Station " << Stationsname << endl;
cout << " Bedienzeiten gleichverteilt zwischen a="
<< a << " und b=" << b << endl
<< "ZielstationR, ---> " << Ziel->Name() << endl;
cout << " Anfangszustand: " << endl ;
cout << " Modellzeit, Uhr= " << Sim -> Uhr << endl ;
cout << " Anzahl Kunden im System, N= " << N << endl ;
cout << " Anzahl Kunden in der Warteschlange, Q= " << Q << endl ;
cout << " Anzahl Kunden in der Bedienung= " << Bediener << endl ;
cout << endl;
}

```

```

PSS_orientierte_DES_WSS.cpp
// Fuer den Ergebnisbericht:
void berichte()
{
    cout << "Station " << Stationsname << endl;
    cout << "    Mittlere Kundenzahl in der Station= "
        << Kundenzahl.Mittelwert() << endl ;
    cout << "    Mittlere Kundenzahl in der Warteschlange= "
        << Wartende.Mittelwert() << endl ;
    cout << "    Mittlere Kundenzahl im Bediener= "
        << Bedienerbelegung.Mittelwert() << endl ;
    cout << "    Mittlere Wartezeit= "
        << Wartezeiten.Mittelwert() << endl ;
    cout << "    Anzahl der Kunden, die die Warteschlange \n"
        << "    wieder verlassen haben= "
        << Wartezeiten.Anzahl() << endl ;
    cout << "    Mittlerer Anteil der Kunden, die nicht auf Bedienung \n"
        << "    warten mußten= "
        << Bedienungsstart.Mittelwert() << endl ;
    cout << "    Anzahl der Kunden, deren Bedienung begann= "
        << Bedienungsstart.Anzahl() << endl ;
    cout << endl;
}

// Fuer die Spuraufzeichnung:
void Spurdaten1()
{
    Sim->Spurtritt();
    cout << "    Station " << Stationsname
        << " nach einer Kundenankunft " << endl ;
    cout << "        N= " << N << "        Q= "
        << Q << "        Bediener= " << Bediener << endl;
    Warteschlange.drucke("    WS nach spiele E: ");
}

void Spurdaten2()
{
    Sim->Spurtritt();
    cout << "    Station " << Stationsname
        << " nach Bedinungsbeginn " << endl ;
    cout << "        N= " << N << "        Q= "
        << Q << "        Bediener= " << Bediener << endl;
    Warteschlange.drucke("    WS nach spiele E: ");
    cout << "    Anzahl der Wartezeiten "
        << Wartezeiten.Anzahl() << endl;
    cout << "    Summe der Wartezeiten "
        << Wartezeiten.Summe() << endl;
    cout << "    Anzahl der Bedienungsbeginne "
        << Bedienungsstart.Anzahl() << endl ;
    cout << "    Anzahl der Kunden, die nicht warten mußten "
        << Bedienungsstart.Summe() << endl;
    cout << "    Kundenzahl integriert " << Kundenzahl.Wert() << endl;
    cout << "    Wartende integriert " << Wartende.Wert() << endl;
    cout << "    Bedienerbelegung integriert "
        << Bedienerbelegung.Wert() << endl;
}

void Spurdaten3()
{
    Sim->Spurtritt();
    cout << "    Station " << Stationsname
        << " nach Bedienunsende " << endl ;
    cout << "        N= " << N << "        Q= "
        << Q << "        Bediener= " << Bediener << endl;
}
}; // Ende der Klasse WSS

class Senke: public Ereignis
// Modellaspekt: Senke
// Simulationsaspekt: Station, ein Prozess mit einer spontanen
// Aktivitaetsroutine, keine Ereignisroutine

```

```

{
    public:

// Umgebung:
    Modell* Mod;
    Simulation* Sim;

// Attribute:
    string Stationsname;
                                // keine Ziel-Station

    string Name()
    {return Stationsname;
    }

// Kein Zustand

// Statistikvariable, Statistische Zaehler:
    summierender_Zaehler Verweilzeiten;           // im System

// Initialisierung:
    Senke(Simulation* S, Modell* M, string Na)
    {    Stationsname= Na;
        Sim = S;
        Mod = M;
    }

    void initialisiere_Statistik()
    {    Verweilzeiten.initialisiere(0,0);
    }

// Prozess:

    /*do*/
                                void Eingang(Entitaet* Kunde) {

        // warten auf Kunden
        // Statistik:
        Verweilzeiten.addiere(Sim->Uhr - Kunde->Erzeugungszeit);
        // vernichte Kundendaten:
        delete(Kunde);
        // Spuraufzeichnung:
        if (Sim -> Spur>0)
        {    Spurdaten();
        }

    /*while(true)*/
                                }

// Fuer die Modellbeschreibung:
    void beschreibe_Station()
    {
        cout << "Station " << Stationsname << endl;
    }

// Fuer den Ergebnisbericht:
    void berichte()
    {
        cout << "Station " << Stationsname << endl;
        cout << "    Mittlere Verweilzeit im gesamten System= "
            << Verweilzeiten.Mittelwert() << endl ;
    }

// Fuer die Spuraufzeichnung:
    void Spurdaten()
    {
        Sim->Spurtritt();
        cout << "    Station " << Stationsname << ": " <<
            Verweilzeiten.Anzahl() << "-ter Kunde " << endl;
    }
}

```

```

}; // Ende der Klasse Senke

class Simulationsende: public Ereignis
{
public:

string E_Name;

Ereignis *zQuelle, *zE_Simulationsende;
WSS* zWSS;
Senke* zSenke;

Simulationsende(WSS* w, Senke* s)
{
    zE_Simulationsende=( (Ereignis*) this);

    zWSS = w;
    zSenke = s;
    E_Name= "Simulationsende";
}

string Name()
{return E_Name;
}

void Ereignisroutine()
{
    // alle statistischen Zaehler mit stetiger Zeit
    // bis zur aktuellen Uhrzeit akkumulieren:
    zWSS -> Kundenzahl.akkumuliere(0);
    zWSS -> Wartende.akkumuliere(0);
    zWSS -> Bedienerbelegung.akkumuliere(0);
    zWSS -> WS_Belegung.speichern();
}
}; // Ende der Klasse Simulationsende

void Bericht()
{
    // zweiter Teil des Berichtes:

    cout << endl;
    cout <<
        "*****" << endl ;
    cout <<
        "*                Bericht                *" << endl ;
    cout <<
        "*****" << endl ;

    zQuelle -> berichte();
    zWSS -> berichte();
    zSenke -> berichte();
    cout <<
        "*****" << endl ;
}

void setze_Attribute()
{
    zQuelle -> Stationsattribute( 0.25, zWSS);
        // Stationsname
        // Parameter des Poissonprozesses der Zwischenankunftszeiten
        // Zielstation

    zWSS -> Stationsattribute(2, 4, zSenke);
        // Stationsname und
        // 2 Parameter der Bediendauern, gleichverteilt zwischen a und b
}

```

```

        PSS_orientierte_DES_WSS.cpp
    // Zielstation
}

void initialisiere_Modellzustand(double t_Anfang, double t_Ende)
{
    zWSS -> initialisiere_Stationszustand(0, 0, frei);
        // Bedeutung der Parameter:
        // WSS anfangs leer
        // Warteschlange anfangs leer
        // Bediener anfangs frei

    Sim -> Uhr = t_Anfang;

    // plane erste Ankunft:
    Sim->plane_Ereignis( Sim->Uhr,
        zE_Quelle, NULL);

    // plane Simulationsende:
    Endzeitpunkt = t_Ende;
    Sim -> plane_Ereignis(Endzeitpunkt, zE_Simulationsende, NULL);
}

void initialisiere_Statistik()
{
    zQuelle -> initialisiere_Statistik();
    zWSS -> initialisiere_Statistik();
    zSenke -> initialisiere_Statistik();
}

void Modellbeschreibung()
{
    // Erster Teil des Berichtes, das Modell:

    cout <<
        "*****" << endl ;
    cout <<
        "*"           Warteschlangenmodell           "*" << endl ;
    cout <<
        "*****" << endl ;
    zQuelle -> beschreibe_Station();
    zWSS -> beschreibe_Station();
    zSenke -> beschreibe_Station();

    cout << endl << "Netzgraph:" << endl<< endl<<
        "Quelle ---> WSS ---> Senke" << endl << endl;

    cout <<
        "*****" << endl ;
}

// Deklarationen von Zeigern auf Stationen und Ereignisse (Klassen):
Ereignis *zE_Quelle, *zE_WSS, *zE_Senke, *zE_Simulationsende;
    // An manchen Stellen werden Zeiger einheitlichen Typs,
    // naemlich Ereignis, benoetigt, zum Beispiel in Ereignissen.
    // An anderen Stellen Zeiger auf die speziellen Stationen bzw.
    // Ereignisse:
Quelle* zQuelle;
WSS* zWSS;
Senke* zSenke;
Simulationsende* zSimulationsende;

Modell(Simulation * S)    // Konstruktor mit Instanzierung der Stationen
                        // und der Ereignisse, ferner
                        // Initialisierung der Zeiger

{
    Sim = S;
    zWSS = new WSS(S,this,"WSS");
    zE_WSS = ((Ereignis*) zWSS);
    zQuelle = new Quelle(S,this,"Quelle");
}

```

```

        PSS_orientierte_DES_WSS.cpp
        zE_Quelle = ((Ereignis*) zQuelle);
        zSenke = new Senke(S,this,"Senke");
        zE_Senke = ((Ereignis*) zSenke);
        zSimulationsende = new Simulationsende(zWSS,zSenke);
        zE_Simulationsende = ((Ereignis*) zSimulationsende);
    }

    ~Modell()                // Destruktor
    { delete zWSS;
      delete zQuelle;
      delete zSenke;
      delete zSimulationsende;
    }
}; // Ende der Klasse Modell

class Experiment
{
public:
    Experiment(){}

    // Simulationsexperimente:

    void eineSEH () // Simulation mit endlichem Horizont
    { double Simulationsdauer;
      int Spur;

      cout << "Simulationsdauer? " ;
      cin >> Simulationsdauer;
      cout << "Spur aufzeichnen? (ja:1, nein:0) ";
      cin >> Spur;
      cout << endl;

      // Fuer jeden Simulationslauf, also jede Replikation,
      // wird ein Modellobjekt und ein Simulator instanziiert,
      // jedes Modellobjekt hat seinen eigenen Simulator

      // Instanzierung des Simulators:
      Simulation * S = new Simulation(Spur);

      // Instanzierung des Modellobjektes der Klasse Modell:
      Modell *M = new Modell(S);

      S -> initialisiere_Simulator();
      M -> setze_Attribute();
      M -> initialisiere_Modellzustand(0, Simulationsdauer);
          // Bedeutung des 1. Parameters:
          // Modellzeit anfangs 0
      M -> initialisiere_Statistik();
      M -> Modellbeschreibung();
      S -> setze_Simulationsende(Simulationsdauer);
      S -> simuliere();
      M -> Bericht();

      cout << "Eine Simulation mit endlichem Horizont \n";
      cout << "Uhr am Simulationsende: " << S->Uhr;
      cout << endl;
      cout << endl;

      delete S;
      delete M;
    }
};

int main ()
{ int i;

```

Experiment E;

```
do
{ cout << endl <<
  "*****"
  << endl << endl <<
  "Simulationsprogramm PSS_orientierte_DES_WSS " << endl << endl;
  cout <<
  "Eine Simulation mit endlichem Horizont (1), Replikationen (2) \n";
  cout << "Ende (99) \n";
  cin >> i;

  switch(i)

  {case 1: E.eineSEH();

  break;

  };
} while (i < 99);
}
```