Discrete-event simulation system Delsi 1.1

Programmer's Guide

INTRODUCTION

The best way to study *Delsi* is to works with examples. This document contains the comments to 19 simulation applications developed in Delphi™ using *Delsi* components. Going from the first to the nineteenth sample you will get step-by-step explanation of the most common aspects of *Delsi* simulation.

Sample 1. Barbershop. The Simplest model

This example is the first in the well-known Red Book of Thomas J. Schriber "Simulation Using GPSS". Let's imagine a barbershop with one barber and the hall for waiting customers. Customers arrive to the shop. If the barber is busy they wait in the hall. They go for the service with discipline "First come - first served" (FIFO). After service they leave the barbershop.

We can describe arrival and service time intervals with help of probability distributions. These are input parameters for our model.

The arrival time is uniformly distributed in the range 12...24 min.
The service time is uniformly distributed in the range 14...20 min.
The total time of simulation is 480 min.

We are interested to determine the following values:

- Usage of the barber
- Average queue length
- Maximal queue length
- Average waiting time
- Deviation of waiting time
- Average waiting time for transactions with zero time spent in the queue
- Deviation of waiting time for transactions with zero time spent in the queue

The components for the building model:

Entrance	*TGenerator*	Arrival of customers
Hall	*TQueue*	Waiting for service with FIFO discipline
Barber	*TServer*	Serving by barber
ExitDoor	*TTerminator*	Leaving the barbershop

For the simplicity we output results into *TMemo* component. Actually, you can output the results wherever you want: to ASCII file, database, canvas, QuickReport, HTML, etc.

To make the model ready for the new simulation start-up we use *TModel.Reset* method:

```
procedure TForm1.Button1Click(Sender: TObject);
begin
   ......................
   Model.Simulate(SimTime);
   ......................
   Model.Reset;
end;
```

The results of simulation are the following.

```
Usage of the barber: 0,92
Average queue length: 0,07
Maximal queue length: 1
Average waiting time: 1,33
Deviation of waiting time: 1,82
Average waiting time / excluding zero times: 3,03
Deviation of waiting time/ excluding zero times: 1,56
```

Sample 2. Input parameters and progress status

This sample is interesting by the editing of input parameters and using progress status line. We repeat model scheme of Sample 1. The arrival and service intervals are exponentially distributed. We edit input parameters and simulation time with help of *TEdit* components.

For viewing simulation progress we use *TProgressBar* component.
For changing *TProgressBar.Position* property we use the event *Model.OnNewTime*.
This event is initialized when new model time is taken from the List of Future Events.

The source code for event handling is the following.

```
procedure TForm1.ModelNewTime(Sender: TAggregate; Trans: TTransaction);
   var NewPos: integer;
   Ratio: real;
begin
   Ratio:=ModelTime/SimTime;
   if Ratio>1.0 then Ratio:=1.0;
   NewPos:=Round(Ratio*100.0);
   with ProgressBar do
      if Position<>NewPos then Position:=NewPos;
end;
```

If you want to use the same sequence of random numbers in each simulation run, do it with *TMultiRand.Reset* method.

Sample 3. Clearing statistics during simulation run

This sample demonstrates how to clear statistics during simulation run. It may be useful when you want to define the transient of output parameters.

We use *OnPlanned* event of *TScheduler* component for:

- printing results into *Memo*
- clearing statistics
- ordering the next event for *TScheduler*

The source code for event handling is the following.

```
procedure TForm1.Scheduler1Planned(Sender: TAggregate);
begin
   Memo.Lines.Add('Avarage time in the queue: '+
                  FormatFloat('0.000',Hall.AverageTime)+' '+
                  FormatFloat('0.00',ModelTime));
   Model.ClearStatistics;
   Sender.NextTime(ClearTime);
end;
```

The arrival and service time are uniformly distributed.

The example of simulation results is the following.

```
Average time in the queue: 2,735 10000,00
Average time in the queue: 2,500 20000,00
Average time in the queue: 2,094 30000,00
Average time in the queue: 2,070 40000,00
Average time in the queue: 2,348 50000,00
Average time in the queue: 2,430 60000,00
Average time in the queue: 2,140 70000,00
Average time in the queue: 2,525 80000,00
Average time in the queue: 2,431 90000,00
Average time in the queue: 2,804 100000,00
```

Sample 4. Experiments with changing parameters

This sample demonstrates an experiment management. In this sample the arrival and service time are exponentially distributed. The mean of service time is 10.0. The mean of arrival time changes from 10.0 to 15.0 with step 1.0.

We need to estimate how the average queue length depends on the average arrival time.

The source code of experiment management:

```
ArrivalTime:=10.0;
ServiceTime:=10.0;
Memo.Lines.Clear;
for i:=0 to 5 do
begin
   Model.Simulate(LimitTime);
   Memo.Lines.Add('Average arrival time'+FormatFloat('0.00',ArrivalTime)+
                  ' Average queue length: '+
                  FormatFloat('0.00',Hall.AverageCount) );
   Model.Reset;
   ArrivalTime:=ArrivalTime+1.0;
end;
```

The results of experiment for simulation time 1000.0

```
Average arrival time: 10,00 Average queue length: 6,88
Average arrival time: 11,00 Average queue length: 2,43
Average arrival time: 12,00 Average queue length: 3,12
Average arrival time: 13,00 Average queue length: 1,06
Average arrival time: 14,00 Average queue length: 1,07
Average arrival time: 15,00 Average queue length: 0,63
```

Sample 5. Tracing

In this sample we repeat the model of Sample 1. The only thing we demonstrate here is how to trace the simulation process. To trace the transaction passing from one block to another we use the event *OnAfterPass* of *TModel* component.

The source code of event handling is the following.

```
procedure TForm1.ModelAfterPass(Sender, Receiver: TBlock; Trans: TTransaction);
begin
   Memo.Lines.Add(PadCh(Sender.Name,' ',12)+ // Name of Sender
                  PadCh(Receiver.Name,' ',12)+ // Name of Receiver
                  PadCh(IntToStr(Trans.GetTransID),' ',6)+ // Transaction ID
                  FormatFloat('0.00',ModelTime)); // System time
end;
```

The result of tracing looks like this:

```
Entrance Hall 1 0,00
Hall Barber 1 0,00
Barber ExitDoor 1 14,50
Entrance Hall 1 20,49 // The second life of Transaction 1
Hall Barber 1 20,49
Barber ExitDoor 1 34,79
Entrance Hall 1 41,89
Hall Barber 1 41,89
Barber ExitDoor 1 60,08
Entrance Hall 1 64,48 // The third life of Transaction 1
Hall Barber 1 64,48
Entrance Hall 2 79,40
Barber ExitDoor 1 80,94
Hall Barber 2 80,94
Barber ExitDoor 2 96,11
Entrance Hall 2 96,44 // The second life of Transaction 2
Hall Barber 2 96,44
Entrance Hall 1 112,58
```

From these tracing results you can see that each transaction can be used several times. It looks like the transactions may have several lives per simulation run. But each transaction in each moment of model time has unique ID.

By use of *OnAfterPass* and *OnNewTime* events you can create more sophisticated tracing, for instance, with the queue lengths, transaction priorities or something else. Of course, you can customize the look and feel of the output.

Sample 6. Limited queue capacity and routing

Let's take in consideration that number of chairs in the hall is limited. If the customer finds out that all chairs are busy she goes to another barbershop.

> The arrival time is exponentially distributed with the mean equal to 11.0.
> The service time is exponentially distributed with the mean equal to10.0.
> We need to estimate the percentage of lost customers.

We define the capacity of queue in the property *Capacity* of *TQueue* component. We can do it in Object Inspector or directly in source code just like in this sample.

```
Hall.Capacity:=QCapacity;
```

If the number of transactions in the queue is equal to the capacity, the queue becomes not ready to receive transactions. We use this fact to route transactions.

```
procedure TForm1.EntranceRouting(Sender: TBlock; Trans: TTransaction);
begin
   if Hall.isReadyToReceive(Trans) then
      Sender.PassTo(Hall)
   else
      Sender.PassTo(AnotherShop);
end;
```

After the simulation run it is very easy to calculate the percentage of the lost customers.

```
Memo.Lines.Add('Percent of lost customers: '+
            FormatFloat('0.00',AnotherShop.Entries*100.0/Entrance.Exits)+'%');
```

The example of simulation results:

```
Capacity: 5
Total customers: 8990
Lost customers: 936
Percent of lost customers: 10,41%
```

Sample 7. Limited waiting in the queue

In previous sample we have discussed the queue with limited capacity. We repeat that sample with some addition. Usually, some of customers cannot spend a long time in the queue. Let's imagine that half of the customers are not ready to spend in the queue more than 20 ... 30 min. (uniformly distributed). After this period of time they leave the queue.

As it was in previous sample we need to calculate the percentage of lost customers.

To limit the waiting time for some customers we order the limiting time in the *OnEnter* event of *TQueue* component.

```
procedure TForm1.HallEnter(Sender: TBlock; Trans: TTransaction);
begin
   if MultiRand.Uniform(0.0,1.0) > 0.5 then // only half of customers can't wait
       Sender.NextTime(MultiRand.Uniform(20.0,30.0));
end;
```

When the waiting time expires, the event *OnTimeFinish* occurs. We process this event by passing transaction into terminator *NoTime.*

```
procedure TForm1.HallTimeFinish(Sender: TBlock; Trans: TTransaction);
begin
    Sender.Pass(AnotherShop); // Pass to terminator
end;
```

After simulation run we can calculate results.

```
Memo.Lines.Add('Total customers: '+IntToStr(Entrance.Exits));
Memo.Lines.Add('Total amount of lost customers: '+IntToStr(AnotherShop.Entries));
Memo.Lines.Add('Lost customers due to limited time waiting: '
               +IntToStr(Hall.TimeLimitExits));
Memo.Lines.Add('Lost customers due to limited capacity: '+
               IntToStr(AnotherShop.Entries-Hall.TimeLimitExits));
Memo.Lines.Add('Percent of lost customers: '+FormatFloat('0.00',
               AnotherShop.Entries*100.0/Entrance.Exits)+'%');
```

The example of simulation results:

```
Capacity: 5
Total customers: 9084
Total amount of lost customers: 1465
Lost customers due to limited time waiting: 1136
Lost customers due to limited capacity: 329
Percent of lost customers: 16,13%
```

Sample 8. Changing the parameters during simulation run

In this sample we repeat the model scheme of Sample 1. Let's consider the fact that arrival rate usually is not the same all time of the day. We need to build model with this additional condition.

Both arrival and service time are exponentially distributed.
The mean of service time is 10.0.

We need to simulate barbershop with the following rate dependence.

Period (hrs)	Period since opening (min.)	Average arrival interval (min.)
8.00 - 10.00	0-120.00	14.0
10.00 - 12.00	120-240	12.0
12.00 - 15.00	240-420	10.0
15.00 - 17.00	420-540	12.0
17.00 - 20.00	540-720	14.0

To keep those values we use two arrays.

```
ChangeTime: array[1..5] of real = (0.0,120.0,240.0,420.0,540.0);
ArrivalMeans: array[1..5] of real = (14.0,12.0,10.0,12.0,14.0);
```

To change average arrival interval in defined moments of model time we use *TScheduler* component.

```
{Initializing first planned event}
procedure TForm1.Scheduler1BeforeTimeGoOn(Sender: TAggregate);
begin
   Sender.NextTime(0.0); // The first planned event will occur at 0.0
end;

{Handling planned event}
procedure TForm1.Scheduler1Planned(Sender: TAggregate);
begin
   TArrival:=ArrivalMeans[Counter]; // Setting the mean of arrival time
   if Counter<5 then
      Sender.NextTime(ChangeTime[Counter+1]); // Ordering the next event
   Inc(Counter);
end;
```

The results of simulation are the following.

```
Usage of the barber: 0,83
Average queue length: 2,16
Maximal queue length: 7
Average waiting time: 22,67
Deviation of waiting time: 19,46
Average waiting time / excluding zero times: 30,39
Deviation of waiting time/ excluding zero times: 16,53
```

Sample 9. Using TCreator component

Sometimes we need to generate transactions explicitly. In *Delsi* it is possible with the help of *TCreator* component.

Let's imagine that before opening there are several people waiting for service. We need to define the output parameters that depend on the number of that people. Both arrival and service time are exponentially distributed with means 10.0 and 11.0 accordingly.

To create transactions directly we use *OnBeforeTimeGoOn* event of *Entrance* component.

```
procedure TForm1.EntranceBeforeTimeGoOn(Sender: TAggregate);
begin
   Sender.NextTime(MultiRand.Exponential(11.0));
   if NumberOfPeople>0 then
      PeopleInStreet.Generate(NumberOfPeople); // Direct generation of transactions
end;
```

Actually, you can use any event for direct generation of transaction. We route generated transaction to queue *Hall*.

```
procedure TForm1.PeopleInStreetRouting(Sender: TBlock; Trans: TTransaction);
begin
   Sender.Pass(Hall);
end;
```

The results of simulation for 10 initially waiting clients are the following.

```
Usage of the barber: 0,90
Average queue length: 3,36
Maximal queue length: 11
Average waiting time: 32,26
Deviation of waiting time: 20,69
Average waiting time / excluding zero times: 35,79
Deviation of waiting time/ excluding zero times: 18,67
```

Sample 10. Using TStorage component

Let's return for sample 7. We have the system with limited capacity of the queue and limited waiting period in the queue. Let's assume that there are several barbers in the barbershop. The owner of the saloon needs our help to decide how many barbers should work for him.

The arrival and service time are exponentially distributed.
The mean of arrival time is 3.0
The mean of service time is 10.0.
The hall capacity is 5.

To simulate several barbers we use *TStorage* component. We carry out experiments changing capacity of storage *Barbers*.

```
Memo.Lines.Add('Barbers Losses Usage of one barber');
for i:=1 to 8 do
begin
   Barbers.Capacity:=i;
   Model.Simulate(TLimit);
   Memo.Lines.Add(IntToStr(Barbers.Capacity)+' '+
   FormatFloat('00.00',(AnotherShop.Entries+NoTime.Entries)*
               100.0/Entrance.Exits)+'% ' +
   FormatFloat('00.00',Barbers.AverageCount*100.0/
   Barbers.Capacity)+'%');
   Model.Reset;
end;
```

The results of simulation for simulation time 720.0

```
Barbers      Losses       Usage of one barber
1            69,70%       100,00%
2            41,30%       97,36%
3            23,05%       93,16%
4            11,60%       81,84%
5            04,03%       75,26%
6            00,00%       50,50%
7            00,00%       49,53%
8            00,00%       45,07%
```

Sample 11. Failures and recovers, component TGate, method TServer.Release

Let's consider a workshop, which consists of the box for details and machine tool. The details arrive to be processed on the machine tool from the box in LIFO order. After processing, details move to another site.

From time to time machine tool breaks. The worker needs time to fix it. After the breakage of the tool the detail is removed from the machine tool and omitted in the box for further processing, which will take as much time as any other details in the box.

Both arrival and processing intervals are exponentially distributed with the mean values 2.0 and 1.5 min. The input parameters are the mean of the interval between tool failures and the mean of recovery interval. These times are exponentially distributed. Another input parameter is time of simulation.

Before simulation run we unlock the *Gate* and plan the first failure using TScheduler component *Scheduler*.

```
procedure TForm1.SchedulerBeforeTimeGoOn(Sender: TAggregate);
begin
   Gate.Unlock;
   Sender.NextTime(MultiRand.Exponential(TFailure));
end;
```

During simulation we lock and unlock Gate using *OnPlanned* event of *Scheduler.* While locking *Gate*, we imitate removal of details by use of *Release* method.

```
procedure TForm1.SchedulerPlanned(Sender: TAggregate);
begin
   if Gate.isLocked then
   begin
      Sender.NextTime(MultiRand.Exponential(TFailure));
      Gate.UnLock;
   end
   else
   begin
      Sender.NextTime(MultiRand.Exponential(TRecovery));
      Tool.Release; // Removal of a details
      Gate.Lock;
   end;
end;
```

We pass removed transaction back into the *Box* using *OnRelease* event of the *Tool* component.

.

```
procedure TForm1.ToolRelease(Sender: TBlock; Trans: TTransaction);
begin
   Sender.PassTo(Box);
end;
```

If there is a transaction in the gate after locking, we pass it to bunker.

```
procedure TForm1.GateRouting(Sender: TBlock; Trans: TTransaction);
begin
   if Gate.isLocked then
      Sender.PassTo(Bunker)
   else
      Sender.PassTo(Tool);
end;
```

Sample 12. Routing that depends on the state of blocks. Transaction fields. Tabulation.

Let's imagine the bank with three cash desks. A visitor goes to cash desk, which is free. If all cash desks are busy she goes to the queue with minimal length. The arrival and service time are exponentially distributed. We need to calculate the mean, the deviation of the time spent in the bank and to build the corresponding histogram.

We route transactions to the queues when they exit from generator *Entrance*.

```
procedure TForm1.EntranceRouting(Sender: TBlock; Trans: TTransPtr);
var QueueID: integer;
    MinCount: longint;
begin

   if (CashDesk1.Count=0) and (Queue1.Count=0) then
   begin
      Sender.PassTo(Queue1);
      Exit;
   end;
   if (CashDesk2.Count=0) and (Queue2.Count=0) then
   begin
      Sender.PassTo(Queue2);
      Exit;
   end;
   if (CashDesk3.Count=0) and (Queue3.Count=0) then
   begin
      Sender.PassTo(Queue3);
      Exit;
   end;

   {Choosing the queue with minimal length}
   MinCount:=Queue1.Count;
   QueueID:=1;

   if Queue2.Count<MinCount then
   begin
      MinCount:=Queue2.Count;
      QueueID:=2;
   end;

   if Queue3.Count<MinCount then
   begin
      MinCount:=Queue3.Count;
      QueueID:=3;
   end;

   {Pass transaction to queue with minimal length}
   case QueueID of
        1: Sender.PassTo(Queue1);
        2: Sender.PassTo(Queue2);
        3: Sender.PassTo(Queue3);
   end;

end;
```

How to determine the time spent in the bank? We need to store the birth moment of the transaction when the transaction exits form generator. In order to have a field to store the birth time in the transaction we need to declare new transaction class inherited from *TTransaction*.

```
MyTransaction = class(TTransaction)
public
   BirthTime: real;
end;
```

In order to inform the internal simulation manager about the new declaration of transaction we use the method *TModel.SetTransactionClass* before simulation run.

```
Model.SetTransactionClass(MyTransaction);
Model.Simulate(LimitTime);
```

We store the value of birth time when handle the event *TGenerator.OnExit*.

```
procedure TForm1.EntranceExit(Sender: TBlock; Trans: TTransPtr);
begin
   Sender.NextTime(MultiRand.Exponential(TArrival));
   (Trans as MyTransaction).BirthTime:=ModelTime;
end;
```

When transaction reaches the terminator we calculate the difference between current model time and the birth time. To obtain the mean, deviation and histogram we tabulate these values using *TTabulator* component.

```
procedure TForm1.ExitDoorEnter(Sender: TBlock; Trans: TTransPtr);
begin
   Tabulator1.PutValue(ModelTime-(Trans as MyTransaction).BirthTime);
end;
```

After simulation run we output the information collected in the tabulator.

```
with Tabulator1 do
begin
   Memo.Lines.Add('The histogram of time spending in the bank');
   Memo.Lines.Add('Below '+FormatFloat('000.00',LowerBound)+': '+
   IntToStr(Hits(0)));
   for i:=1 to Tabulator1.IntervalCount do
   begin
      Memo.Lines.Add(FormatFloat('000.00',LowerBound+Interval*(i-1))+' - '+
                     FormatFloat('000.00',LowerBound+Interval*(i))+' '+
                     IntToStr(Hits(i)));
   end;
   Memo.Lines.Add('Upper '+FormatFloat('000.00',LowerBound+
                  Interval*IntervalCount)+': '+
                  IntToStr(Hits(IntervalCount+1)));
                  Memo.Lines.Add('_____________________________________________');
   Memo.Lines.Add('Average time in the bank: '+
                  FormatFloat('0.00',Tabulator1.Mean));
   Memo.Lines.Add('Deviation of time in the bank: '+
                  FormatFloat('0.00',Tabulator1.Deviation));
end;
```

The example of the result output is the following.

```
The histogram of time spent in the bank
Below 000,00: 0
000,00 - 002,00 789
002,00 - 004,00 526
004,00 - 006,00 269
006,00 - 008,00 126
008,00 - 010,00 83
010,00 - 012,00 35
012,00 - 014,00 10
014,00 - 016,00 0
016,00 - 018,00 0
018,00 - 020,00 0
Upper 020,00: 0
_____________________________________________
Average time in the bank: 3,09
Deviation of time in the bank: 2,60
```

Sample 13. TDivider and TAssembler

In this sample we consider some workshop. The details arrive to the workshop in boxes by batches of 10 pieces. After processing on the machine tool they go to another box and then to another workshop. The box arrival interval and processing time are uniformly distributed. It is necessary to build a histogram of the box departure interval. The input parameters are the lower and upper bounds of arrival interval and processing time. One more input parameter is the simulation time.

We simulate box arriving by *TGenerator* component *BoxArrival.* To decompose "one box" into "ten details" we use *TDivider* component *BoxToDetails* with capacity 10. After processing on machine tool (*TServer* component *Tool*) we compose every "ten details" into "one box" by use of *Tassembler* component *DetailsToBox* with capacity 10.

We build histogram with help of *TTabulator* component *Tabulator1*.

```
procedure TForm1.AnotherWorkShopEnter(Sender: TBlock; Trans: TTransaction);
begin
   Tabulator1.PutValue(ModelTime-LastArriveTime);
   LastArriveTime:=ModelTime;
end;
```

The result of simulation for arrival interval from 2.0 to 4.0 and interval of processing from 25.0 to 35.0

```
The histogram of box departure interval
Below 020,00: 1
020,00 - 021,00 0
021,00 - 022,00 0
022,00 - 023,00 0
023,00 - 024,00 20
024,00 - 025,00 99
025,00 - 026,00 228
026,00 - 027,00 349
027,00 - 028,00 338
028,00 - 029,00 335
029,00 - 030,00 335
030,00 - 031,00 341
031,00 - 032,00 313
032,00 - 033,00 333
033,00 - 034,00 321
034,00 - 035,00 213
035,00 - 036,00 95
036,00 - 037,00 20
037,00 - 038,00 0
038,00 - 039,00 0
039,00 - 040,00 0
Upper 040,00: 0
_____________________________________
Average output interval 29,93
Deviation of output interval: 3,03
```

That is a good illustration for the Theorem of Large Numbers.

Sample 14. Using priorities with TQueuePrty component

In this sample we are considering the dental clinic with several doctors. The patients may be divided into two categories: regular patients and patients with tooth pain. The patients may form the queue. In that case patients with pain will go for the treatment first. So, we can say that they have higher non-preemptive priority in the medical service.

The following parameters are known:

The arrival time is exponentially distributed.
The mean of arrival time for regular patients is 4.0 min.
The mean of arrival time for patients with tooth pain is 25.0 min.
The service time is uniformly distributed from 7.0 to 15 min.

We need to determine the average value and standard deviation of time spent in clinic for the both categories of patients in dependence on the number of doctors.

By default, the priority of each new transaction in *Delsi* is equal to 0. So, the generator *RegularPatients* generates transaction with lower priority level.

We set the priority level for transactions generated by the generator *ToothpainPatiens* with help of *OnAfterGeneration* event:

```
procedure TForm1.ToothpainPacientsAfterGeneration(Sender: TBlock;
                                                  Trans: TTransaction);
begin
   Trans.SetPrty(1); // Sets priority value into 1
end;
```

The *TQueuePrty* component handles transactions in the way that high-priority transactions will be placed at the beginning of the queue. They will leave the queue first. Actually, there are two hidden internal FIFO queues inside the general queue. So, the number of the hidden queues is equal to numbers of priority levels of transactions stored in that real queue. Each hidden queue stores transaction of some priority level. If there are high-priority transactions in the queue, low priority transaction can leave the queue only for the reason of limitation of waiting time.

The results of simulation for 3 doctors and simulation time 720.0:

```
1. Regular patients
   Number of patients: 155
   The average time spent in clinic: 24,88
   The deviation of time spent in clinic: 9,22
2. Patients with tooth pain
   Number of patients: 33
   The average time spent in clinic: 12,99
   The deviation of time spent in clinic: 3,36
```

Sample 15. Preempted and postponed service

In this sample we are considering the firm, which executes some orders. There are two types of orders: regular and urgent. The last one costs twice as much. When the firm executes regular order and urgent order comes for processing, the firm preempts processing of regular order. After finishing urgent order, the firm continues processing regular order.

The following parameters are known:

The arrival and service times are exponentially distributed.
The mean of arrival time of regular orders is 5.0 days
The mean of arrival time of urgent orders is 15.0 days
The mean of service time is 3.5 days
Simulated time – 1 year (about 264 working days)

We need to determine the number and the average time of execution for both types of orders. To solve this task we use transactions with different priority levels. To be able to preempt service in the server the high-priority transactions have to be preemptive. We set the priority level and its preemptive ability in *OnAfterGeneration* event.

```
procedure TForm1.UrgentOrdersAfterGeneration(Sender: TBlock; Trans: TTransPtr);
begin
   SetPrty(Trans,1);
   SetPreempt(Trans);
end;
```

When high-priority transaction preempts the service of low-priority one, the server generates *OnPreempt* event. (Do not forget to set server's property *Preemptive* into *True*). Handling that event, we postpone the service of preempted transaction with help of *Postpone* method.

```
procedure TForm1.ProcessingPreempt(Sender: TBlock; Trans: TTransaction);
begin
   Sender.Postpone;
end;
```

When high-priority transaction leaves the server, the last one continues processing of low-priority transaction. The results of simulation are the following.

```
1. Regular orders
   Number of arrived orders: 44
   Number of executed orders: 34
   The average execution time: 21,95
   The deviation of execution time: 16,59
2. Urgent orders
   Number of arrived orders: 17
   Number of executed orders: 17
   The average execution time: 5,53
   The deviation of execution time: 4,22
_____________________________________________
The average number of orders on processing: 1,12
The maximal number of orders on processing: 2
The loading of firm: 0,82
The average length of the queue: 2,84
The maximal length of the queue: 10
The cost of executed orders: 680000,00
```

Sample 16. Passing preempted low-priority transactions

In this sample we consider another business strategy for the firm of Sample 15. When the processing of regular order is preempted by urgent order, the firm gives that regular order to the company-subcontractor. The input parameters and the task are the same like in the Sample 15.

By Handling *OnPreempt* event, we pass preempted transaction to another block.

```
procedure TForm1.ProcessingPreempt(Sender: TBlock; Trans: TTransaction);
begin
   Sender.PassTo(Subcontractor);
end;
```

The results of simulation are the following.

```
1. Regular orders
   Number of arrived orders: 45
   Number of executed orders: 25
   The average execution time: 12,22
   The deviation of execution time: 12,51
2. Urgent orders
   Number of arrived orders: 17
   Number of executed orders: 17
   The average execution time: 7,20
   The deviation of execution time: 7,98
3. Orders passed to subcontractor
   Number of orders: 13
_____________________________________________
The average number of orders in processing: 0,69
The maximal number of orders in processing: 1
The loading of firm: 0,685
The cost of executed orders: 590000,00
```

Sample 17. Preempting in TQueuePrty

In this sample we consider the third business strategy for the firm of Samples 15,16. According to this strategy the firm limits the size of queue up to 3 orders. Urgent order may preempt the regular order waiting in the queue. Also this strategy supposes preempting the processing.

Let's imagine that firm process one order and keeps three other in the queue. We can consider two ways of the new order arrival:

1. The new order is regular. In this case the firm passes the new order to a subcontractor.
2. The new order is urgent.
    2a: The queue contains three urgent orders. In this case new order will be passed to the subcontractor.
    2b: The queue contains at least one regular order. In this case urgent order will replace the regular one. The replaced regular order will be passed to the subcontractor.

The preempting service is in use as well.

We implement the first way by handling of *OnRouting* event of *RegularOrders* generator.

```
procedure TForm1.RegularOrdersRouting(Sender: TBlock; Trans: TTransaction);
begin
   if Queue.IsReadyToReceive(Trans) then
      Sender.PassTo(Queue)
   else
      Sender.PassTo(Subcontractor)
   end;
```

We implement *2a* in similar way.

```
procedure TForm1.UrgentOrdersRouting(Sender: TBlock; Trans: TTransaction);
begin
   if Queue.IsReadyToReceive(Trans) then
   Sender.PassTo(Queue)
else
   Sender.PassTo(Subcontractor)
end;
```

In this procedure the function *Queue.IsReadyToReceive(Trans)* returns *False* only if the queue contains three high-priority transactions, otherwise it returns *True*.

Considering the variant 2b, we can say that *TQueuePrty* is being preempted like a server. We pass preempted transaction by handling *OnPreempt* event (Do not forget to set *Preemptive* property of components *Queue* and *Processing* into *True*).

```
procedure TForm1.QueuePreempt(Sender: TBlock; Trans: TTransaction);
begin
   Sender.PassTo(Subcontractor);
end;
```

The results of simulation are the following.

```
1. Regular orders
   Number of arrived orders: 44
   Number of executed orders: 24
   The average execution time: 11,40
   The deviation of execution time: 10,52
2. Urgent orders
   Number of arrived orders: 17
   Number of executed orders: 17
   The average execution time: 4,55
   The deviation of execution time: 3,86
3. Orders passed to subcontractor
   Number of orders: 17
   Orders passed from queue: 2
   Orders passed from processing: 10
_____________________________________________
The average number of orders in processing: 0,72
The maximal number of orders in processing: 1
The loading of firm: 0,72
The average length of the queue: 1,0971
The maximal length of the queue: 3
The cost of executed orders: 580000,00
```

Sample 18. Preempting in TStoragePrty

In this sample we consider the forth business strategy for that firm. This strategy is similar to the strategy of Sample 17 with preempting in *TQueuePrty*.

Let's imagine that firm is able to process several orders simultaneously. One employee can process one order. If there is at least one urgent order in the queue and one regular order is on processing, the urgent order will replace the regular one, which is being processed. The replaced regular order will be passed to a subcontractor.

We simulate the processing with *TStoragePrty* component. The value of *Capacity* is the number of employees.

We pass preempted transaction to the block *Subcontractor* by handling of *OnPreempt* event of *TStoragePrty*.

```
procedure TForm1.ProcessingPreempt(Sender: TBlock; Trans: TTransPtr);
begin
   Sender.PassTo(Subcontractor);
end;
```

The results of simulation for three employees are the following.

```
1. Regular orders
   Number of arrived orders: 45
   Number of executed orders: 30
   The average execution time: 12,58
   The deviation of execution time: 8,79
2. Urgent orders
   Number of arrived orders: 17
   Number of executed orders: 16
   The average execution time: 8,59
   The deviation of execution time: 6,45
3. Orders passed to subcontractor
   Number of orders: 11
_____________________________________________________
The average number of orders in processing: 2,31
The maximal number of orders in processing: 3
The cost of executed orders: 620000,00
```

Sample 19. Multiple forms

What should we do if we have so many components in the model that we cannot place them on one form? We can place them on several forms. Tthe internal control subsystem will gather all aggregates into one united model before simulation run. In this sample we place the blocks of the model on different forms.

On Form2: generator *Entrance*, queue *Hall*;
On Form3: server *Barber*, terminator *ExitDoor*.

If we pass transaction to the block of another form, we need to do it this way.

```
procedure TForm2.HallRouting(Sender: TBlock; Trans: TTransaction);
begin
   Sender.PassTo(Form3.Barber);
end;
```

The same way we refer to *MultiRand*, which is placed in Form1.

```
procedure TForm2.EntranceExit(Sender: TBlock; Trans: TTransaction);
begin
   Sender.NextTime(Form1.MultiRand.Exponential(ArrivalTime));
end;
```

When we implement output of results, we also refer to the blocks placed in other forms.

```
procedure TForm1.Button1Click(Sender: TObject);
begin
   { Checking all input paramters }
   . . . . . .
   Model.Simulate(SimTime);
   Memo.Lines.Add('Usage of the barber:'+
                  FormatFloat('0.00',Form3.Barber.Usage));
   Memo.Lines.Add('Average queue length: '+
                  FormatFloat('0.00',Form2.Hall.AverageCount));
   Memo.Lines.Add('Maximal queue length: '+
                  IntToStr(Form2.Hall.MaxCount));
   Memo.Lines.Add('Average waiting time: '+
                  FormatFloat('0.00',Form2.Hall.AverageTime));
   Memo.Lines.Add('Deviation of waiting time: '+
                  FormatFloat('0.00',Form2.Hall.DeviationTime));
   Memo.Lines.Add('Average waiting time / excluding zero times: '+
                  FormatFloat('0.00',Form2.Hall.SAverageTime));
   Memo.Lines.Add('Deviation of waiting time/ excluding zero times: '+
                  FormatFloat('0.00',Form2.Hall.SDeviationTime));
   Model.Reset;
   MultiRand.Reset;
end;
```

Do not forget to define **Uses** clause in **implementation** part of the unit.

In Sample19.pas: `Uses Unit2,Unit3;`
In Unit2.pas: `Uses Sample19, Unit3;`
In Unit3.pas: `Uses Sample19;`