Discrete-event Simulation System

DELSI 2.0

Tutorial

Table of Content

Introduction

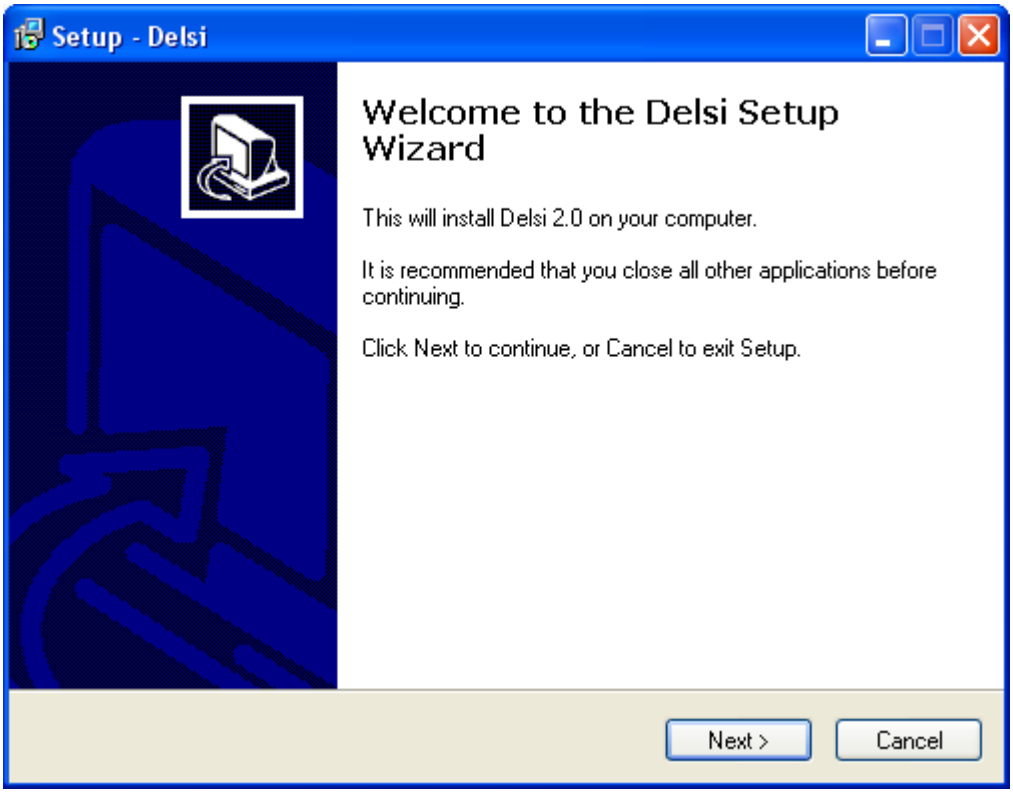
The best way to study Delsi is to work with examples. This document contains the comments to numerous simulation applications developed in Visual Studio 2010 using Delsi components. Going from the first to the last sample you will get step-by-step explanation of the most common aspects of Delsi simulation.

This tutorial is intended for .NET developers of entry level. The experienced programmers may find some of the examples or comments too obvious. The simulation tasks may seem naive and not quite real. Our goal is to make this tutorial clear for a wide range of developers, engineers and analysts.

Setup

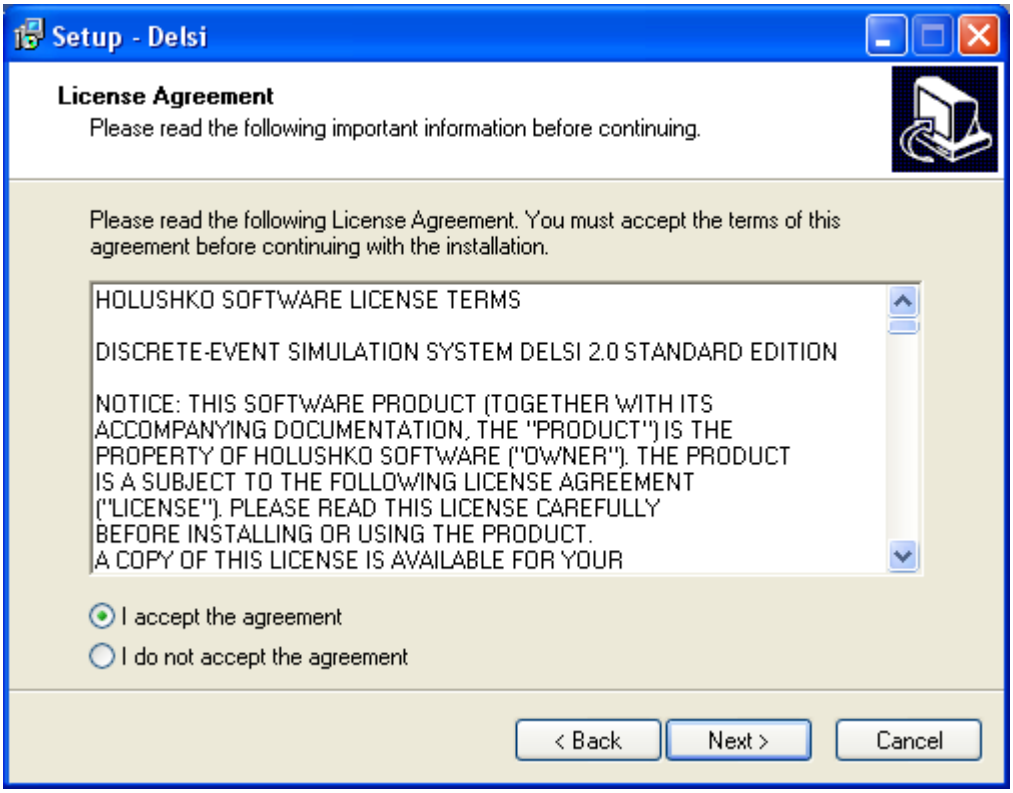
1. Run Delsi installer `delsi_setup.exe`

2. You will see the following wizard.

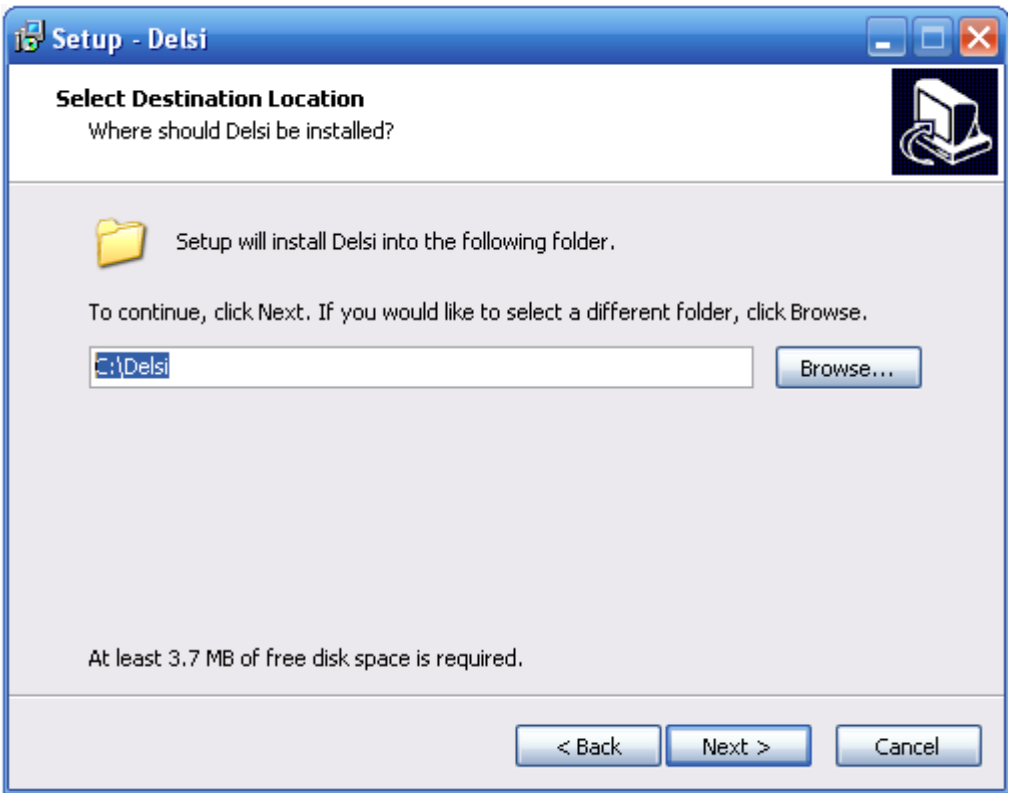


3. Press button “Next >”.

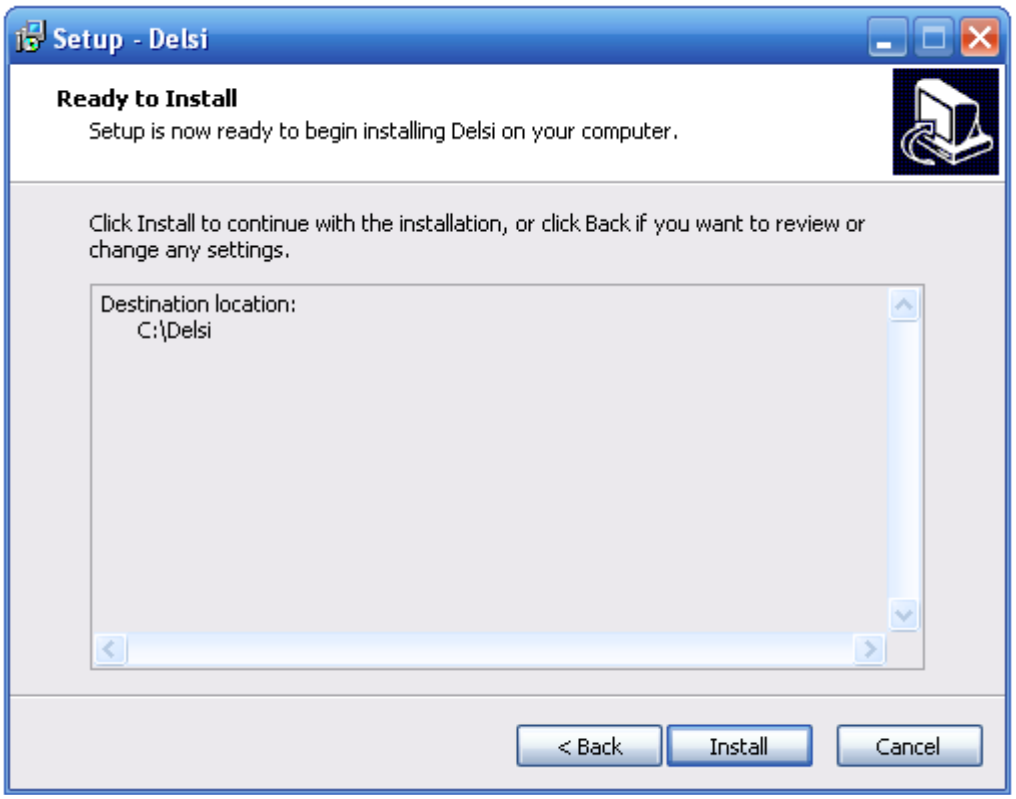
4. Read license agreement. If you accept it, choose option “I accept the agreement”and press button “Next >”



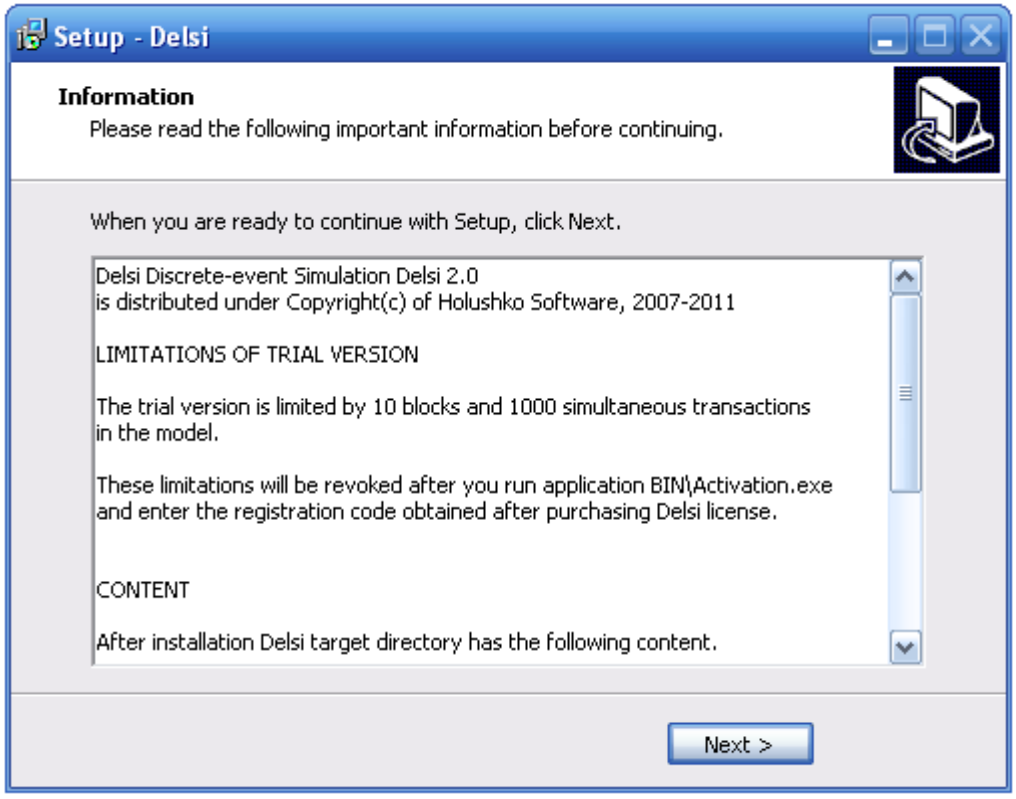
5. Select the location for Delsi home directory and press button “Next >”



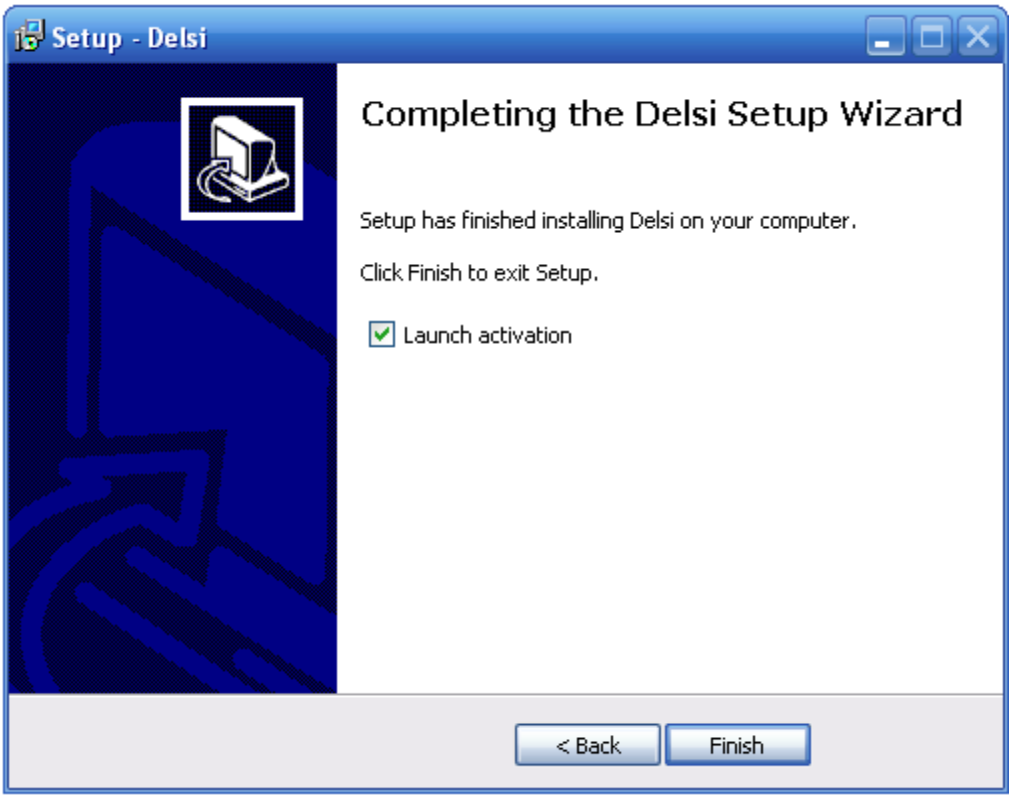
6. On the next screen button “Install”.



7. Read the information related to the installation and press button “Next >”.



8. Choose if you want to activate Delsi with registration code and press button “Finish”.



At this point installation is completed. As a result of installation, Delsi home directory will have the following content.

```
Delsi.dll - .NET 2.0 assembly with Delsi 2.0 components
readme.txt - This file
license.txt - License agreement
BIN\Activation.exe - Activation application
MANUALS\API_Reference.pdf - API Reference
MANUALS\Tutorial.pdf - Tutorial
MANUALS\DelsiHelp.chm - Help file
SAMPLES 2010 - 26 sample projects for step-by-step study
```

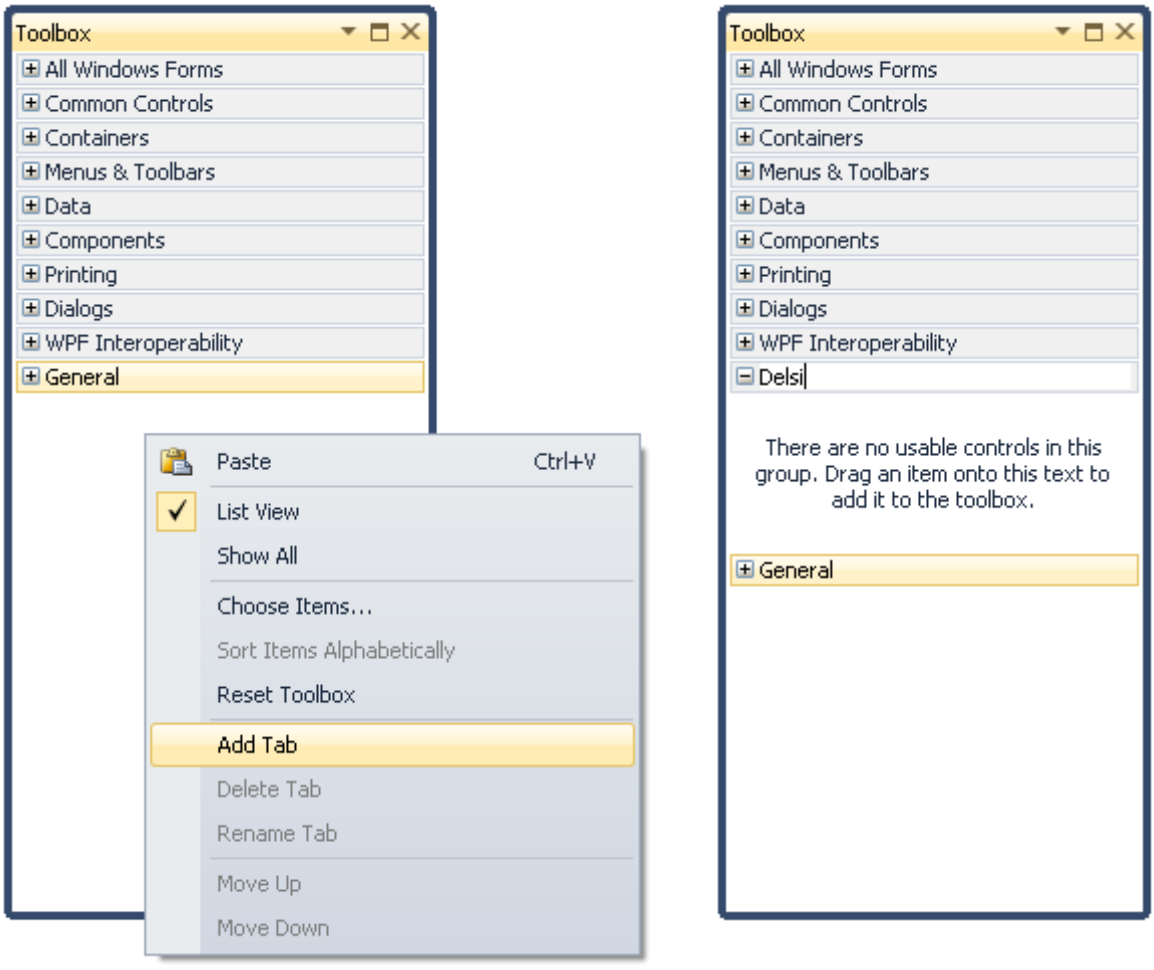
Besides that, Delsi setup program creates a registry key:

```
HKEY_CURRENT_USER\SOFTWARE\Microsoft\.NETFramework\AssemblyFolders\Delsi
```

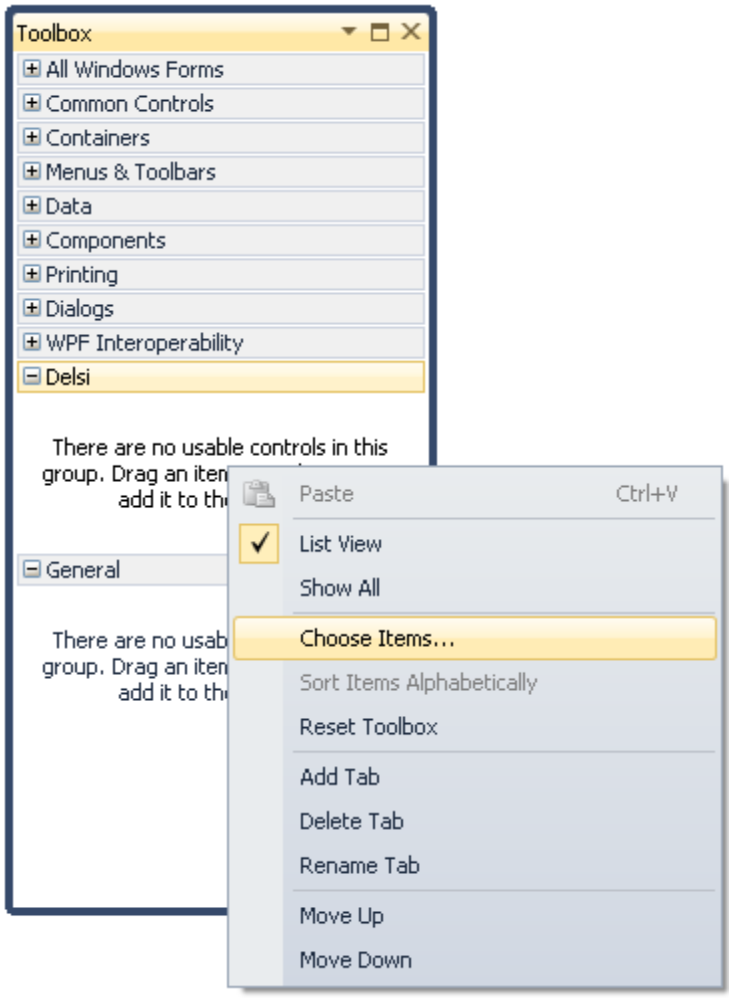
Installing Delsi 2.0 in Visual Studio 2010

This section describes how to install *Delsi* in *Microsoft Visual Studio 2010.*

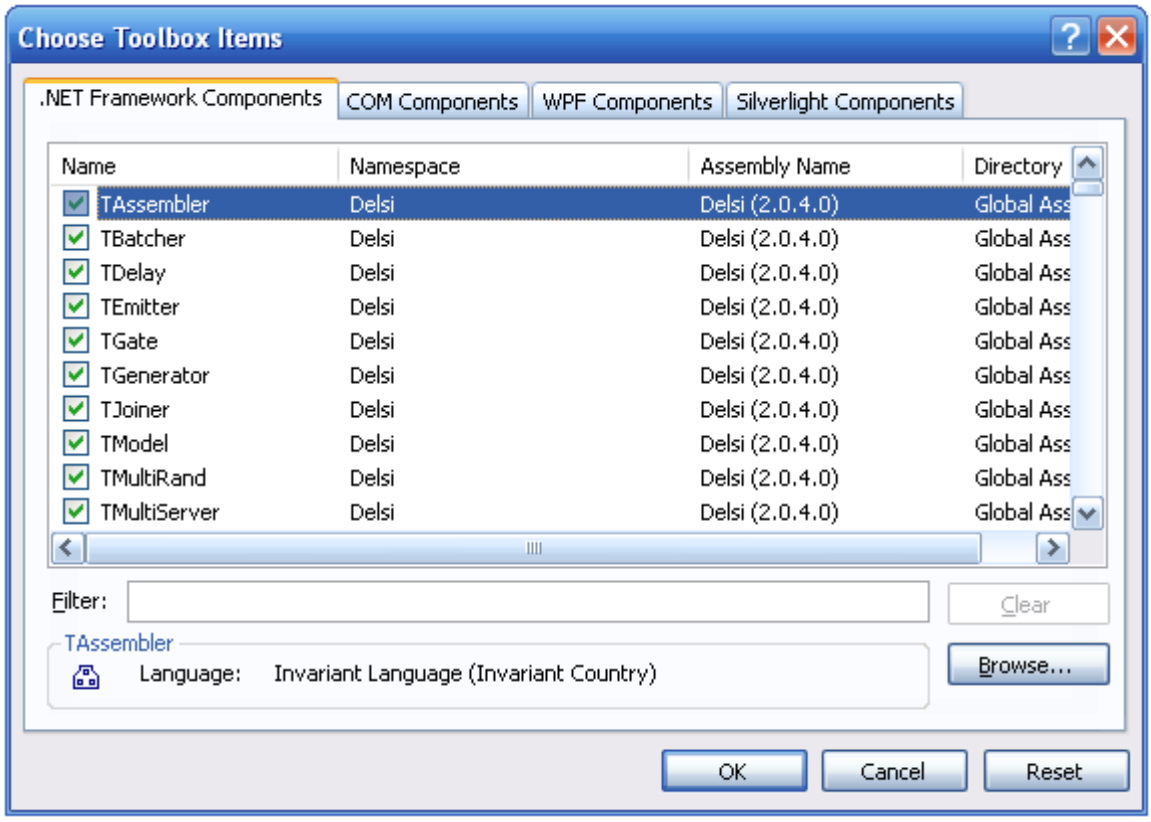
1. Start Microsoft Visual Studio 2010.
2. In its top menu “View” choose option “Toolbox”.
3. Right click on Toolbox and choose option “Add Tab”.
4. Type “Delsi” in the name field.



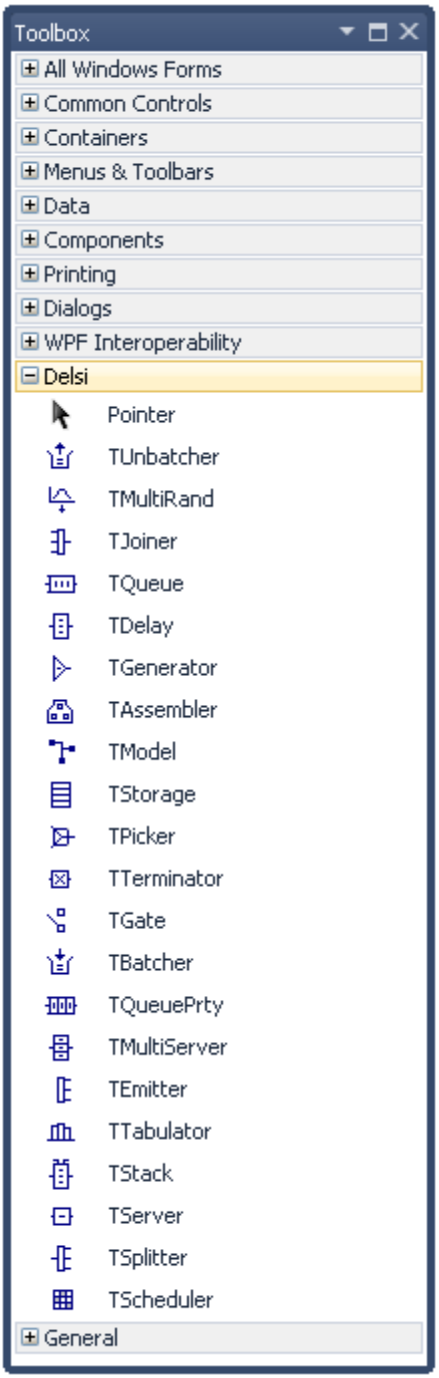
5. Right click on Toolbox on the tab “Delsi” and choose option “Choose Items...”.



6. In the dialog window “Choose Toolbox Item” press button “Browse...” and select Delsi.dll in the folder where it was installed. Then choose all components of namespace Delsi and press button “OK”.



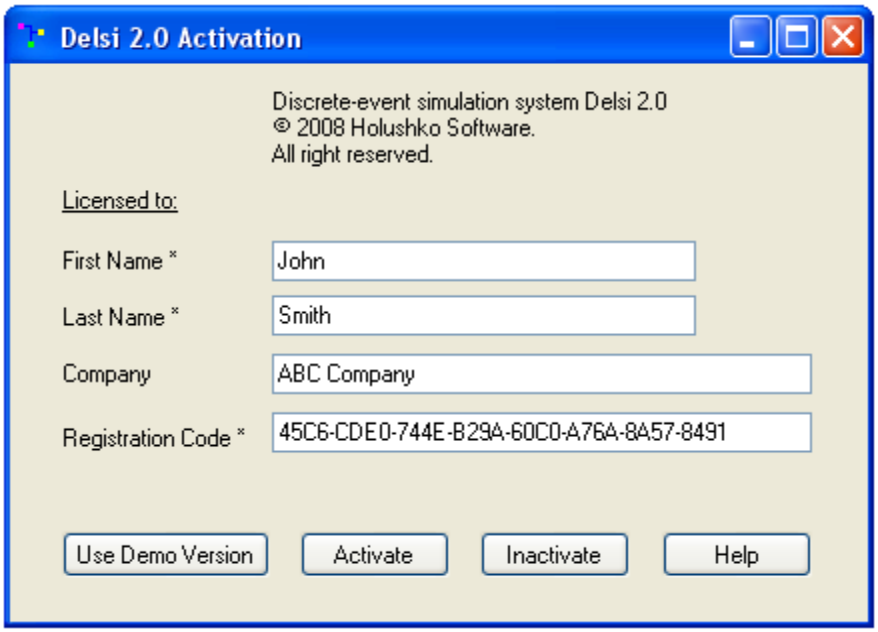
7. Now Delsi components are installed in the Toolbox.



Activation

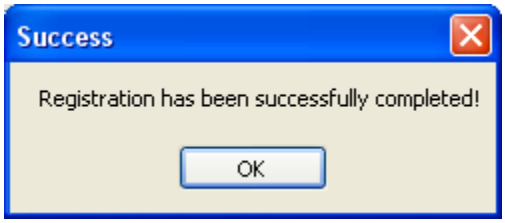
The version of Delsi 2.0 available for download is a trial version. This version has some capacity limitation: a model cannot contain more than 10 model items and 1000 transactions at a time.

After the license fee is paid, you will receive a registration code for activating Delsi 2.0 to make it fully functional (which means that all the limitations will be revoked). To activate Delsi 2.0, run application <*DELSI_HOME*>\BIN\Activation.exe, where <*DESLI_HOME*> is Delsi home directory chosen on installation.



Enter your fist and last name, the company name (if applicable), the registration code and press button "Activate".

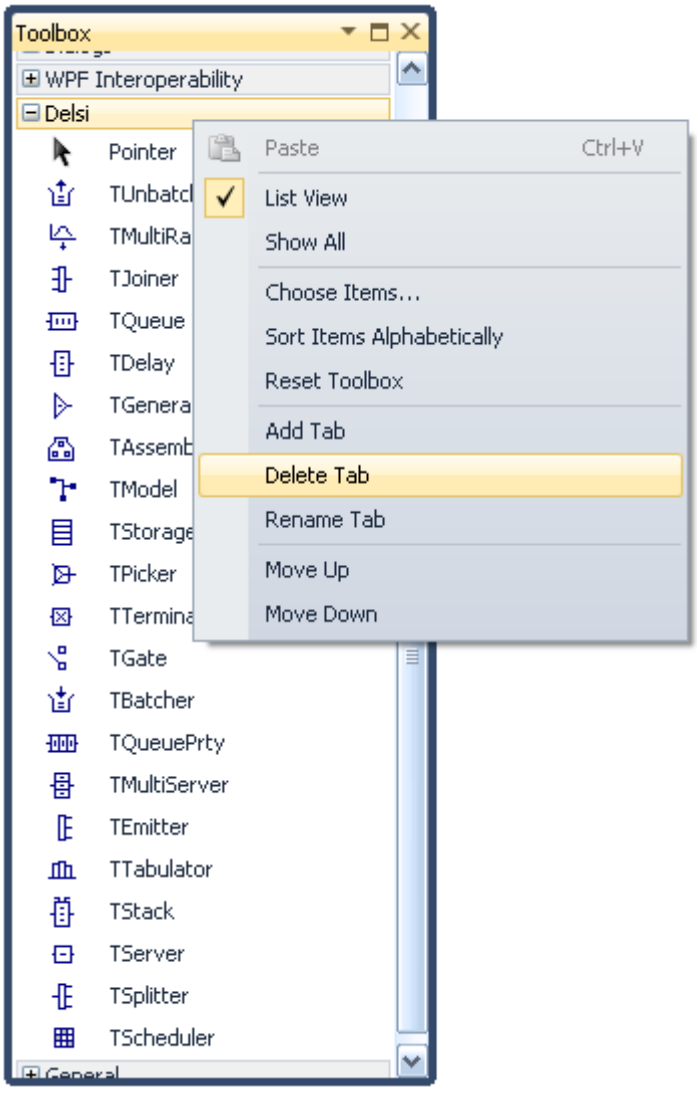
On successful activation you will receive the following confirmation message.



NOTE: If you decided to move and activate Delsi 2.0 on another computer, you have to inactivate Delsi 2.0 on your current computer according to Delsi 2.0 license agreement. To do so, run application **Activation.exe** ans press button "Inactivate".

Uninstalling

1. Start Microsoft Visual Studio 2010
2. In its top menu "View" choose option "Toolbox".
3. Right click on the tab "Delsi" (which you created previously) and choose option "Delete".



4. To remove Delsi files from your computer, go to "Control Panel" >> "Add or Remove Programs", find "Delsi 2.0" and press button "Remove".

CHAPTER 1. Basics

Sample 1. Barbershop

This example is taken from the well-known “Red Book” of Thomas J. Schriber "Simulation Using GPSS". Let's imagine a barbershop with one barber. Customers arrive to the barbershop and if the barber is busy, they will wait in the waiting room. They will get the service according to the rule "First come - first served" (FIFO). After the service done 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:

- Fraction of time the barber was busy (usage)
- Average queue length
- Maximum 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

To build a model we are using the following components:

Arrival	*TGenerator*	Arrival of customers
WaitingRoom	*TQueue*	Waiting for service with FIFO discipline
Barber	*TServer*	The barber
Exit	*TTerminator*	Leaving the barbershop
tModel1	*TModel*	The model
tMultiRand1	*TMultiRand*	Random number generator

Results are output into the textbox. However in real projects you can output the results in any format you want: ASCII file, database, canvas, HTML, etc.

Before writing the code, we added a namespace called *Delsi* to the project *References* and to the directive *Using*.

```
using Delsi;
```

Before we start the simulation run we need to add our model items to the model.

```
private void Form1_Load(object sender, EventArgs e)
{
      tModel1.Add(Arrival);
      tModel1.Add(WaitingRoom);
      tModel1.Add(Barber);
      tModel1.Add(Exit);
}
```

To define time intervals between arrivals, we use the *SetTime* method in the *OnExit* event of generator *Arrival*.

```
private void Arrival_OnExit(TGenerator sender,
                            Transaction transaction)
{
    sender.SetTime(tMultiRand1.Uniform(12.0, 24.0));
}
```

To define service time, we use the *SetTime* method in the event *OnEnter* of server *Barber*.

```
private void Barber_OnEnter(TServer sender,
                            Transaction transaction)
{
    sender.SetTime(tMultiRand1.Uniform(14.0, 20.0));
}
```

In order to route transactions from block to block, we use the *Send* method in the *OnRouting* event of blocks *Arrival*, *WaitingRoom* and *Barber*.

```
private void Arrival_OnRouting(TGenerator sender,
                               Transaction transaction)
{
    sender.Send(WaitingRoom);
}

private void WaitingRoom_OnRouting(TQueue sender,
                            Transaction transaction)
{
    sender.Send(Barber);
}

private void Barber_OnRouting(TServer sender,
                              Transaction transaction)
{
    sender.Send(Exit);
}
```

We define how to manage our experiment in the *OnClick* event of button *button1*. We will start simulation and run it until simulation time reaches 480.0. Then we will output the results into the text box. Finally we will reset the model to make it ready for the next run.

```
private void button1_Click(object sender, EventArgs e)
{
    tModel1.Simulate(480.0);
    textBox1.Text = tModel1.Report();
    tModel1.Reset();
}
```

The results are combined and formatted by method *Report* of component *tModel1*. All of the values in the results can be obtained using the methods of the blocks. So, you can write your own code to format the results.

```
System Time: 475.528475799488

Arrival (TGenerator)
-----------------------------------
Exits: 27
Count: 0
Average time: 18.2895567615188
Deviation time: 3.54829534356622


WaitingRoom (TQueue)
-----------------------------------
Entries: 27
Zero Entries: 27
Exits: 27
Exits by time limit: 0
Count: 0
Max count: 1
Average count: 0.1058978572661
Usage: 0.1058978572661
Min time: 0
Max time: 11.2237012379858
Average time: 1.86509061689556
Deviation time: 2.60237940895093
Average non-zero time: 3.14734041601125
Deviation non-zero time: 2.71895425693285


Barber (TServer)
-----------------------------------
Entries: 27
Exits: 26
Exits served: 26
Exits by interruption: 0
Count: 1
Count on service: 1
Max count: 1
Average count: 0.930545085534549
Average count on service: 0.930545085534549
Usage: 0.930545085534549
Min time: 14.1220948696421
Max time: 19.9046571268478
Average time: 17.0192571610365
Deviation time: 1.6205265462961
Min time served: 14.1220948696421
Max time served: 19.9046571268478
Average time served: 17.0192571610365
Deviation time served: 1.6205265462961


Exit (TTerminator)
-----------------------------------
Entries: 26
Exits: 0
Count: 0
```

Sample 2. Input parameters and progress bar

In this sample we demonstrate how to edit input parameters and how to enable the progress bar. Here we will use the same model as in Sample 1. The arrival and service intervals are exponentially distributed. The following input parameters will be edited with the *TextBox* component:

- Average arrival interval
- Average service time
- End time of simulation

We store the values of the parameters in the private variable of the form.

```
private double meanArrivalTime;    // Mean of the arrival time
private double meanServiceTime;    // Mean of the service time
private double endTime;            // Simulation end time
```

On the event of pressing the “Start” button we parse and validate the input parameters and then start the simulation run.

```
private void button1_Click(object sender, EventArgs e)
{
    // Parse and validate
    try
    {
        meanArrivalTime = Double.Parse(textBoxArrivalTime.Text);
        meanServiceTime = Double.Parse(textBoxServiceTime.Text);
        endTime = Double.Parse(textBoxEndTime.Text);
    }
    catch
    {
        MessageBox.Show("Invalid parameters");
        return;
    }

    if (meanArrivalTime <=0 || meanServiceTime<=0 || endTime<=0 )
    {
        MessageBox.Show("Invalid parameters");
        return;
    }

    // Start simulation run
    tModel1.Simulate(endTime);
    // Output the results
    textBox1.Text = tModel1.Report();
    // Reset the model
    tModel1.Reset();
}
```

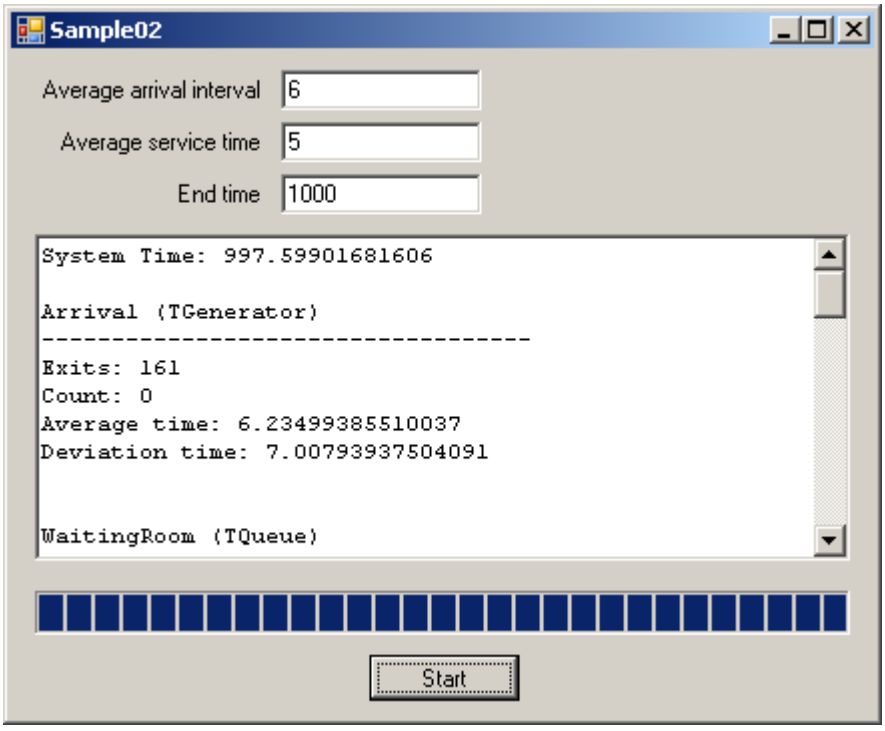
We use stored values of the input parameters to set the service time and the time between arrivals.

```
private void Arrival_OnExit(TGenerator sender, Transaction transaction)
{
    sender.SetTime(tMultiRand1.Exponential(meanArrivalTime));
}

private void Barber_OnEnter(TServer sender, Transaction transaction)
{
    sender.SetTime(tMultiRand1.Exponential(meanServiceTime));
}
```

In order to display the progress of a simulation run we use the *ProgressBar* component. We change property *progressBar1.Value* on handling the *OnNewTime* event of the *tModel1* component. This event is fired when new value of *system time* is taken from the *List of Future Event*s.

```
private void tModel1_OnNewTime(TModel model,
                               IModelItem modelitem,
                               Transaction transaction)
{
    double ratio;
    ratio = model.SysTime / endTime;
    progressBar1.Value = (int)Math.Round(100 * ratio);
}
```

The user interface looks like the following.



Sample 3. Clearing statistics during simulation run

This sample demonstrates how to clear statistics during a simulation run. It may be useful when you want to detect the “warm-up period” of the stochastic process of the model.

To schedule the clearing of the statistics we use a *TScheduler* component. On handling its *OnPlanned* event we execute the following steps:

- Output the results into the *textBox1* component
- Clear statistics
- Order the next *OnPlanned* event for the *tScheduler1* component

In this example the results we are interested in are: average time spent in the queue and average time spent in the server. Here is how it looks like in source code.

```
private void tScheduler1_OnPlanned(TScheduler sender)
{
    // Output the results
    textBox1.AppendText(tModel1.SysTime.ToString("0.000   "));
    textBox1.AppendText(WaitingRoom.AverageTime().ToString("0.000          "));
    textBox1.AppendText(Barber.AverageTime().ToString("0.000"));
    textBox1.AppendText(System.Environment.NewLine);
    // Clear statustics
    tModel1.ClearStatistics();
    // Clear next OnPlanned event
    sender.SetTime(clearenceInterval);
}
```

The results look like this.

```
SysTime     AveTime(Queue)    AveTime(Server)
1000.000    12.986            4.758
2000.000    20.047            4.924
3000.000    17.600            4.828
4000.000    20.488            4.853
5000.000    23.444            4.808
6000.000    21.416            4.806
7000.000    22.076            4.908
8000.000    24.186            5.006
9000.000    23.457            4.983
10000.000    25.758            5.015
```

Sample 4. Changing parameters between runs

This sample demonstrates the basic technique for managing simulation experiments. In these experiments both inter-arrival and service times are exponentially distributed. The mean of service time is 10.0. The mean of inter-arrival time is changing from 10.0 to 15.0 with the step of 1.0. Our goal is to observe how the average queue length depends on the average arrival time.

Here is the fragment of the source code which demonstrates how to manage the experiments.

```
private void button1_Click(object sender, EventArgs e)
 {
     meanArrivalTime = 10.0;

     textBox1.Clear();
     textBox1.AppendText("Average arrival interval    ");
     textBox1.AppendText("Average queue length");
     textBox1.AppendText(System.Environment.NewLine);

     for (int i = 0; i < 5; i++)
     {
         // Execute simulation run
         tModel1.Simulate(10000.0);

         // Output result for one sumulation run
         textBox1.AppendText(meanArrivalTime.ToString("0.000"));
         textBox1.AppendText("                        ");
         textBox1.AppendText(WaitingRoom.AverageCount().ToString("0.000"));
         textBox1.AppendText(System.Environment.NewLine);
         tModel1.Reset();

         // Increment average arrival time
         meanArrivalTime += 1.0;
     }
 }

 private void Arrival_OnExit(TGenerator sender, Transaction transaction)
 {
     sender.SetTime(tMultiRand1.Exponential(meanArrivalTime));
 }

 private void Barber_OnEnter(TServer sender, Transaction transaction)
 {
     sender.SetTime(tMultiRand1.Exponential(10.0));
 }
```

The results look like the following.

```
Average arrival interval    Average queue length
10.000                      7.515
11.000                      4.186
12.000                      3.817
13.000                      2.111
14.000                      2.122
```

Sample 5. Tracing

In this sample we demonstrate how to trace transaction passing from block to block. For this purpose we use the *TModel.AfterPass* event.

```
private void tModel1_AfterPass_1(TModel model,
                                 IBlock sender,
                                 IBlock receiver,
                                 Transaction transaction)
{
    textBox1.AppendText(model.SysTime.ToString("0.0000").PadRight(10,' '));
    textBox1.AppendText("  ");
    textBox1.AppendText(sender.Caption.PadRight(12, ' '));
    textBox1.AppendText("  ");
    textBox1.AppendText(receiver.Caption.PadRight(12, ' '));
    textBox1.AppendText("  ");
    textBox1.AppendText(transaction.Id.ToString());
    textBox1.AppendText(System.Environment.NewLine);
}
```

The result of the tracing looks like this.

```
SysTime     Sender        Receiver      Transaction
0.0000      Arrival       WaitingRoom   1
0.0000      WaitingRoom   Barber        1
3.8700      Barber        Exit          1
35.1236     Arrival       WaitingRoom   2
35.1236     WaitingRoom   Barber        2
40.5349     Arrival       WaitingRoom   3
41.0784     Barber        Exit          2
41.0784     WaitingRoom   Barber        3
43.7361     Barber        Exit          3
44.1646     Arrival       WaitingRoom   4
44.1646     WaitingRoom   Barber        4
47.5415     Barber        Exit          4
69.2100     Arrival       WaitingRoom   5
69.2100     WaitingRoom   Barber        5
71.9152     Barber        Exit          5
74.0059     Arrival       WaitingRoom   6
74.0059     WaitingRoom   Barber        6
82.2736     Barber        Exit          6
86.0441     Arrival       WaitingRoom   7
86.0441     WaitingRoom   Barber        7
86.2043     Barber        Exit          7
```

Using the *AfterPass* and *OnNewTime* events you can create more sophisticated tracing. For instance, you can output the queue lengths or transaction priorities. And of course, you can customize the look and feel of the output.

Sample 6. Changing parameters during a run

In this sample we use the model scheme of Sample 1. Let's consider the fact that usually the arrival rate is not the same all day long. We need to to change it during the simulation run. In this sample, the service time is normally distributed with *mean* 10 min. and *standard deviation* 2 min. The inter-arrival time is exponentially distributed, it changes its *mean* value according to the following table.

Period (hrs)	Period since opening (min.)	Average inter-arrival interval (min.)
8.00 - 10.00	0-120	14
10.00 - 12.00	120-240	12
12.00 - 15.00	240-420	10
15.00 - 17.00	420-540	12
17. - 20.00	540-720	10

To keep those values we use two arrays.

```
double[] arrivalMeans = {  14.0, 12.0, 10.0, 12.0, 10.0 };
double[] timeEffective = { 0.0, 120.0, 240.0, 420, 540.0 };
// Indicates array index in use
int Index = 0;
```

The values for intergeneration intervals are taken from an array called *arrivalMeans*.

```
 private void Arrival_OnExit(TGenerator sender, Transaction transaction)
 {
     sender.SetTime(tMultiRand1.Exponential(arrivalMeans[Index]));
 }
```

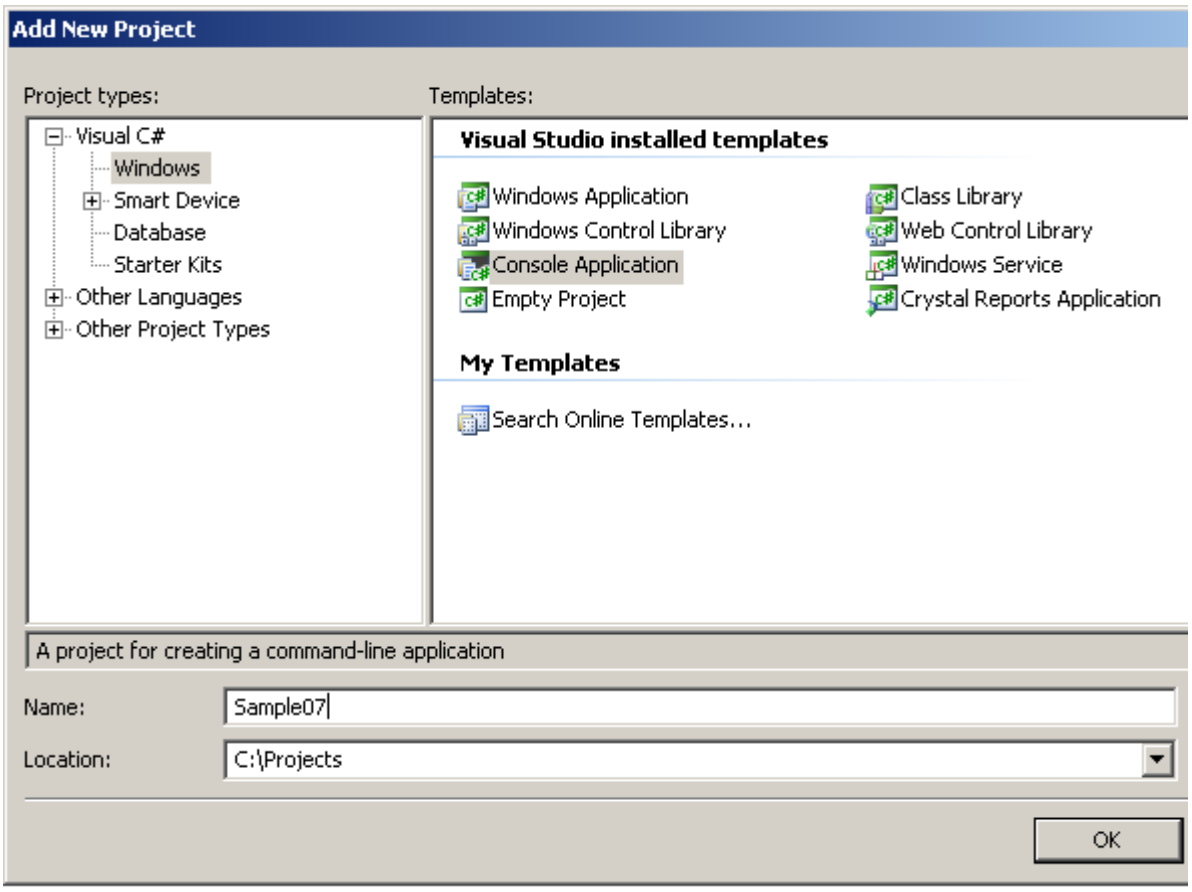
To change the arrival rate in the specified moments of system time we use a *TScheduler* component.

```
private void tScheduler1_OnPlanned(TScheduler sender)
{
    if (tModel1.SysTime > sender.FirstTime) Index++;
    if (Index < 4)
    {
         sender.SetTime(timeEffective[Index + 1] - tModel1.SysTime);
    }
}
```

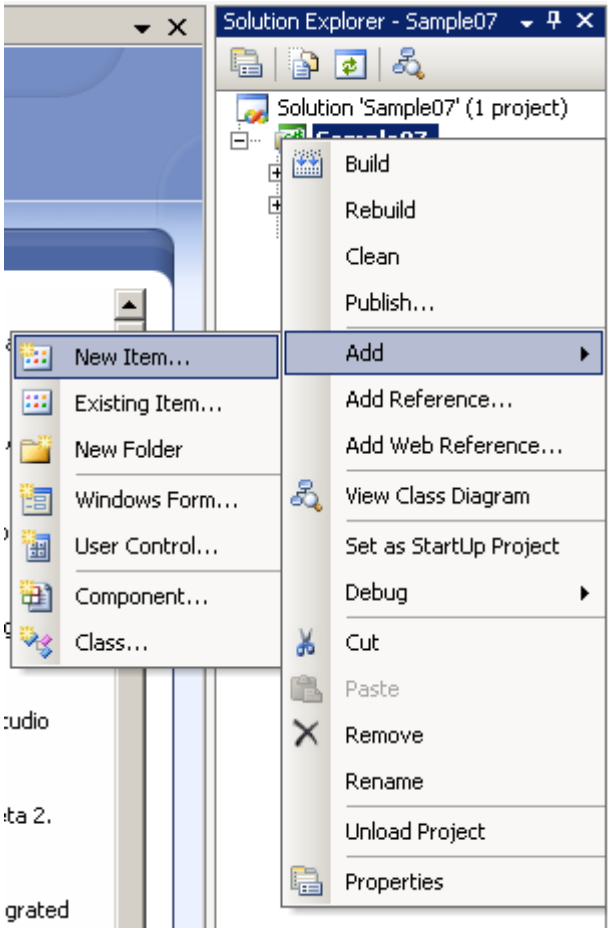
Sample 7. Wrapping into a custom component

In this sample we demonstrate how to wrap our model into a custom component and how to call the component from the console application.

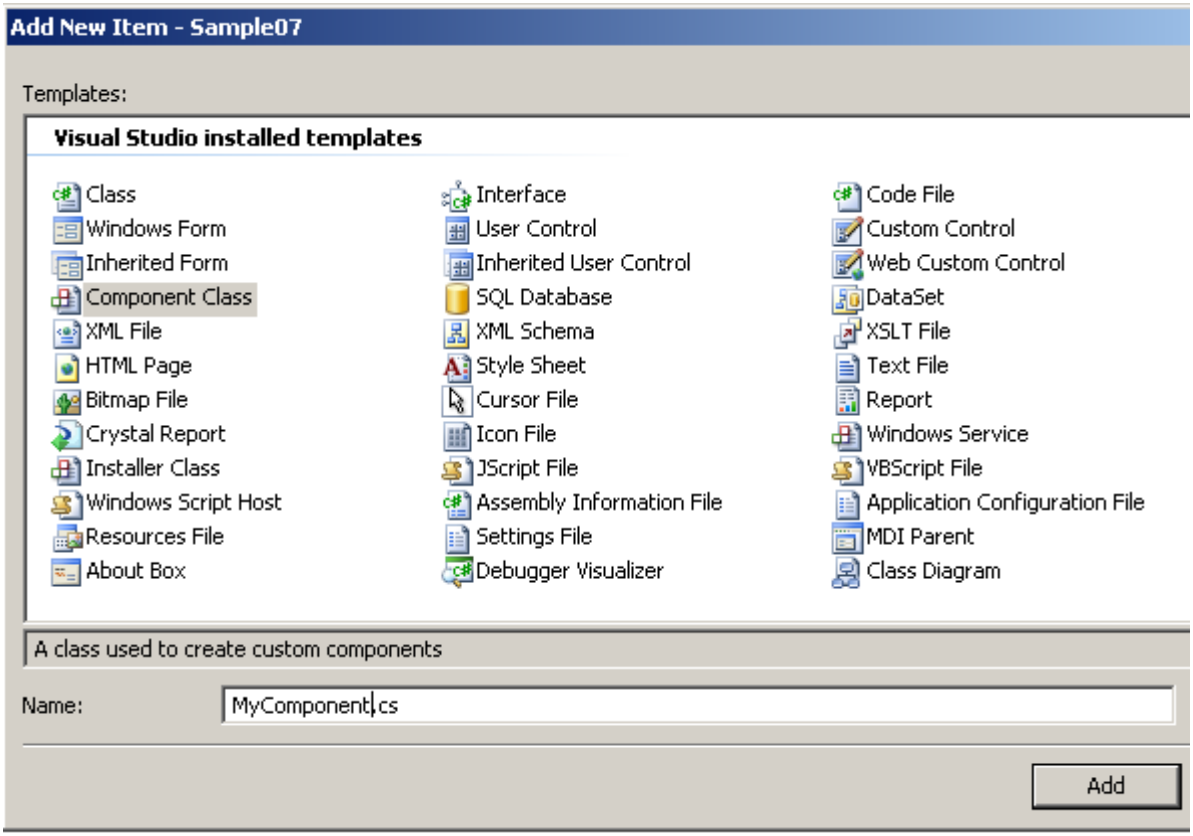
First, we created console application *Sample07*.



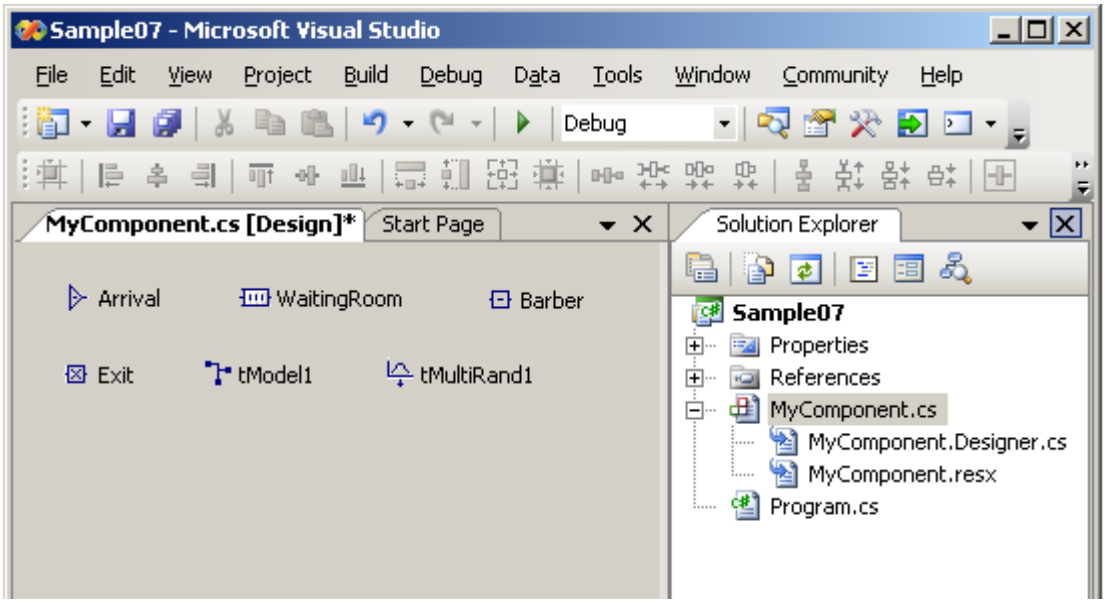
Next we added new item to the project.



This item is *Component Class* named *MyComponent*.



In the [Design] mode of viewing this component we drag-and-drop *Delsi* components from *Toolbox*.



Then we setup properties and events for Delsi components like we do it in Windows Desktop applications.

It is always our responsibility to add model items to the model. For that we have developed the private method named *PopulateModel*. We call this method from the constructors of the *MyComponent* class. That's how we do it in the file *MyComponent.cs*

```
public MyComponent()
{
    InitializeComponent();
    PopulateModel();
}

public MyComponent(IContainer container)
{
    container.Add(this);
    InitializeComponent();
    PopulateModel();
}

// This method populates model with the model items
private void PopulateModel()
{
    tModel1.Add(Arrival);
    tModel1.Add(WaitingRoom);
    tModel1.Add(Barber);
    tModel1.Add(Exit);
}
```

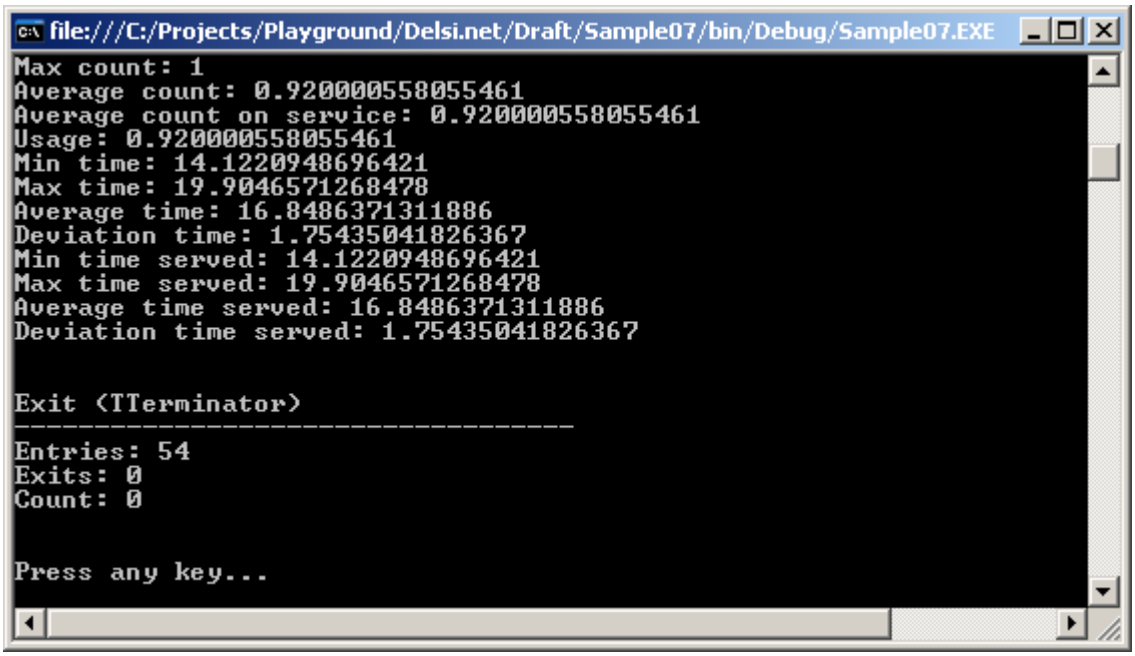
In order to perform a simulation run, our custom component exposes a method called *Run*. This method takes the end time of simulation as an input parameter and returns the text with the standard model report.

```
public string Run(double endTime)
{
    tModel1.Simulate(endTime);
    string result = tModel1.Report();
    tModel1.Reset();
    return result;
}
```

In the main program of our console application *Program.cs,* we instantiate the component, call its method *Run* and output the results into the console.

```
static void Main(string[] args)
{
    MyComponent component = new MyComponent();
    Console.Write(component.Run(1000));
    Console.Write("Press any key...");
    Console.ReadKey(true);
}
```

That's what we see in the console.



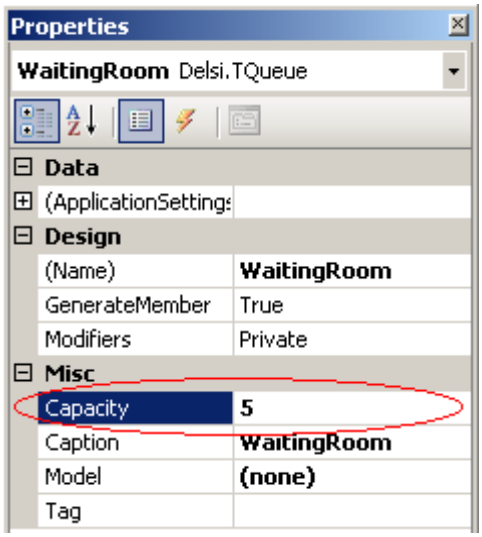
We just learned how to wrap our simulation components into a custom component. The impact of this idea is very promising. Using the demonstrated approach we can create a custom *class library* and run the simulation experiments via different kinds of interfaces: Windows Desktop applications, Web-applications, Web Services, Windows Communication Foundation, COM+, and console applications.

CHAPTER 2. Simulation Essentials

Sample 8. Limited queue capacity and routing

Imagine that the waiting room in the barbershop has only 5 chairs. If a customer walks in the barbershop and finds all the chairs are occupied, she will go find another barbershop. In this example we presume that the time between arrivals is exponentially distributed with the *mean* 11 min, and the service time is normally distributed with mean 10 min and standard deviation 2 min. We need to determine the percentage of customers lost.

We define capacity of the queue equal to 5 in its property *Capacity*.



Of course, it can be done programmatically as well.

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.

```
private void Entrance_OnRouting(TGenerator sender, Transaction transaction)
{
    if (WaitingRoom.isReadyToGet(transaction))
    {
        sender.Send(WaitingRoom);
    }
    else
    {
        sender.Send(AnotherShop);
    }
}
```

After the simulation run is completed, it is very easy to calculate the percentage of the lost customers and output the results.

```
double lost = 100.0 * AnotherShop.Entries() / Entrance.Exits();
```

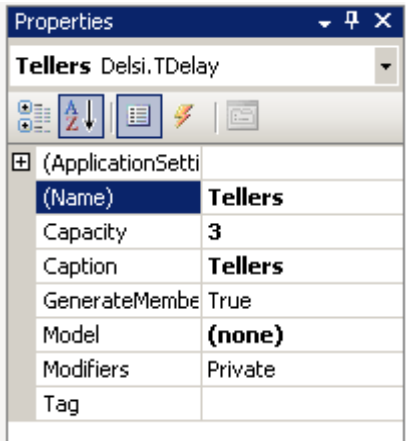
Here are the results of the simulation.

```
Total customers: 47
Lost customers: 4
Fraction of lost customers (%): 8.51
```

Sample 9. Bank tellers. Using TDelay

Imagine a bank branch where three tellers serve customers. Customers form one queue. From the queue they come to the first available teller. Inter-arrival time is exponentially distributed with mean 4 min. Service time at tellers is normally distributed with mean 10 min. and standard deviation 3 min. The bank branch is open for 8 hours.

The simulation program for this sample is similar to the one in Sample 1. But instead of *TServer*, we use *TDelay* component with capacity 3.



The standard report looks like this.

```
System Time: 478.738236495977

Arrival (TGenerator)
------------------------------------
Exits: 125
Count: 0
Average time: 3.80271365012305
Deviation time: 3.93149450966144


QueueToTellers (TQueue)
------------------------------------
Entries: 125
Zero Entries: 42
Exits: 125
Exits by time limit: 0
Count: 0
Max count: 11
Average count: 2.69932023484604
Usage: 0.659705003280621
Min time: 0
Max time: 34.8564611219401
Average time: 10.3381424717448
Deviation time: 9.03324991133789
Average non-zero time: 13.4611230100844
Deviation non-zero time: 8.01315537969416


Tellers (TDelay)
------------------------------------
Entries: 125
Exits: 122
Count: 3
Max count: 3
Average count: 2.65711768075142
Usage: 0.993234631346067
Min time: 1.55396695354591
Max time: 18.0288615431477
Average time: 10.3832188303248
Deviation time: 3.07510313287262


Exit (TTerminator)
------------------------------------
Entries: 122
Exits: 0
Count: 0
```

Can we model tellers using *TMuliServer*? Yes, we can. Then why do we use *TDelay* here? The reason is that *TMultiServer* is a heavy-weighted component designed mostly for the scenarios with high-priority interruptions. For the simple cases like this, it is more efficient to use *TDelay.*

Sample 10. Call center. Limited waiting time in the queue

In this example we are modeling a call center. The call center has 5 customer service representatives. The telephone system has a calling queue with the ability to hold 10 calls.

If all reps are busy, the next customer call will be placed into the calling queue (and the customer starts to hear the annoying phrase "stay on the line, your call is very important for us..."). When a customer gets tired of waiting, she hangs up. Each customer has a different level of patience; one can wait longer than another before choosing to hang up. The tolerated amount of waiting time in the queue is normally distributed with mean 25 min. and standard deviation 6 min.

If the calling queue runs out of capacity, the customer will get busy signal when trying to call.

The time interval between sequential calls is exponentially distributed with mean 4 min. The duration of a conversation with a rep is normally distributed with mean 10 min. and standard deviation 3 min. The call center is open for 8 hours.

Besides getting the standard report, our goal is to determine how many calls were successfully served, how many customers hanged up and how many customers got busy signal.

In this solution those outcomes are modeled with three components:

EndOfService	End of Service
BusySignal	Busy Signal
HangedUp	Hanged up the line

Similar to previous sample, we use component *TDelay* to model the reps.

Like in Sample 8, we handle limited queue capacity by sending a transaction to the different block.

```
private void IncomingCalls_OnRouting(TGenerator sender, Transaction transaction)
{
    if (CallingQueue.isReadyToGet(transaction))
    {
        sender.Send(CallingQueue);
    }
    else
    {
        sender.Send(BusySignal);
    }
}
```

In order to limit waiting time in the queue (to simulate customer impatience) we use method *SetTime* in the event *OnEnter* of *CallingQueue*.

```
private void CallingQueue_OnEnter(TQueue sender, Transaction transaction)
{
    sender.SetTime(tMultiRand1.Normal(25.0, 6.0));
}
```

When the waiting time for a transaction reaches its limit, Delsi fires the *AfterTimeEnded* event for the queue. In this event we send the transaction to terminator *HangedUp*..

```
private void CallingQueue_AfterTimeEnded(TQueue sender, Transaction transaction)
{
    sender.Send(HangedUp);
}
```

Here are the results.

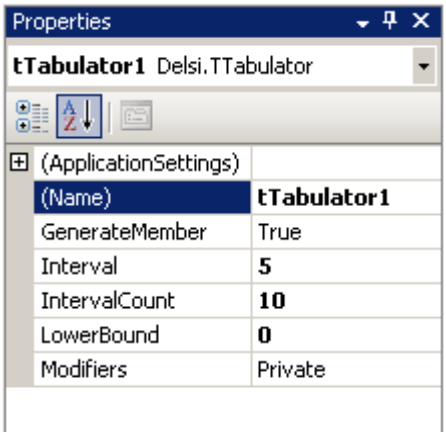
```
Successfully received the service: 163
Got busy signal: 17
Hanged up the line: 45


System Time: 479.14272445682

Incoming Calls (TGenerator)
-----------------------------------
Exits: 239
Count: 0
Average time: 2.00781535561782
Deviation time: 2.01537397417941


Calling Queue (TQueue)
-----------------------------------
Entries: 222
Zero Entries: 18
Exits: 213
Exits by time limit: 45
Count: 9
Max count: 10
Average count: 5.44131378157454
Usage: 0.911668226775435
Min time: 0
Max time: 27.0282950775604
Average time: 11.8490025081333
Deviation time: 6.5017404145208
Average non-zero time: 12.3717526187862
Deviation non-zero time: 6.13761046891684


Reps (TDelay)
-----------------------------------
Entries: 168
Exits: 163
Count: 5
Max count: 5
Average count: 4.91197354494776
Usage: 1
Min time: 1.83225665092964
Max time: 35.5613466896346
Average time: 14.1335913228919
Deviation time: 7.49053225228747
```

```
End Of Service (TTerminator)
------------------------------------
Entries: 163
Exits: 0
Count: 0


Busy Signal (TTerminator)
------------------------------------
Entries: 17
Exits: 0
Count: 0


Hanged Up (TTerminator)
------------------------------------
Entries: 45
Exits: 0
Count: 0
```

Sample 11. Custom transaction fields. Tabulation.

The objective of this sample is to get the statistics about the total time the customers spend to get served in the call center (discussed in previous sample). To get this total we need to record the birth time of a transaction. When a transaction leaves the model we subtract the birth time from the current system time. Obviously, the most convenient place to store the birth time is in the transaction itself.

To collect the statistics about total time in the system for successfully served customers, we use component *TTabulator*.



We will tabulate the total time in the system using the range from 0 to 50 min. with a 5 min. interval.

In order to store information of an arbitrary nature in a transaction, its property *Tag* can contain the reference to the object of any type.
To store the birth time we declare class *MyClass*, containing only one private field *_birthtime* exposed through the property *BirthTime*.

```
public class MyClass
{
    private double _birthtime;

    public MyClass(double time)
    {
        _birthtime = time;
    }

    public double BirthTime
    {
        get { return _birthtime; }
    }
}
```

After the transaction is generated, Delsi Runtime Engine fires the event *TGenerator.AfterGeneration*. In this event we instantiate new instance of *MyClass* and assign it to the transaction field *Tag*.

```
private void IncomingCalls_AfterGeneration(TGenerator sender,
                                           Transaction transaction)
{
    MyClass myclass = new MyClass(tModel1.SysTime);
    transaction.Tag = myclass;
}
```

When a transaction enters block *EndOfServive* we calculate and tabulate the total time spent in the system.

```
private void EndOfService_OnEnter(TTerminator sender,
                                  Transaction transaction)
{
    double time_in_the_system = tModel1.SysTime -
                               ((MyClass)transaction.Tag).BirthTime;
    tTabulator1.Tabulate(time_in_the_system);
}
```

These are the results of the simulation.

```
Total Served: 163
Average time in the system: 25.898
Deviation time in the system: 9.786

--- Histogram data ---
Below low bound: 0
Interval 1: 0
Interval 2: 5
Interval 3: 16
Interval 4: 27
Interval 5: 28
Interval 6: 40
Interval 7: 20
Interval 8: 10
Interval 9: 7
Interval 10: 8
Above upper bound: 2
```

Sample 12. Subway station. Using TEmitter

In this sample we will build the model of a subway (metro) station. Our subway station has one booth with a cashier and 6 token validators. Some of the passengers arrive to the station by bus, others come on foot. The inter-arrival time for the customers coming on foot is exponentially distributed with mean 12 sec. The buses arrive at the station every 10 min. The number of passengers in the bus is uniformly distributed between 40 and 70.

Regardless of the way the passengers arrive, they can be divided into three types.

Type	Description	Part, %	Distribution of time to get through
1	Use tokens and go through validators	80	Uniform from 2 to 4 sec.
2	Show the pass to the cashier	10	Uniform from 3 to 5 sec.
3	Pay cash to the cashier	10	Uniform from 10 to 20 sec.

We need to simulate 2 hours of the station's functioning. The first bus arrives 1 min. after the beginning of time. We need to get the statistical data for the queues in the system.

To simulate the arrival of customers on foot we use *TGenerator* component *ArrivalOnFoot*.

```
private void ArrivalOnFeet_OnExit(TGenerator sender, Transaction transaction)
{
    sender.SetTime(tMultiRand1.Exponential(12));
}
```

For customers arriving by bus we use the combination of components *ArrivalByBus* (*TEmitter*) and *BusScheduler* (*TScheduler*). The scheduler calls the emitter every 600 sec. asking to emit the amount of transactions uniformly distributed between 40 and 70.

```
private void BusScheduler_OnPlanned(TScheduler sender)
{
    int amount = (int)Math.Round(tMultiRand1.Uniform(40, 70));
    ArrivalByBus.Emit(amount);
    sender.SetTime(600);
}
```

The queues to the cashier and to the validators are represented by TQueue components *QueueCashier* and *QueueValidators.* We route transactions from generator *ArrivalOnFoot* and emitter *ArrivalByBus* to queues QueueCashier and QueueValidators in the proportion 20 to 80.

```
private void ArrivalOnFeet_OnRouting(TGenerator sender, Transaction transaction)
{
    if (tMultiRand1.Uni01() < 0.8)    // 80% use validators
    {
        sender.Send(QueueValidators);
    }
    else
    {
        sender.Send(QueueCashier);    // the rest go to the cashier
    }
}

private void ArrivalByBus_OnRouting(TEmitter sender, Transaction transaction)
{
    if (tMultiRand1.Uni01() < 0.8)    // 80% use validators
    {
        sender.Send(QueueValidators);
    }
    else
    {
        sender.Send(QueueCashier);    // the rest go to the cashier
    }
}
```

The cashier is modeled by *TServer* component *Cashier* and the validators are modeled by *TDelay* component *Validators* with the capacity of 6 which has been set at the design time.

At the cashier booth, half of the customers will show pass and another half will pay cache.

```
private void Cashier_OnEnter(TServer sender, Transaction transaction)
{
    if (tMultiRand1.Uni01() < 0.5)    // 10% show pass
    {
        sender.SetTime(tMultiRand1.Uniform(3, 5));
    }
    else                              // 10% pay cache
    {
        sender.SetTime(tMultiRand1.Uniform(10, 20));
    }
}
```

Finally, the transactions go to terminator *GotInside* which represents the fact that a customer got inside the subway.

The results are the following.

```
Queue to get to the cashier
---------------------------
Entries: 255
Zero Entries: 121
Exits: 255
Exits by time limit: 0
Count: 0
Max count: 16
Average count: 1.36454932964952
Usage: 0.215482717979611
Min time: 0
Max time: 211.678258914425
Average time: 38.522247797949
Deviation time: 48.5148491466833
Average non-zero time: 62.9690589004935
Deviation non-zero time: 48.0415045255149


Queue to get to the validators
------------------------------
Entries: 1032
Zero Entries: 544
Exits: 1032
Exits by time limit: 0
Count: 0
Max count: 57
Average count: 0.812035696872709
Usage: 0.0365380170488058
Min time: 0
Max time: 30.5855870271544
Average time: 5.66445308820928
Deviation time: 7.38523312583074
Average non-zero time: 11.6680949840958
Deviation non-zero time: 6.50370978112345
```

Sample 13. Photo Lab. Using TQueuePrty

In this example we will discuss a photo lab. This photo lab works 7 days a week, 12 hours per day. There is one printing machine in the lab. 40% of all orders are urgent, other 60% are non-urgent. The urgent orders have to be completed within 1 business hour and non-urgent within 24 business hours.

The urgent orders have a higher priority, so they get processed first. If a non-urgent order waits for printing 23 hours, it gets the priority of an urgent order. Let's call this procedure an "escalation".

The time between order arrivals is distributed exponentially with mean 2.4 min. The processing time is distributed normally with mean 3 min and standard deviation 0.5 min. We will simulate the work of the photo lab for one week. We want to get the statistical data about the overdue time for both types of orders.

To carry necessary information with a transaction, we create our custom class *Order*.

```
public class Order
{
    public double TimeIssued;
    public int OrderType;    // 1 - Urgent, 2 - Non-Urgent
}
```

Each time when generator *Arrival* generates a new transaction, we create the new instance of the *Order* class. Next we record the current value of the system time and the order type in the *Order* instance. Finally, we assign the *Order* instance to the transaction property *Tag*.

```
private void Arrival_AfterGeneration(TGenerator sender, Transaction transaction)
{
    Order order = new Order();

    if (tMultiRand1.Uni01() < 0.4)
    {
        transaction.Priority = 1;
        order.OrderType = 1;  // Indicates type "Urgent"
    }
    else
    {
        transaction.Priority = 0;
        order.OrderType = 2;  // Indicates type "Non-urgent"
    }

    order.TimeIssued = tModel1.SysTime;
    transaction.Tag = order;
}
```

The queue of orders is ordered according order priorities. To reflect that fact, we use a *TQueuePrty* component *QueueProcessing*.

The escalation mechanism is implemented the following way. First, we limit waiting time in the queue for non-urgent orders by 23 hours.

```
private void QueueProcessing_OnEnter(TQueuePrty sender, Transaction transaction)
{
    if (((Order)transaction.Tag).OrderType == 2)
    {
        sender.SetTime(60 * 23);
    }
}
```

Second, if the waiting time for non-urgent transaction reaches the limit, we send the transaction to a queue named *Escalation*.

```
private void QueueProcessing_AfterTimeEnded(TQueuePrty sender,
                                            Transaction transaction)
{
    sender.Send(Escalation);
}
```

Third, when the transaction enters the queue *Escalation*, we increase its priority to 1.

```
private void Escalation_OnEnter(TQueue sender, Transaction transaction)
{
    transaction.Priority = 1;
}
```

Finally, we send the transaction back to *QueueProcessing*. There it will be last among the transactions with priority 1, but it will procede all transactions with priority 0.

```
private void Escalation_OnRouting(TQueue sender, Transaction transaction)
{
    sender.Send(QueueProcessing);
}
```

Why do we actually need the queue *Escalation* at all? One of the reasons is to overcome Delsi limitation that a block cannot send a transaction to itself. So, *Escalation* serves as an intermediate queue.

After processing a transaction in a *TServer* component *Processing*, we send it to one of two *TTerminator* components (*EndUrgent* and *EndNonUrgent*) depending on the order type.

```
private void Processing_OnRouting(TServer sender, Transaction transaction)
{
    if (((Order)transaction.Tag).OrderType == 1)
    {
        sender.Send(EndUrgent);
    }
    else
    {
        sender.Send(EndNonUrgent);
    }
}
```

In the terminators we detect and tabulate the amount of time overdue for each order type.

```
private void EndUrgent_OnEnter(TTerminator sender, Transaction transaction)
{
    double overdue = tModel1.SysTime - ((Order)transaction.Tag).TimeIssued - 60;
    if (overdue > 0)
    {
        tTabulator1.Tabulate(overdue);
    }
}

private void EndNonUrgent_OnEnter(TTerminator sender, Transaction transaction)
{
    double overdue = tModel1.SysTime - ((Order)transaction.Tag).TimeIssued -
                     60 * 24;
    if (overdue > 0)
    {
        tTabulator2.Tabulate(overdue);
    }
}
```

Here are the results of the simulation.

```
Urgent orders:
   - Number of completed orders: 810
   - Number of overdue orders: 109
   - Average overdue, min.: 77.947
   - Standard deviation of overdue, min.: 54.571

Non-urgent orders:
   - Number of completed orders: 878
   - Number of escalated orders: 304
   - Number of overdue orders: 165
   - Average overdue, min.: 94.529
   - Standard deviation of overdue, min.: 52.899
```

Sample 14. Failures and recoveries. Method TServer.Pause

In this sample, let's examine a workshop which has a machine tool and a box for workpieces. The workpieces are delivered into the box by the conveyor belt with the time interval having normal distribution with the mean 10 min. and the standard deviation 1 min.

The worker processes the workpieces out of the box in LIFO order. The processing time has normal distribution with mean 9 min. and standard deviation 1.5 min. After the processing is complete the workpieces gets moved to another workshop.

From time to time the machine tool breaks. In this case the worker removes the workpiece and calls the repair service. After the recovery, the processing of the detail continues. The total time of the actual processing of the workpiece is not affected.

The failure happens 4 hours after the shift starts. The recovery takes 45 min. The duration of the shift is 8 hours. Frankly speaking, this time we don't have any serious goals except of demonstrating the method *Pause*. Besides that, the LIFO order provides us with perfect reason to introduce a component *TStack*.

In real life, the systems with failures and recoveries have an important place in many technical applications. In this simple example we demonstrate how Delsi can address this issue.

To imitate the failure of the machine tool we call the *Pause* method of the server *Tool* from the *OnPlanned* event of scheduler *Failure*.

```
private void Failure_OnPlanned(TScheduler sender)
{
    Tool.Pause();
}
```

The *Pause* method stops the service of the transaction and fires the *OnInterruption* event for the server. In our case, we have decided to suspend the transaction which was in service.

```
private void Tool_OnInterruption(TServer sender, Transaction transaction)
{
    sender.Suspend();
}
```

In the *OnPlanned* event of scheduler *Recovery* we resume the server. On resuming, the suspended transaction will be put back in service.

```
private void Recovery_OnPlanned(TScheduler sender)
{
    Tool.Resume();
}
```

The same method can be used for a *TMultiServer* component.

Sample 15. High-priority interruption in TServer

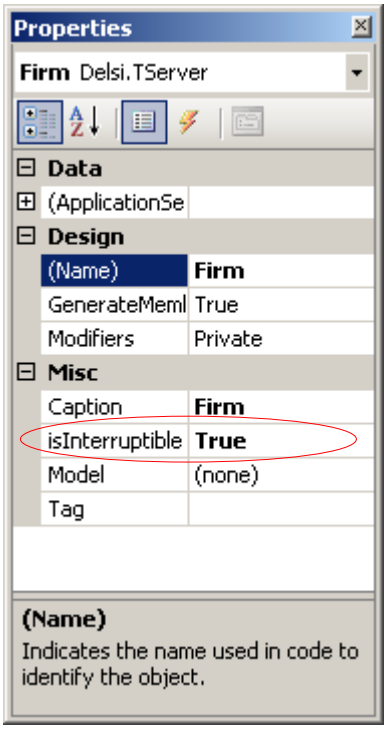
In this sample we are considering the firm, which executes some orders coming from customers. The orders can have one of the following priorities: high, medium or low. The firm can execute one order at a time. The firm has the policy that the order with higher priority will interrupt the execution of the order with lower priority. If the interrupted order has priority "low", it will be suspended from being executed. The interrupted order with priority "medium" will be transferred to a sub-contractor regardless of its completion degree.
If the order comes at the time when firm is busy and this new order cannot interrupt the current order execution, this arrived order will be transferred to subcontractors.

The time between order arrivals has exponential distribution with mean 4 business hours, the service time is uniformly distributed between 3 and 5 business hours. The subcontractors have enough capacity to execute the orders with the service time uniformly distributed also between 3 and 5 hours. The fractions of orders with different priorities are the following: 20% - low, 50% - medium, 30% - high. We need to simulate 1000 business hours of the firm work.

We would like to find out the following:

- how many orders of each priority were completed by the firm
- how many orders of each priority were completed by subcontractors
- how many started but unfinished orders were transferred to subcontractors
- utilization of the firm
- the total time of orders processing for each priority

First of all, we define our *TServer* component *Firm* as interruptible.



Then we determine the priority of the transactions in the *AfterGeneration* event. Here we also record the birth time of the transaction as well as define the transaction as interruptive.

```
private void Arrival_AfterGeneration(TGenerator sender, Transaction transaction)
{
    double p = tMultiRand1.Uni01();
    MyClass myclass = new MyClass();
    myclass.birthTime = tModel1.SysTime;

    if (p < 0.2)
    {
        transaction.Priority = 0;    // 20% - priority 0
    }
    else if (p < 0.7)
    {
        transaction.Priority = 1;    // 50% - priority 1
    }
    else
    {
        transaction.Priority = 2;    // 30% - priority 2
    }
    transaction.isInterruptive = true;
    transaction.Tag = myclass;
}
```

In the *OnRouting* event of generator *Arrival* we analyze if the transaction can be taken to be served by server *Firm*. If not, the transaction will be transferred to the *TDelay* component *Subcontractors*, which simulates the subcontractors. If the server can accept the transaction for service, it means that either the server is empty or it contains a transaction with a lower priority which can be displaced by the incoming one.

```
private void Arrival_OnRouting(TGenerator sender, Transaction transaction)
{
    if (Firm.isReadyToGet(transaction))
    {
        sender.Send(Firm);
    }
    else
    {
        sender.Send(Subcontractors);
    }
}
```

When the service of the transaction with a lower priority is interrupted, we need to decide what to do with it. We do it on handling the *OnInterruption* event.

```
private void Firm_OnInterruption(TServer sender, Transaction transaction)
{
    if (transaction.Priority == 0)
    {
        sender.Suspend();
    }
    else
    {
        sender.Send(Subcontractors);
    }
}
```

After processing transactions in server *Firm* and in delay block *Subcontractors* we route transactions to terminators *CompletionFirm* and *CompletionSub* correspondingly.

```
private void Firm_OnRouting(TServer sender, Transaction transaction)
{
    sender.Send(CompletionFirm);
}

private void Subcontractors_OnRouting(TDelay sender, Transaction transaction)
{
    sender.Send(CompletionSub);
}
```

In the terminators we collect the statistics about how many orders of each priority were completed by the firm and by the subcontractors. Using *TTabulator* components, we also collect the statistics about the total processing time for the orders of each priority.

```
private void CompletionFirm_OnEnter(TTerminator sender, Transaction transaction)
{
    double processingTime = tModel1.SysTime -
                            ((MyClass)transaction.Tag).birthTime;

    switch (transaction.Priority)
    {
        case 0:
            Completed_0++;
            tTabulator0.Tabulate(processingTime);
            break;
        case 1:
            Completed_1++;
            tTabulator1.Tabulate(processingTime);
            break;
        case 2:
            Completed_2++;
            tTabulator2.Tabulate(processingTime);
            break;
    }
}

private void CompletionSub_OnEnter(TTerminator sender, Transaction transaction)
{
    double processingTime = tModel1.SysTime -
                            ((MyClass)transaction.Tag).birthTime;

    switch (transaction.Priority)
    {
        case 0:
            OutSourced_0++;
            tTabulator0.Tabulate(processingTime);
            break;
        case 1:
            OutSourced_1++;
            tTabulator1.Tabulate(processingTime);
            break;
        case 2:
            OutSourced_2++;
            tTabulator2.Tabulate(processingTime);
            break;
    }
```

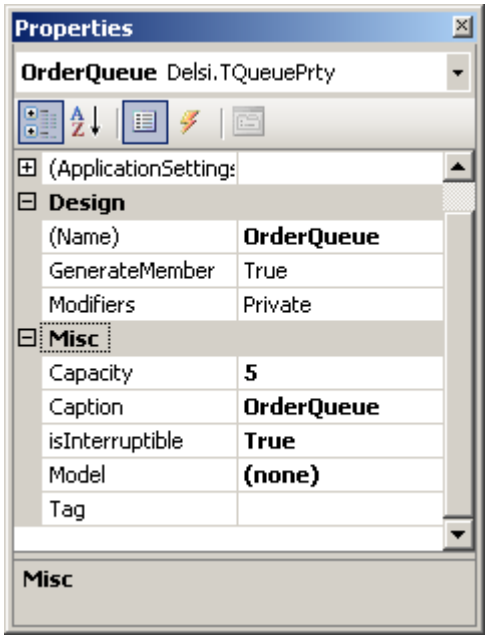
The output of the results looks like this.

```
Completed orders with low priority: 22
Completed orders with medium priority: 46
Completed orders with high priority: 60
Outsourced orders with low priority: 25
Outsourced orders with medium priority: 79
Outsourced orders with high priority: 19
Started and then outsourced orders: 15
Firm utilization: 0.54151469349915
Processing time for low-priority order:
   Average: 5.70096475879345
   Deviation: 3.65933140380536
Processing time for medium-priority order:
   Average: 4.166003695925
   Deviation: 0.919640158363294
Processing time for high-priority order:
   Average: 4.0391750187215
   Deviation: 0.548959997005325
```

Sample 16. High-priority interruption in TQueuePrty

If we take a look at the results in the previous sample we will find out that the manager of the firm is too much obsessed with making the delivery time short. However the firm doesn't utilize its resources well. It is occupied with order execution only 54% of time. That's why the manager of the firm hired a business consultant who recommended to line-up the incoming orders into a queue with limited capacity. In this queue the orders will be sorted according to their priorities. The orders of the same priority will be ordered in FIFO order. The queue supports the interruption by priority, so when the queue reaches its capacity the incoming order may displace the order with a lower priority from the queue. In that case the displaced order will be transferred to a subcontractor. If the order cannot be placed in the queue (because the queue is full and the priority of the order is too low to preempt any other in the queue), the order will also be transferred to a subcontractor. The capacity of the queue is 5. All other parameters of the model are the same as in previous example.

To simulate the proposed queue of orders we will use *TQueuePrty* component *OrderQueue.* We define this queue as interruptible and set its capacity to 5.



In the *OnRouting* event of generator *Arrival* we analyze if the transaction can be taken into queue *OrderQueue*. If not, the transaction will be transferred to *TDelay* component *Subcontractors*. If the interruptible queue with priorities can accept the transaction, it means that either the queue did not reach its capacity or it contains at least one transaction with lower priority which can be displaced by the incoming one.

```
private void Arrival_OnRouting(TGenerator sender, Transaction transaction)
{
    if (OrderQueue.isReadyToGet(transaction))
    {
        sender.Send(OrderQueue);
    }
    else
    {
        sender.Send(Subcontractors);
    }
}
```

If the transaction is displaced from the queue, it will be transferred to *TDelay* component *Subcontractors*.

```
private void OrderQueue_OnInterruption(TQueuePrty sender, Transaction transaction)
{
    sender.Send(Subcontractors);
}
```

On normal circumstances (if the transaction is ready to exit from the queue) we will route it to the server.

```
private void OrderQueue_OnRouting(TQueuePrty sender, Transaction transaction)
{
    sender.Send(Firm);
}
```

Everything else stays the same as in previous sample. Let's take a look at the results.

```
Completed orders with low priority: 44
Completed orders with medium priority: 92
Completed orders with high priority: 72
Outsourced orders with low priority: 15
Outsourced orders with medium priority: 46
Outsourced orders with high priority: 0
Started and then outsourced orders: 38
Firm utilization: 0.936267624449309
Processing time for low-priority order:
   Average: 26.2219066469008
   Deviation: 24.5736366732403
Processing time for medium-priority order:
   Average: 12.1874636571233
   Deviation: 7.87203464227049
Processing time for high-priority order:
   Average: 4.72603074831262
   Deviation: 1.57911442433329
```

Here we see that the strategy suggested by the business consultant works. Although the average values of the order processing time have increased (greater average time for less priority), the utilization of the firm resources has dramatically increased up to 93%.

Sample 17. High-priority interruption in TMultiServer

In this sample, the firm discussed previously is able to execute 4 orders simultaneously. The inter-arrival time in this sample has exponential distribution with the mean equal to 1 business hour. The rest of the problem description stays the same as in the previous sample. To model the simultaneous execution of orders we will use *TMultiServer* component *Firm* with the capacity equal to 4. We also define the multi-server as interruptible.

The events for multi-server *Firm* are the same as for server *Firm* in the previous example.

```
private void Firm_OnEnter(TMultiServer sender, Transaction transaction)
{
    sender.SetTime(tMultiRand1.Uniform(3, 5));
}

private void Firm_OnInterruption(TMultiServer sender, Transaction transaction)
{
    if (transaction.Priority == 0)
    {
        sender.Suspend();
    }
    else
    {
        sender.Send(Subcontractors);
    }
}

private void Firm_OnRouting(TMultiServer sender, Transaction transaction)
{
    sender.Send(CompletionFirm);
}
```

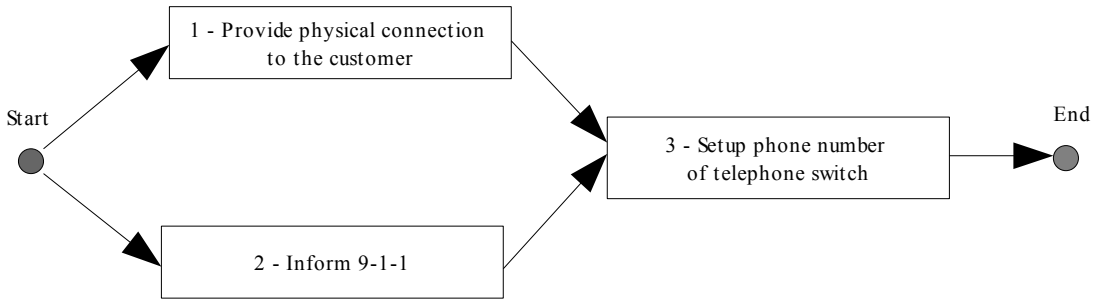
The rest of the code stays the same. Just for curiosity, let's take a look at the results.

```
Completed orders with low priority: 165
Completed orders with medium priority: 406
Completed orders with high priority: 301
Outsourced orders with low priority: 28
Outsourced orders with medium priority: 126
Outsourced orders with high priority: 0
Started and then outsourced orders: 121
Firm utilization: 0.989708034414055
Processing time for low-priority order:
   Average: 11.8126607927471
   Deviation: 7.98061335299436
Processing time for medium-priority order:
   Average: 4.99776835559728
   Deviation: 1.5981405084236
Processing time for high-priority order:
   Average: 4.06484069558765
   Deviation: 0.636886614743831
```

In this example we have increased both servicing capacity and the arrival rate in 4 times. The efficiency of our business has been improved.

Sample 18. Simulating workflow. TSplitter and TJoiner

Workflows play an important role in contemporary business applications. Typically, a workflow represents the sequence of tasks. In order to complete a task, some resources should be involved, at least the agent who executes the task. From this prospective, we can only imagine how wide the possible area of applying the methodology of queuing systems for workflow applications. In this sample we will take a look at the domain area of telecommunications and simulate simplified version of the workflow for phone line activation order. In our example the workflow has the following structure.



When a customer places the order the computer system will dispatch tasks 1 and 2 into the queues of corresponding departments. Task 3 with placed in the queue of the third department only after both tasks 1 and 2 are completed.

The orders for phone line activations arrive through the company's call center with the exponentially distributed intervals having mean 15 min. The time of task execution is normally distributed with the following parameters.

#	Task	Mean. min	Deviation, min.
1	Provide physical connection to the customer	30	5
2	Inform 9-1-1	10	2
3	Setup phone number of telephone switch	15	3

In our model each department will be represented by a combination of a queue and a server. The challenge here is the fact that in Delsi the same transaction cannot occupy more than one block. To overcome the limitation we will split each generated transaction into two in a *TSplitter* component named *Splitter.* Let's call those two transactions *copies.* The copies have the same *Id* but different values of *CopyId*.

```
private void Arrival_OnRouting(TGenerator sender, Transaction transaction)
{
    sender.Send(Splitter);
}
```

Obviously, component *Splitter* will have its property *SplitFactor* equal to 2. We will forward those two copies into the queues of first and second departments (*Queue1* and Queue2).

```
private void Splitter_OnRouting(TSplitter sender, Transaction transaction)
{
    sender.Send(Queue1);
    sender.Send(Queue2);
}
```

Here you can see something interesting: we call the *Send* method two times on handling one event. Yes, we may do this, because we know for sure that there are two transactions ready to go in the splitter when the *OnRouting* event is fired.

After completion of the service in servers *Department1* and *Department2*, two previously divided transactions will join back into one transaction in *TJoiner* component *Joiner*. This component has its *JoinFactor* equal to 2 as well.

```
private void Department1_OnRouting(TServer sender, Transaction transaction)
{
    sender.Send(Joiner);
}

private void Department2_OnRouting(TServer sender, Transaction transaction)
{
    sender.Send(Joiner);
}
```

The beauty of the combination of *TSplitter* and *TJoiner* is that the first copy arrived to the *TJoiner* component will wait for the arrival of copy with the same *Id* and ignore all other entering transactions. That is exactly what we need: we are joining two branches of the same workflow instance belonging to the same order.

After the joining, the resulted transaction will be sent to the queue of the third department (*Queue3)*.

```
private void Joiner_OnRouting(TJoiner sender, Transaction transaction)
{
    sender.Send(Queue3);
}
```

Among the other standard results we can see the statistics collected in *Splitter* and *Joiner*.

```
Splitter (TSplitter)
-----------------------------------
Entries: 43
Exits: 86
Count: 0

.....

Joiner (TJoiner)
-----------------------------------
Entries: 57
Exits: 22
Count: 13
```

In the results for *Joiner,* we can see that 40 incoming transactions were joined into 20 outgoing, and 12 transactions are still waiting for their "buddies" to join them.

Sample 19. Selecting from multiple queues. TPicker

In a real life environment, the provisioning department (a team of agents) may execute tasks from several queues. In this sample we have a team of 4 agents that complete the tasks from queues A, B and C. The tasks arrive to the department with the time intervals having the exponential distribution with mean 3 min. By the nature of the tasks, 35% of them come to queue A, 35 % - to queue B and the rest to queue C. The task execution time is normally distributed with mean value of 15 min. and standard deviation 3 min. for all task types. We will simulate the work of the department for 8 hours.

The agents can pick the task for different queues using the following rules:

- Randomly
- Using "Round Robin" discipline
- By prioritizing the queues (taking the tasks from the most important queue (A) first and from less important queue (C) last)

We want to conduct the simulation runs for all three rules.

After generating a transaction we send it to one of the three queues.

```
private void Arrival_OnRouting(TGenerator sender, Transaction transaction)
{
    double p = tMultiRand1.Uni01();

    if (p < 0.35)
    {
        sender.Send(QueueA);
    }
    else if (p < 0.70)
    {
        sender.Send(QueueB);
    }
    else
    {
        sender.Send(QueueC);
    }
}
```

In order to provide picking the transaction from one of the queues according to some rule, we use *TPicker* component *Picker*, where we try to send all the transaction from the queues.

```
private void QueueA_OnRouting(TQueue sender, Transaction transaction)
{
    sender.Send(Picker);
}

private void QueueC_OnRouting(TQueue sender, Transaction transaction)
{
    sender.Send(Picker);
}
```

```
private void QueueB_OnRouting(TQueue sender, Transaction transaction)
{
    sender.Send(Picker);
}
```

Our *TPicker* component *Picker* has to be informed from which block it should pick the transactions. It is our responsibility to inform the picker before the simulation runs. In this application we decided to do it in the form loading event.

```
private void Form1_Load(object sender, EventArgs e)
{
    ..............

    // Populate picker with possible sources
    Picker.AddToPicking(QueueA);
    Picker.AddToPicking(QueueB);
    Picker.AddToPicking(QueueC);

    ..............
}
```

If we plan to use picking by the block priority, we should add the source blocks in the order of their priorities, the block with highest priority should be added first (in our case it is block *QueueA*).

After *Picker* receives a transaction from the chosen queue, it will send it to *TDelay* component *Team* simulating the team of 4 agents. Obviously, this component has capacity 4.

```
private void Picker_OnRouting(TPicker sender, Transaction transaction)
{
    sender.Send(Team);
}
```

In our application we allow users to choose the picking rule using *comboBox1*. Before we start the simulation, we set the picking rule chosen by a user.

```
switch (comboBox1.SelectedIndex)
{
    case 0:
        Picker.PickingRule = PickingRules.Random;
        textBox1.AppendText("RANDOM PICKING");
        break;
    case 1:
        Picker.PickingRule = PickingRules.RoundRobin;
        textBox1.AppendText("ROUND-ROBIN PICKING");
        break;
    case 2:
        Picker.PickingRule = PickingRules.Prioritized;
        textBox1.AppendText("PRIORITIZED PICKING");
        break;
}

tModel1.Simulate(480.0);
```

Experimenting with picking rules we can research how the picking rule changes the output parameters of the model. For instance, the *average count* and the *average time in the queue* for queue *C* have greater values in the case of prioritized picking than in other two cases (random and Round-Robin).

Sample 20. Changing transactions in numbers. TTerminator

In all previous examples we were using *TTerminator* only to end the life of transactions. In this sample we will demonstrate more of the *TTerminator* functionality. We also will discuss how to increase and decrease the amount of transactions during a simulation.

Imagine the turning workshop with four lathes. The workpieces arrive to the workshop in packages of ten items each. After unpacking they will be placed in a depot. Workers will take those workpieces and process them on the lathes. After processing the manufactured details (each made out of one workpiece) will be combined in packages of five and moved to the electroplating workshop. The time of this moving is negligibly small. There, all five details will be electroplated in a bulk and shipped together to a warehouse.

The packages of workpieces arrive to the turning workshop with the time intervals having uniform distribution between 28 and 32 min. The time of processing on a lathe is uniformly distributed between 8 and 16 min. The time of electroplating is uniformly distributed between 3 and 17. We need to collect statistical data for shipping intervals.

The supply of packages of workpieces will be simulated by generator *Supply*. The transaction generated in *Supply* represents a package of 10 workpieces. After that, we need to deal with the workpieces which will be placed on the depot for further processing. To make 10 transactions (workpieces) out of one package (10 workpieces) we will use the following trick: every time generator *Supply* generates a transaction we will emit 9 more transactions using *TEmitter* component *Unpacking*.

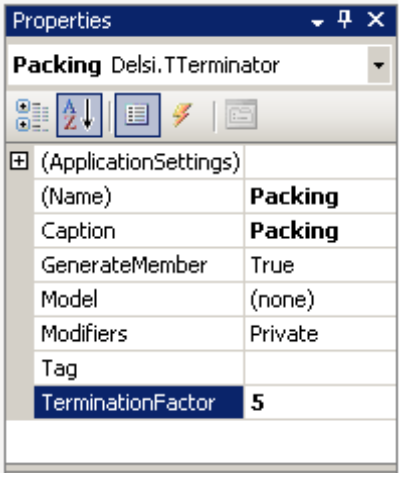
```
private void Supply_AfterGeneration(TGenerator sender, Transaction transaction)
{
    Unpacking.Emit(9);
}
```

Transactions from both generator Supply and emitter Unpacking will be sent to queue *WorkPieceDepot*.

```
private void Supply_OnRouting(TGenerator sender, Transaction transaction)
{
    sender.Send(WorkPieceDepot);
}

private void Unpacking_OnRouting(TEmitter sender, Transaction transaction)
{
    sender.Send(WorkPieceDepot);
}
```

Of course, this trick is not the only one possible to simulate the unpacking of packages. Instead we could use a combination of *TScheduler* and *TEmitter* or the combination of *TGenerator* and *TSplitter*.

After waiting in the depot and processing on the lathes (*TDelay* component *Tools*), the details will be packed in the packages of 5, which means that we need to make 1 transaction out of 5. To do so we will transfer the transactions to *TTerminator* component *Packing* with its property *TerminationFactor* equal to 5.



After the transactions (this time imitating package of 5 items) arrive to the depot in the electroplating workshop (*TQueue* component *PackDepot*) and after the electroplating (*TServer* component *Electroplating)* has been completed, the packages will be shipped to the warehouse (*TTerminator* component *Shipping*).

At this last step we will collect the statistics about the intervals between shipments (the *OnEnter* event of *Shipping*).

```
private void Shipping_OnEnter(TTerminator sender, Transaction transaction)
{
    if (lastShipmentTime == 0)
    {
        lastShipmentTime = tModel1.SysTime;
    }
    else
    {
        double interval = tModel1.SysTime - lastShipmentTime;
        tTabulator1.Tabulate(interval);
        lastShipmentTime = tModel1.SysTime;
    }
}
```

The result will look like the following.

```
Shipments: 29
Shipping intervals
   - Minimal: 13.164
   - Maximum: 18.705
   - Average: 15.231
   - Deviation: 1.343
```

Sample 21. Building models dynamically

In all previous examples we were adding Delsi components using the Toolbox at design time by “drag-n-drop”. In this sample we will see how to instantiate Delsi components at runtime. We will again use the barbershop model.

We have declared our components as private members of the form.

```
public partial class Form1 : Form
{
    private TModel MyModel;
    private TGenerator Arrival;
    private TQueue Hall;
    private TServer Barber;
    private TTerminator Exit;
    private TMultiRand MultiRand;
```

We have placed the initialization code for the components in the *InitializeMyModel* method which will be called from the form constructor.

```
    public Form1()
    {
        InitializeComponent();
        InitializeMyModel();
    }

    private void InitializeMyModel()
    {
        MyModel = new TModel();
        Arrival = new TGenerator();
        Hall = new TQueue();
        Barber = new TServer();
        Exit = new TTerminator();
        MultiRand = new TMultiRand();

        Arrival.Caption = "Arrvial to barbershop";
        Arrival.OnExit += new TGenerator.EventHandler(this.OnExitFromArrival);
        Arrival.OnRouting += new TGenerator.EventHandler(this.OnRoutingFromArrival);

        Hall.Caption = "Waiting Hall";
        Hall.OnRouting += new TQueue.EventHandler(this.OnRoutingFromHall);

        Barber.Caption = "Barber";
        Barber.OnEnter += new TServer.EventHandler(this.OnEnterToBarber);
        Barber.OnRouting += new TServer.EventHandler(this.OnRoutingFromBarber);

        Exit.Caption = "Exit from barbershop";

        MyModel.Add(Arrival);
        MyModel.Add(Hall);
        MyModel.Add(Barber);
        MyModel.Add(Exit);
    }
```

We had to manually write all of the methods used in the event handlers.

```
private void OnExitFromArrival(TGenerator sender, Transaction transaction)
{
    sender.SetTime(MultiRand.Uniform(12.0, 24.0));
}

private void OnRoutingFromArrival(TGenerator sender, Transaction transaction)
{
    sender.Send(Hall);
}

private void OnRoutingFromHall(TQueue sender, Transaction transaction)
{
    sender.Send(Barber);
}

private void OnEnterToBarber(TServer sender, Transaction transaction)
{
    sender.SetTime(MultiRand.Uniform(14.0, 20.0));
}

private void OnRoutingFromBarber(TServer sender, Transaction transaction)
{
    sender.Send(Exit);
}
```

So what's the value of this approach? The value is in the ability to build models dynamically. For instance, you may create your own XML-based simulation language and an environment where users will supply XML-files with the model description. Your application will parse the XML-files, build the models dynamically and then run them. You will be able to suggest such environments as SOA (Service Oriented Architecture) services for your own customers.

Sample 22. Using arrays of blocks in complex topologies

In many real-life simulation applications the amount of object types (modeled by blocks) can be counted by dozens and the amount of the object instances can reach hundreds. Imagine for a moment a telecommunication network with hundreds of routers and switches combined within a complex network topology. In this scenario, you probably will not be able to build a model by defining hundreds of blocks individually.

What to do? The answer is to unite the blocks having the same behavior into the arrays (or other data structures).

In this sample we will demonstrate how to use the arrays of blocks. Imagine the arrival terminal in an international airport. After landing and picking up luggage, the passengers arrive to a customs examination area. There they get into the shortest line to a customs officer. In our airport there are four customs officers. The passengers arrive to the customs examination area with inter-arrival interval having exponential distribution with mean 2 min. The time spent by an officer for examining one passenger is uniformly distributed between 6 and 10 min.

Like in previous sample, we are creating Delsi components at runtime. First, we declare our components and arrays of components as private members of the form. We model the lines to officers by the array of queues *Lines*, and the officers themselves by the array of servers *Officers*.

```
public partial class Form1 : Form
{
    private TModel MyModel;
    private TGenerator Arrival;
    private TQueue[] Lines;
    private TServer[] Officers;
    private TTerminator Exit;
    private TMultiRand MultiRand;
```

As in the previous sample, we place the initialization code for the components in the *InitializeMyModel* method which will be called from the form constructor. The code of *InitializeMyModel* looks familiar. Here we explicitly declare and instantiate the event handlers for the elements of the arrays.

```
    // Instantiate handlers to be re-used by block in arrays
    TQueue.EventHandler handler_OnRoutingFromLine =
           new TQueue.EventHandler(this.OnRoutingFromLine);
    TServer.EventHandler handler_OnEnterToOfficer =
           new TServer.EventHandler(this.OnEnterToOfficer);
    TServer.EventHandler handler_OnRoutingFromOfficer =
           new TServer.EventHandler(this.OnRoutingFromOfficer);
```

Obviously, to instantiate and initialize elements of the array we use a loop. Please note, we store the array index of a block in its *Tag* property.

```
for (int i = 0; i < 4; i++)
{
    Lines[i] = new TQueue();
    Lines[i].Tag = i;
    Lines[i].Caption = "Queue" + i.ToString();
    Lines[i].OnRouting += handler_OnRoutingFromLine;
    MyModel.Add(Lines[i]);

    Officers[i] = new TServer();
    Officers[i].Tag = i;
    Officers[i].Caption = "Officer" + i.ToString();
    Officers[i].OnEnter += handler_OnEnterToOfficer;
    Officers[i].OnRouting += handler_OnRoutingFromOfficer;
    MyModel.Add(Officers[i]);
}
```

When we route a transaction from generator *Arrival*, we need to choose the queue, which together with its corresponding server, contains the minimal number of transactions.

```
// Routing to minimal line
private void OnRoutingFromArrival(TGenerator sender, Transaction transaction)
{
    long min = long.MaxValue;
    int index = 0;

    for (int i = 0; i < 4; i++)
    {
        if (Lines[i].Count() + Officers[i].Count() < min)
        {
            min = Lines[i].Count() + Officers[i].Count();
            index = i;
        }
    }
    sender.Send(Lines[index]);
}
```

When a transaction should be routed from a queue, we read its array index from its *Tag* property and send the transaction to the server with the same index.

```
// Routing to the server with the same index
private void OnRoutingFromLine(TQueue sender, Transaction transaction)
{
    int index = (int)sender.Tag;
    sender.Send(Officers[index]);
}
```

The routing topology in this sample is very simple. In real-life systems, it can be very complex. One of the convenient ways to define a routing topology between the blocks of any two arrays in the model is to use *adjacency matrices.*

Sample 23. Batching and Unbatching

Imagine a factory producing some auto parts. Besides others, there are two workshops in the factory: "*Metalwork*" and "*Grinding*".

Workpieces come to the *Metalwork* workshop from a conveyor with inter-arrival time uniformly distributed between 0.9 - 1.1 min. The workpieces are being lathed on 5 machine tools. The lathing time is normally distributed with mean 5 min. and standard deviation 0.5 min.

After lathing, the box man will pack every 20 workpieces in a box and move it to *Grinding* workshop, where he will unpack them. The cumulative time of packing, moving and unpacking is normally distributed with mean 10 min. and standard deviation 3 min.

At the *Grinding* workshop, the workpieces will be processed on 7 grinding machines. The grinding time is normally distributed with mean 7 min. and standard deviation 1 min.

We need to find the average and the standard deviation of the time interval between the arrival of a workpiece to the *Metalwork* workshop and the end of grinding. Also, we would like to find out how busy is the box man.

To model this manufacturing process we will use the following blocks:

Arrival	*TGenerator*	Arrival of workpieces
StockLathing	*TQueue*	Stock of workpieces in the *Metalworks* workshop
Lathing	*TDelay*	Lathing
Packing	*TBatcher*	Packing
Moving	*TDelay*	Cumulative delay of packing, moving and unpacking
Unpacking	*TUnbatcher*	Unpacking
StockGrinding	*TQueue*	Stock of workpieces in the *Grinding* workshop
Grinding	*TDelay*	Grinding
EndOfProcessing	*TTerminator*	End of processing

Please, note that *TDelay* blocks *Lathing*, *Moving* and *Grinding* have the *Capacity* property set to 5, 1 and 7 respectively. The *BatchFactor* property in the *TBatcher* component *Packing* is set to 20.

The routing is very straight forward, transactions go from one block to another (as the blocks listed above). The manipulation with custom transaction fields and the use of *TTabulator* is the same as in *Sample 11*.

So, the point of this sample is to demonstrate the use of components *TBatcher* and *TUnbatcher*. There is nothing sophisticated here, they just work.

Here are the results of the simulation run.

```
Total Served: 400
Average time in the system: 61.145
Deviation time in the system: 13.221
Box man utilization: 0.364883885585624
```

Sample 24. Assembling with TAssembler

In this sample we are going to discuss an assembling workshop in a furniture factory producing tables. The following parts are required to assemble a table: 1 table-top, 4 legs, 8 screws and 8 nuts. Three workers are working independently in the workshop. The time interval required to assemble one table by one worker is normally distributed with the mean 30 min. and standard deviation 4 min. The table-tops arrive to the workshop every 10 min. The legs arrive in packs of 20 every hour. The stock of screws and nuts at the beginning of the shift is 430 and 370 respectively. The first table-top and the first 20 legs arrive at the beginning of the shift. We will simulate the process for 480 min. We need to find out the load of the workers and the residual stock at the end of the shift.

To build the model we use the following model items.

Parts	*TEmitter*	Arrival of parts
LegScheduler	*TScheduler*	Scheduling the arrival of legs
TopScheduler	*TScheduler*	Scheduling the arrival of tops
Stock	*TAssembler*	} Assembling
Workers	*TDelay*	}
Exit	*TTermintor*	End of assembling

The assembling scheme for the table can be presented like this.

Description	Part ID	Amount
table-top	1	1
leg	2	4
screws	3	8
nuts	4	8

We populate the assembling scheme right in the *Form1_Load* event.

```
private void Form1_Load(object sender, EventArgs e)
{

    ........
    // Define the assembling scheme for a table
    Stock.AddToScheme(1, 1);   // table-top
    Stock.AddToScheme(2, 4);   // legs
    Stock.AddToScheme(3, 8);   // screws
    Stock.AddToScheme(4, 8);   // nuts
}
```

To fill the stock with the initial amount of screws and nuts, we call emitter *Parts* before we start the simulation.

```
private void button1_Click(object sender, EventArgs e)
{
    // fill assembling stock with 700 screws
    partid = 3;
    Parts.Emit(430);

    // fill assembling stock with 600 nuts
    partid = 4;
    Parts.Emit(370);

    tModel1.Simulate(480.0);
    .....
}
```

To schedule the arrival of tops and legs we use schedulers *LegScheduler* and *TopScheduler*.

```
private void LegScheduler_OnPlanned(TScheduler sender)
{
    // arrival of 20 legs
    partid = 2;
    Parts.Emit(20);

    sender.SetTime(60.0);
}

private void TopScheduler_OnPlanned(TScheduler sender)
{
    // arrival of 1 table-top
    partid = 1;
    Parts.Emit(1);

    sender.SetTime(10.0);
}
```

We store part ID in the *Tag* property of emitted transactions.

```
private void Parts_AfterGeneration(TEmitter sender, Transaction transaction)
{
    transaction.Tag = partid;
}
```

When a transaction enters assembler *Stock* we designate the transaction with a part ID. In our particular case we have decided to store part ID in the *Tag* property of transactions. (However in general case you can designate the transaction with any other value of your choice).

```
private void Stock_OnEnter(TAssembler sender, Transaction transaction)
{
    // Tell the assembler what part Id is it.
    sender.Designate( (int)transaction.Tag );
}
```

The routing between blocks here is pretty straight forward.

To detect the left-overs in the stock after simulation we use the *Count(int PartId)* method of *TAssembler*.

```
tModel1.Simulate(480.0);
 .......
textBox1.AppendText("Screws left: " + Stock.Count(3).ToString());
textBox1.AppendText(System.Environment.NewLine);
textBox1.AppendText("Nuts left: " + Stock.Count(4).ToString());
textBox1.AppendText(System.Environment.NewLine);
```

The results of the simulation are the following.

```
Load of workers: 0.855085983789795
Screws left: 102
Nuts left: 42
```

Sample 25. Selecting from TStorage

Let' modify the previous sample of furniture manufacturing. This time the factory has the internal warehouse, where they store parts purchased from their suppliers (like screws and nuts). When the number of the screws or the nuts in the assembling stock decreases to 40, a pack of 50 screws and 50 nuts will be requested from the warehouse. The delivery interval is uniformly distributed between 4 and 6 min. At the beginning of the shift there are 420 screws and 310 nuts in the warehouse; there are 60 screws and 70 nuts in the assembling stock.

We need to find out how many tables were produced, what was the load of the workers, how many screws and nuts are left in the warehouse and in the assembling stock.

To build the model we use the following model items.

Parts	*TEmitter*	Arrival of parts
Init	*TEmitter*	Initial filling of the warehouse with parts
Warehouse	*TStorage*	Warehouse
Delivery	*TServer*	Delivery from the warehouse to the assembling workshop
Unpacking	*TUnbutcher*	Unpacking from the delivered pack
LegScheduler	*TScheduler*	Scheduling the arrival of legs
TopScheduler	*TScheduler*	Scheduling the arrival of tops
Stock	*TAssembler*	Assembling
Workers	*TDelay*	
Exit	*TTermintor*	End of assembling

Before we start simulation we fill both warehouse and assembling with initial amounts of screws and nuts.

```
private void button1_Click(object sender, EventArgs e)
{
    // fill the warehouse with 420 screws
    partid = 3;
    Init.Emit(420);
    // fill the assembling stock with 60 screws
    Parts.Emit(60);

    // fill the warehouse with 310 nuts
    partid = 4;
    Init.Emit(310);
    // fill the assembling stock with 70 nuts
    Parts.Emit(70);

    tModel1.Simulate(480.0);
    ........
}
```

On exiting from *Stock* we check the amount of screws and nuts there. If either the number of screws or the number of nuts is less then 40, we select 50 screws and 50 nuts from *Warehouse*. When we post selected transaction with the *Post(bool Batched)* method, the parameter *Batched* equals *true*. By doing this we pack all selected transaction into one batch.

```
private void Stock_OnExit(TAssembler sender, Transaction transaction)
{
    // If we are about to run out of screws, order a pack from the warehouse
    if (Stock.Count(3) < 40 || Stock.Count(4) < 40)
    {
        Warehouse.Select(isScrew, 50);
        Warehouse.Select(isNut, 50);
        Warehouse.Post(true);
    }
}
```

To specify the criteria for the selection we were using two methods: *isScrew* and *isNut.*

```
// Here are our customer methods implementing selection criteria
// Check if the transaction is a "screw"
private bool isScrew(Transaction transaction)
{
    if ((int)transaction.Tag == 3)
    {
        return true;
    }
    return false;
}

// Check if the transaction is a "nut"
private bool isNut(Transaction transaction)
{
    if ((int)transaction.Tag == 4)
    {
        return true;
    }
    return false;
}
```

After the simulation ended we determine the amounts of screws and nuts in the *Warehouse* also by calling method *Select(SelectionHandler Criteria, int Top),* this time with the *Top* parameter equal to 0.

```
private void button1_Click(object sender, EventArgs e)
 {
     .....
     tModel1.Simulate(480.0);
     .....
     int n = Warehouse.Select(isScrew, 0);
     textBox1.AppendText("Screws left in the warehouse: " + n.ToString());
     textBox1.AppendText(System.Environment.NewLine);

     n = Warehouse.Select(isNut, 0);
     textBox1.AppendText("Nuts left in the warehouse: " + n.ToString());
     textBox1.AppendText(System.Environment.NewLine);

     Warehouse.Unselect();
     .....
 }
```

These are the simulation results.

```
Tables produced: 40
Load of workers: 0.865678091961092
Screws left in the assembling stock: 66
Nuts left in the assembling stock: 36
Screws left in the warehouse: 20
Nuts left in the warehouse: 0
```

Sample 26. Selecting from TStorage using SQL queries

In *API Reference* we have discussed a solution of selecting transactions from *TStorage* with complex criteria.

What if a number of transactions in the storage is too big and the criteria are very sophisticated? Then using looping through all transactions in the storage will be quite insufficient. Wouldn't it be nice to select transactions from the storage with some sort of SQL query? We are far from the idea to implement SQL functionality in our simulation tool, however we can suggest developers quite simple work-around.

When a transaction enters the storage, you can add a record into a database of your choice. The record will contain the transaction *Id* (and if needed *CopyId)* plus all fields necessary for possible selections. When you need to select transactions using complex criteria, do it first in the database. Then you can iterate through the obtained result set and select transactions from the storage one by one using their *Id* or a combination of *Id* and *CopyId*. For this, you can use method *Select(long Id)* or method *Select(int Id, int CopyId).*

In order to demonstrate how it works we will use the scenario of the previous sample. To implement the solution with SQL we need to make the following steps.

1. Initialize database objects
2. When a transaction enters *TStorage* (in our case it is *Warehouse*), add a record to the database
3. When need to select from *TStorage:*
    - Select transaction *Id*-s from the database using the criteria required by our business logic
    - For each record in the result set:
        - call method *TStorage.Select(long Id)*
        - delete record from the database.
    - After all selections from *TStorage* are done, call the *Post* method.
4. To determine what is in the warehouse we will use SQL queries as well.

In this sample, we are using *dBase(IV)* file INVENTOR.DBF as a database*. We will interact with the database using OLEDB provider. The database files are located in /SAMPLES/Sample26/Database.

---

* Note: Please, be aware that with each simulation run, the files INVENTOR.DBF and PRIMARYK.NDX grow in size, despite the fact that before each run we delete all the records from the database table. That is the nature of dBase. In order to shrink the files in size you need to have a utility which can “ZAP” DBF-files. Another option is to restore files INVENTOR.DBF and PRIMARYK.NDX from the backup copies Copy of empty INVENTOR.DBF and Copy of empty PRIMARYK.NDX supplied with the sample.

** Note: Reviewing the code of this sample you can notice that we don't handle possible exceptions. Our goal is to demonstrate the main ideas. We leave all other necessary programming steps to you.

1. We initialize database objects together with the form.

```
public Form1()
 {
     InitializeComponent();

     string connString = "Provider=Microsoft.Jet.OLEDB.4.0;Data Source=";
     FileInfo file = new FileInfo(Application.ExecutablePath);
     DirectoryInfo dir = file.Directory;
     connString += dir.Parent.Parent.FullName;
     connString += "\\Database;Extended Properties=dBASE IV;User
                   ID=Admin;Password=";

     conn = new OleDbConnection(connString);

     cmd_insert = new OleDbCommand("insert into Inventory (Id, ProductId)
                                    values (@id, @productid)", conn);
     cmd_insert.Parameters.Add(new OleDbParameter("@id", OleDbType.BigInt));
     cmd_insert.Parameters.Add(new OleDbParameter("@productid",
                                OleDbType.Integer));

     cmd_delete = new OleDbCommand("delete from Inventory where Id=@id", conn);
     cmd_delete.Parameters.Add(new OleDbParameter("@id", OleDbType.BigInt));

     cmd_delete_all = new OleDbCommand("delete from Inventory", conn);

     // this implements our potentially sophisticatd selection.
     cmd_select = new OleDbCommand("select top 50 Id from Inventory where
                                    ProductId = @productid order by Id desc", conn);
     cmd_select.Parameters.Add(new OleDbParameter("@productid",
                               OleDbType.Integer));

     // this will be used to detect how many parts left in the warehouse
     cmd_select_left = new OleDbCommand("select count(*) from Inventory where
                                    ProductId = @productid", conn);
     cmd_select_left.Parameters.Add(new OleDbParameter("@productid",
                                     OleDbType.Integer));

 }
```

2. When a transaction enters *TStorage,* we add a record to the database.

```
private void Warehouse_OnEnter(TStorage sender, Transaction transaction)
{
    // insert a record into DB when a transaction enters the storage
    cmd_insert.Parameters[0].Value = transaction.Id;
    cmd_insert.Parameters[1].Value = (int)transaction.Tag;  // Part Id
    cmd_insert.ExecuteNonQuery();
}
```

3. When we need to select transactions from *TStorage* we do it like this.

```
private void Stock_OnExit(TAssembler sender, Transaction transaction)
{
    // If we are about to run out of screws, order a pack from the warehouse
    if (Stock.Count(3) < 40 || Stock.Count(4) < 40)
    {
        Select50Parts(3);  // select 50 screws from TStorage
        Select50Parts(4);  // select 50 nuts from TStorage
        Warehouse.Post(true);
    }
}


// select transactions from TStorage using SQL query
private void Select50Parts(int PartId)
{
    long trans_id;

    cmd_select.Parameters[0].Value = PartId;
    reader = cmd_select.ExecuteReader();

    object[] values = new object[reader.FieldCount];

    while (reader.Read())
    {
        reader.GetValues(values);
        // this (long)(doulbe) trick is a work around dBase(IV)
        trans_id = (long)(double)values[0];
        Warehouse.Select(trans_id);
        cmd_delete.Parameters[0].Value = trans_id;
        cmd_delete.ExecuteNonQuery();
    }
    reader.Close();
}
```

4. To determine what parts are left in the warehouse we use SQL queries.

```
private void button1_Click(object sender, EventArgs e)
{
    .....

    // Open the connection to the database
    conn.Open();
    // Clean-up database table before simulation
    cmd_delete_all.ExecuteNonQuery();

    tModel1.Simulate(480.0);
    .....

    // Determining left-overs in the warehouse
    textBox1.AppendText("Screws left in the warehouse: " +
                        getPartsLeft(3).ToString());
    textBox1.AppendText(System.Environment.NewLine);
    textBox1.AppendText("Nuts left in the warehouse: " +
                        getPartsLeft(4).ToString());
    textBox1.AppendText(System.Environment.NewLine);

    // close the connection to the database
    conn.Close();
    tModel1.Reset();
}


// determine how many parts left using SQL query
private int getPartsLeft(int PartId)
{
    object[] values = new object[1];
    int n = 0;

    cmd_select_left.Parameters[0].Value = PartId;
    reader = cmd_select_left.ExecuteReader();
    while (reader.Read())
    {
        reader.GetValues(values);
        n = (int)values[0];
    }
    reader.Close();
    return n;
}
```

The result are the same as in previous sample.

```
Tables produced: 40
Load of workers: 0.865678091961092
Screws left in the assembling stock: 66
Nuts left in the assembling stock: 36
Screws left in the warehouse: 20
Nuts left in the warehouse: 0
```