Discrete-event Simulation System

DELSI 2.0

API Reference

Table of Content

INTRODUCTION 3
1. SIMULATION LOGIC....5
1.1. Simulation entities....5
1.2. Routing transactions....6
1.3. Passing transactions....6
1.4. Activations....7
1.5. List of Future Events....7
1.6. Simulation algorithms....8
1.6.1. Initialization....8
1.6.2. Main cycle....8
1.6.3. Activation....9
1.6.4. Passing transactions....9
1.6.5. Ordering next activation event....10
2. DELSI API....12
2.1. Interfaces....13
2.1.1. IModelItem....13
2.1.2. IBlock....14
2.2. Exceptions....15
2.2.1. DelsiException....15
2.2.2. DelsiDeadlockException....15
2.3. Transaction....16
2.4. TModel....17
2.5. TScheduler....19
2.6. TGenerator....20
2.7. TQueue....23
2.8. TStack....27
2.9. TQueuePrty....28
2.10. TServer....34
2.11. TMultiServer....40
2.12. TDelay....46
2.13. TEmitter....50
2.14. TTerminator....52
2.15. TSplitter....55
2.16. TJoiner....58
2.17. TGate....61
2.18. TPicker....64
2.19. TBatcher....67
2.20. TUnbatcher....70
2.21. TAssembler....72
2.22. TStorage....77
2.23. TMultiRand....82
2.24. TTabulator....85

INTRODUCTION

The main idea of Delsi modeling is that your queuing formalization may be presented as an oriented graph with nodes which correspond to some processing facilities (such as generators, queues, servers etc). The arcs of the graph correspond to the streams of processed items. Such items are called transactions.

Imagine a bank office. Customers arrive to the office. Some of them want to be served by a teller, some to see financial advisor, some to use ATM. In order to get certain service, customers may wait in the corresponding line. After customers are served, they exit from the bank office. In this example we can model the bank office with a graph (Figure 1), which has nodes corresponding to processing facilities:

- generator “Arrival”
- queues “Line to tellers”, “Line to Advisor”, “Line to ATM”
- servers “Teller1”, “Teller2”, “Advisor”, “ATM”
- terminator “Exit”.

- In this example the processed items (transactions) will be customers. The transactions originate from the generator, then go from facility to facility and finally get disposed in the terminator.

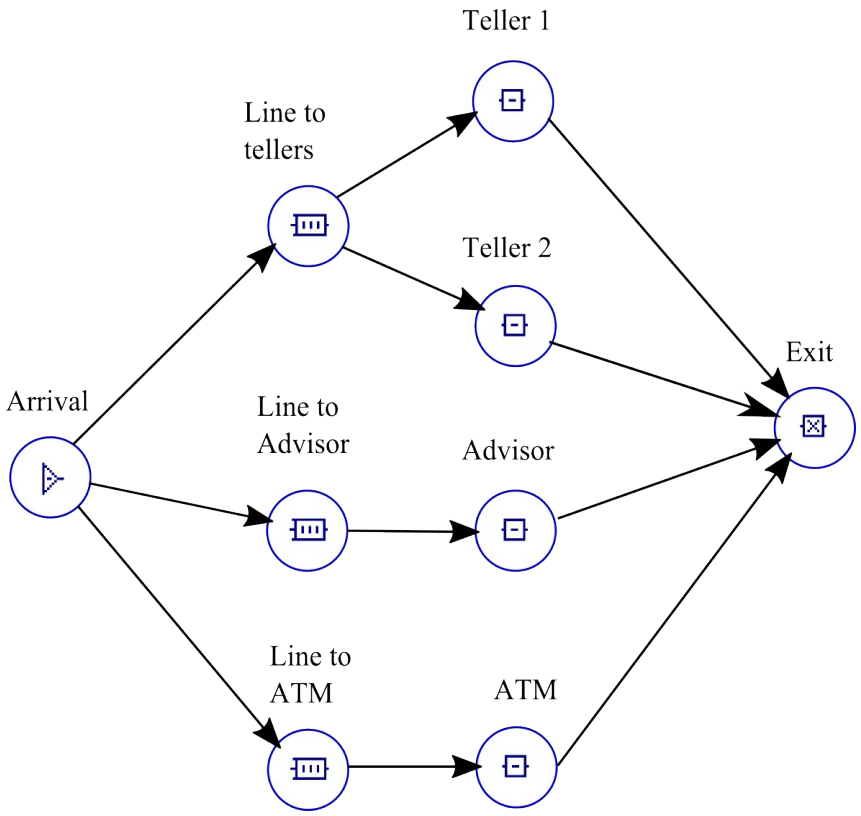

Figure 1.

With Delsi it is possible to control those processing objects and the transaction flow using custom algorithms.

In terms of implementation, Delsi 2.0 is a library (assembly) of simulation components and objects for using in Microsoft .NET Framework 4.0 environment.

During a simulation, transactions are being passed from one block to another. The custom algorithms have to be implemented as event handlers for certain events. The events can be: entering a block, routing, exit from a block, interruption of service and others. By using methods, properties and events of the components, it is possible to control the behavior of a model and obtain necessary statistical results.

Delsi allows a user to utilize all the power of Microsoft .NET Framework 4.0 and its development environment, such as Microsoft Visual Studio 2010, for developing a wide variety of simulation models.

With millions of .NET developers on one hand, and Delsi system as a .NET assembly on the other hand, discrete-event simulation may be quickly adopted by wide variety of customers with minimal training efforts.

.NET is a powerful framework, which enables developing simulation solutions with rich technological features and for various architectures. With Delsi 2.0 and .NET 4.0 Framework you can:

- Read and write data using different data sources (e.g. databases, spread-sheets, XML, flat files).
- Integrate third-party components like graphs, charts, or random generators.
- Build different interfaces to your simulator: Window desktop, Web-application, Web-Services (SOA), console, API.

Delsi 2.0, in a combination with a standard PC and Microsoft Visual Studio 2010, creates an affordable workplace for developing powerful and rich simulation applications.

1. SIMULATION LOGIC

1.1. Simulation entities

In this section we are discussing main simulation entities of the Delsi model not as particular programming classes or interfaces, but as abstract objects.

Transaction is a *dynamic entity,* which goes through the static structure of objects (blocks) of a queuing system. For the example of a bank office given in Introduction, the transactions will correspond to customers.

Model item is a *static entit*y that functions over time. At certain moments of time its internal state can be changed. The dynamics of a model item is event-driven. Examples of the events are:

- a transaction is generated in a model item
- a transaction enters a model item
- a transaction exits from a model item
- some scheduled generic event
- Etc.

Model items can be classified by two types: *block* and *scheduler.*

Model represents a collection of *model items*. A model is also responsible for moving the simulation clock. For the example shown in *Figure 1*, the model will correspond to the complete bank office.

Block is a *model item* which is able to get and/or send *transactions*. For the example of a bank office, *blocks* would model the following facilities: the entrance to the office (arrival of customers), the waiting lines, tellers, advisors, ATM-s and the exit from the office (departure of customers). The blocks are classified by the following types:

- Generator
- Emitter
- Queue
- Queue with priorities
- Stack
- Delay
- Server
- Multi-server
- Terminator
- Splitter
- Joiner
- Gate
- Picker
- Batcher
- Unbatcher
- Assembler
- Storage

Note: Each entity type is explained in more details in the sections below.

Scheduler is a *model item* which initiates some events of generic nature through intervals of time. In our example, the *scheduler* can represent the "Open Hours" for a bank office.

1.2. Routing transactions

Delsi doesn't have any objects corresponding to the transaction flow connections between the blocks. All the transaction routing is managed dynamically using programming code.

This gives programmers the ability to apply any sophisticated business rules to manage the routing. Delsi exposes special routing events, providing programmers with the opportunity to embed routing-related code into a model.

During routing, the transaction can be sent to one block at a time. At any moment of time a transaction may only reside in one block.

1.3. Passing transactions

Any block can be characterized by two attributes: *"Ready to Send Transaction*" and *"Ready to Get Transaction"*.

For example, if the teller is available she is *ready to get* an incoming customer. On the other hand, if the waiting line is empty it is *not ready to send* a customer to the teller.

Therefore, in Delsi a transaction can be passed from one block to another only if the block-sender is *ready to send* a transaction and the block-receiver is *ready to get* the transaction.

Note: In some cases (like a high-priority interruption) the value of the attribute "*Ready to Get Transaction"* depends on the transaction. Thus, this attribute is parameterized.

1.4. Activations

In order to make simulation possible, some of the events happening in the model should be planned on time line in advance. For instance, we need to plan when the next customer arrives, or as a customer gets to the teller, when the service will be completed. So, those special planned events are the customer's arrival and the completion of the service.

Activation is an event associated with a *model item* and initiated by the change of model time. The moment when the activation happens is called ***activation time.***

The *activation* is ***transaction-specific*** when it is relevant only for a specific transaction. For instance, ending of service by a teller is relevant only for the customer who is currently at the teller's front-desk. So the event of ending the service will be *transaction-specific*.

When we are planning for the next arrival of a customer to the bank office, it will be a customer we don't yet know about. So, the corresponding activation event will be *not transaction-specific*.

Table 1. Relationship between model items and activation events

Model Item	Activation Event	Transaction-specific
Generator	Generation of the next transaction	No
Queue	Ending of maximum waiting time in the queue (or stack) for the transaction	Yes
Queue with priorities		Yes
Stack		Yes
Delay	Ending of delay period for the transaction	Yes
Server	Ending of service for the transaction	Yes
Multi-server		Yes
Scheduler	Generic custom event	No

1.5. List of Future Events

When an *activation event* (like ending of the service) is being planned, the record of this event is added to the ***List of Future Event (LFE)*** which represents *system clock*.

The records have the following fields:

- activation time
- model item
- transaction

The records in the LFE are ordered by the *activation time*. When two or more events have the same activation time they are ordered by the FIFO rule (“First In – First Out”). The sequence of the *activation times* represents the flow of the ***system time***.

1.6. Simulation algorithms

The simulation logic is controlled by ***Delsi Runtime Engine.*** Its core simulation algorithms are shown below in simplified form in a pseudo-programming language.

1.6.1. Initialization

After the simulation started, the first step to be done is to initialize the model items.

```
foreach( model_item in model )
{
    model_item.Initialize();
}
```

The *Initialize* method inserts future activation events (into LFE) for generators and schedulers with the a*ctivation time* defined in their property *FirstTime*. For *generator*, the *FirstTime* property defines when the first transaction will be generated. For *scheduler*, it defines when the first scheduled event will be fired.

1.6.2. Main cycle

The main cycle of the simulation algorithm takes the next record from *LFE*, changes the *system time* and processes the *activation event*.

```
do
{
    Take the next record from LFE;
    SystemTime = activation_time;
    if (model_item is scheduler)
    {
         model_item.Activate();
    }
    else  // the model_item is block
    {
        model_item.Activate(transaction);
        // Add model item into List of Current Events (LCE)
        LCE.Add( model_item );
        Using LCE do all possible transaction passes;
    }
}
while (SystemTime<LimitTime);
```

1.6.3. Activation

The *model_item.Activate* method does different jobs for different model items:

- For *queue, queue with priorities* and *stack,* it selects the transaction whose maximum waiting time is expired and fires an *AfterTimeEnded* event.
- For *generator*, it generates new transaction, fires an *AfterGeneration* event and makes the generator *"Ready to Send"*.
- For the *delay* block, it selects the transaction whose delay has ended, releases it for exiting and makes the block *"Ready to Send"*.
- For *server* and *multi-server*, it selects the transaction whose service has ended, releases it for exiting and makes the block *"Ready to Send"*.
- For *scheduler*, it fires an *OnPlanned* event.

1.6.4. Passing transactions

In order to conduct all possible transaction passes at the certain moment of system time, Delsi maintains special data structure called ***List of Current Events (LCE).*** *LCE* includes all the blocks, which are in the state "*Ready to Send"*. Conducting all possible passes is managed be the following algorithm.

```
do
{
    foreach(block in List of Current Events)
    {
        block.OnRouting; // Route transaction to another block
        if ( !block.ReadyToSend )
        {
            Remove block from LCE;
        }
    }
}
while(at least one transaction pass has been done);
```

You can route the transaction at runtime in the routine of the *OnRouting* event of the block. Here you can apply any sophisticated logic for the transaction routing:

```
if ( your condition )
{
    block.Send( another_block );
}
else
{
    block.Send( yet_another_block );
}
```

The *Send* method typically works like this:

```
if ( another_block.ReadyToGet( transaction_to_be_passed ) )
{
    // The transaction exits from this block
    this_block.RemoveTransation;
    this_block.OnExit;  // User-defined handling of exiting event
    // The transaction enters to another block
    another_block.AddTransaction;
    another_block.OnEnter;  // User-defined handling of exiting event
}
```

1.6.5. Ordering next activation event

While handling the *OnEnter* event for a *queue*, *queue with priorities* or *stack,* you may specify the maximum waiting time for an incoming transaction by using the *SetTime* method.

You have to handle the *OnEnter* event for a *server, multi-server* or *delay* and specify the service time using the *SetTime* method; otherwise the service or delay of the incoming transaction will never end.

```
// the body of the event handling will be written by you
OnEnter
{
    .......
    elapsed_time = ...;  // Usually randomly generated number;
    SetTime( elapsed_time );
    .......

}
```

You have to handle the *OnExit* event for a *generator* and specify the time interval in which the next transaction will be generated; otherwise the next transaction will never be generated).

```
// the body of the event handling will be written by you
OnExit
{
    .......
    elapsed_time = ...;  // Usually randomly generated number;
    SetTime( elapsed_time );
    .......

}
```

While handling the *OnPlanned* event for a *scheduler,* you should specify the time interval in which the next custom event will occur; otherwise it will never occur again.

```
// the body of the event handling will be written by you
OnPlanned
{
    .......
    elapsed_time = ...;
    SetTime( elapsed_time );
    .......

}
```

When you are calling the *SetTime* method, you are adding a new record to the *List of Future Events*. Now we can see that in the beginning of the main simulation cycle we remove a record from LFE, and by using the *SetTime* method we add new records to LFE. Thus, the simulation cycle is now closed. The engine perpetually works until it is stopped by a time limit or explicitly by calling a *Stop* method.

2. DELSI API

The classes of Delsi framework belong to the *Delsi* namespace. In this document we will describe only the class members which are related to simulation. All of the source code fragments are presented in C# programming language.

The API includes interfaces, classes, and components. The components are the following.

		TModel
		TScheduler
		TGenerator
		TQueue
		TStack
		TQueuePrty
		TServer
		TMultiServer
		TDelay
		TEmitter
		TTerminator
		TSplitter
		TJoiner
		TGate
		TPicker
		TBatcher
		TUnbatcher
		TAssembler
		TStorage
		TMultiRand
		TTabulator

2.1. Interfaces

Delsi exposes two public interfaces used by the components: *IModelItem* and *IBlock*. These interfaces would not be used much in real-life simulation practices, so describing them has more of a referential value.

2.1.1. IModelItem

This interface represents a *model item* which corresponds to the *static entities* of Delsi model. Model items can be a *block* (see interface *IBlock*) or a *scheduler* (see component *TScheduler*).

Syntax

```
public interface IModelItem
```

Properties

```
string Caption { get; set;}
```

Gets or sets the text of the item caption.

```
TModel Model{ get; }
```

Gets the model (see *TModel*), where the item belongs to.

```
Object Tag { get; set; }
```

Gets or sets the object associated with the component. Any type derived from *Object* can be assigned to this property. If the *Tag* property is set through the Windows Forms designer, only text can be assigned.

2.1.2. IBlock

The **IBlock** interface represents a block, the model item which is able to get and/or send a transaction.

Syntax

```
public interface IBlock : IModelItem
```

Methods

```
void ClearStatistics()
```

Clears simulation statistics accumulated in the block.

```
bool isReadyToGet( Transaction transaction )
```

Returns *true* if the block is ready to get the specified transaction; otherwise returns *false*.

```
bool isReadyToSend()
```

Returns *true* if the block has a transaction ready to exit from the block; otherwise returns *false*.

```
string Report()
```

Returns the text containing the standard report with simulation statistical data for the block. The data returned by the *Report* method can be obtained with help of the variety of statistical methods of the blocks. So this method was implemented for your convenience.

```
bool Send( IBlock block )
```

Tries to send the transaction, which is ready to exit from the block, to another block

Properties

```
string Caption { get; set;}
```

Gets or sets the text of the item caption.

```
TModel Model{ get; }
```

Gets the model (see *TModel*), where the item belongs to.

2.2. Exceptions

Delsi supports two types of exceptions: *DelsiException* and *DelsiDeadlockEcxeption*.

2.2.1. DelsiException

This exception is thrown when *Delsi* framework encounters an error.

Syntax

```
public class DelsiException: ApplicationException;
```

Constructor

```
public DelsiException( string message )
```

Properties

```
public virtual string Message { get; }
```

2.2.2. DelsiDeadlockException

This exception is thrown when Delsi encounters a deadlock, i.e. the situation when the *List of Future Events* is empty and the simulation cannot run any further. Although the chances of a deadlock normally are very low, theoretically this situation can occur.

Syntax

```
public class DelsiDeadlockException: DelsiException;
```

Constructor

```
public DelsiDeadlockException( string message )
```

Properties

```
public virtual string Message { get; }
```

2.3. Transaction

Transaction is a class which represents dynamic entities going through the structure of objects (blocks).

Syntax

```
public class Transaction
```

Fields

```
public bool isInterruptive;
```

Determines if a transaction can interrupt the service of a transaction with lower priority.

```
public Object Tag;
```

Reference to a custom object associated with a transaction. This field actually provides you with the wide opportunities to associate a transaction instance with an object of any structure. After a transaction is terminated (in a *queue, multi-server, terminator, etc.*), this field will be automatically set to `null`.

Properties

```
public long Id { get; }
```

Transaction Id, which is assigned to a transaction during its generation in *TGenerator* and *TEmitter*.

```
public long CopyId { get; }
```

Transaction copy identifier. The original transactions generated by *TGenerator* or *TEmitter* always have *CopyId* = 1. The clones of the original transaction created in *TSplitter* have different values of *CopyId* which are unique within the scope of a simulation run.

```
public int Priority { get; set;}
```

Defines the transaction priority.

2.4. TModel

This component is responsible for functioning of the whole model. It stores the list of model items (blocks and schedulers). Using the methods of *TModel* you can start and stop simulation process or clear statistics collected in the blocks. Another useful feature of *TModel* is the ability to register transaction passes and system time changes.

Syntax

```
public class TModel
```

Methods

```
public void Add( IModelItem modelitem )
```

Adds a model item to the model.

```
public void ClearStatistics()
```

Clear simulation statistics accumulated in all blocks of the model by calling their method *ClearStatistics*. This method is very useful to determine the 'warm-up phase' of the simulated stochastic process.

```
public void Remove( IModelItem modelitem )
```

Removes a model item from the model. If the model does not contain the specified item, the model remains unchanged, no exception is thrown.

```
public string Report()
```

Returns the text with the standard report for all blocks in the model.

```
public void Reset()
```

Resets the model into its initial state, making it ready for the next run. Use this method when you need several simulation runs when changing input parameters.

```
public void Simulate( double endTime )
```

Starts simulation run. The simulation will continue until the system time reaches *endTime*.

```
public void Stop()
```

Explicitly stops simulation run. The run can be resumed by method *Simulate*.

Properties

```
public double SysTime { get;}
```

This property represents the system time which is not physical but modeled time. Before simulation start-up and after model reset, the system time is equal to 0.

Delegates

```
public delegate void AfterPassEventHandler( IBlock sender,
                                            IBlock receiver,
                                            Transaction transaction )
```

Represents the method that will handle the event fired each time a transaction is passed from one block to another.

```
public delegate void NewTimeEventHandler( IModelItem modelitem,
                                          Transaction transaction )
```

Represents the method that will handle the event fired when the system time changes its value.

Events

```
public event TModel.AfterPassEventHandler AfterPass
```

Occurs each time a transaction is passed from one block to another.

```
public event TModel.NewTimeEventHandler OnNewTime
```

Occurs when the system time changes its value (see the *SysTime* property).

2.5. TScheduler

TScheduler is a component which fires events over some specified periods of system time. By handling the events fired by *TScheduler*, you can do numerous useful things, such as:

- change the parameters of the model
- stop simulation run
- clear statistics
- lock/unlock gates
- create transaction by calling method *TEmitter.Emit*
- and much more!

Syntax

```
public class TScheduler : Component, IModelItem
```

Methods

```
public void SetTime( double time )
```

Sets the time interval in which the next event will be fired. Use this method while handling the *OnPlanned* event.

Properties

```
public double FirstTime { get; set;}
```

Determines when the first scheduled event will be fired. Delsi Runtime Engine will use this property to order the first activation for the scheduler by placing a record into *LFE*.

```
public string Caption { get; set;}
```

Gets or sets the text of the item caption.

```
public TModel Model{ get; }
```

Gets the model where the item belongs to.

```
public Object Tag { get; set; }
```

Gets or sets the object associated with the component. Any type derived from *Object* can be assigned to this property. If the *Tag* property is set through the Windows Forms designer, only text can be assigned.

Delegates

```
public delegate void EventHandler( TScheduler sender )
```

Represents the method which will handle the event fired by the scheduler *sender*.

Events

```
public event TScheduler.EventHandler OnPlanned
```

This is a custom event scheduled by you and fired by the scheduler.

2.6. TGenerator

TGenerator periodically generates transactions, one transaction at a time.

When a transaction leaves a generator, it's time to specify when the next transaction will be generated. You should specify the elapsed *"time to the next generation" by* using the *SetTime* method within the *OnExit* event. This mechanism simulates sequential arrivals of dynamic entities into the system. Usually, the inter-arrival interval can be described by a probability distribution. By assigning random numbers to the *time to the next generation* we can simulate statistical properties of the arrival.

Syntax

```
public class TGenerator : Component, IBlock, IModelItem
```

Methods

```
public double AverageTime()
```

Returns the average time between sequential generations.

```
public void ClearStatistics()
```

Clears simulation statistics accumulated in the block.

```
public long Count()
```

Returns the current number of transactions in the block. For *TGenerator*, the property returns 1 if the generator contains a transaction waiting to exit; 0 otherwise.

```
public double DeviationTime()
```

Returns the standard deviation of the time between sequential generations.

```
public long Exits()
```

Returns the total number of transactions exited from the block..

```
public bool isReadyToGet( Transaction transaction )
```

TGenerator cannot receive a transaction, so its method *isReadyToGet* always returns *false*.

```
public bool isReadyToSend()
```

Returns *true* if the block has a transaction to send.

```
public bool Send( IBlock block )
```

Tries to send the transaction which is ready to exit from the block, to another *block*. It returns *true* if the transaction was successfully sent; otherwise, *false*. When using the *Send* method, you have to meet the following conditions:

- For *TGenerator*, this method can be called only on handling event *OnRouting*.
- The block should have a transaction to send.
- The receiving block should belong to the same model as sending.
- The receiving block cannot be the same as sending.

If any of these conditions are not satisfied, Delsi Runtime Engine will throw *DelsiException*.

```
public virtual string Report()
```

Returns text containing the standard statistical report for the block. For *TGenerator* the standard report looks like this.

```
Exits: 27
Count: 0
Average time: 18.2895567615188
Deviation time: 3.54829534356622
```

```
public void SetTime( double time )
```

Sets the time to the next generation. Use this method while handling the *OnExit* event.

Properties

```
public double FirstTime{ get; set; }
```

Determines when the first transaction will be generated. The engine will use this property to order the first activation for the generator by placing a record into *LFE*.

```
public string Caption { get; set;}
```

Gets or sets the text of the item caption.

```
public TModel Model{ get; }
```

Gets the model where the item belongs to.

```
public Object Tag { get; set; }
```

Gets or sets the object associated with the component.

Delegates

```
public delegate void EventHandler( TGenerator sender,
                                   Transaction transaction )
```

Represents the method which will handle the event fired by generator *sender* for the specified *transaction*.

Events

```
public event TGenerator.EventHandler AfterGeneration
```

Occurs right after a transaction has been generated. This is a good place to setup the transaction properties.

```
public event TGenerator.EventHandler OnExit
```

Occurs when a transaction exits from the block. For *TGenerator,* it is the place where you should set the time to the next generation by using method *SetTime.*

```
public event TGenerator.EventHandler OnRouting
```

Fired when the block has a transaction ready to exit from the block. This is the place where the transaction will be sent to another block (see method *Send*). You can apply any sophisticated logic to determine where to send the transaction (if at all). If the transaction is not sent, it will remain in the block until the next *OnRouting* event will be fired by Delsi Runtime Engine.

2.7. TQueue

This component represents a queue that handles transactions in accordance with the FIFO rule “First In – First Out”.

When a transaction enters the queue, you may limit the waiting time (within the *OnEnter* event). Specify the maximum waiting time in the queue by using the *SetTime* method. The maximum waiting time may be described by a probability distribution. By assigning random numbers to the maximum waiting time we can simulate its statistical properties.

When the maximum waiting period for a transaction has ended, Delsi fires the *AfterTimeEnded* event . Within this event you can decide what to do with the transaction. One of the options is to send the transaction to another block. If the transaction was not sent to another block, or if you don’t handle this event at all, the transaction will be terminated.

A queue may have limited capacity. If the number of transactions in a queue is less than its capacity, the queue is in the state of *"Ready to Get"*. If the queue contains at least one transaction, it is in the state of "Ready to Send".

Syntax

```
public class TQueue : Component, IBlock, IModelItem
```

Methods

```
public double AverageCount()
```

Returns the average number of transactions in the block.

```
public double AverageTime()
```

Returns the average time spent by transactions in the block.

```
public double AverageTimeNonZero()
```

Returns the average non-zero time spent in the queue, i.e. the average is calculated only for positive values.

```
public void ClearStatistics()
```

Clears simulation statistics accumulated in the block.

```
public virtual long Count()
```

Returns the current number of transactions in the queue.

```
public double DeviationTime()
```

Returns the standard deviation of the time spent in the queue.

```
public double DeviationTimeNonZero()
```

Returns the standard deviation of non-zero time spent in the queue, i.e. the standard deviation that was calculated only for positive values.

```
public long Entries()
```

Returns the total number of transactions which have entered the block.

```
public long Exits()
```

Returns the total number of transactions which have exited from the block.

```
public virtual long ExitsByTimeLimit()
```

Returns the number of transactions which have left the queue because the maximum waiting period has expired.

```
public virtual bool isReadyToGet( Transaction transaction )
```

Returns *true* if the block is ready to receive a specific transaction. *TQueue* is always ready to get a transaction as long as it has not reached its capacity.

```
public bool isReadyToSend()
```

Returns *true* if the queue has a transaction to send. *TQueue* is always ready to send a transaction if it has one.

```
public long MaxCount()
```

Returns the maximum number of transactions in the block.

```
public double MaxTime()
```

Returns the maximum time spent in the block.

```
public double MinTime()
```

Returns the minimal time spent in the block.

```
public virtual string Report()
```

Returns text containing the standard statistical report for the block. For *TQueue* the standard report looks like this.

```
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
```

```
public virtual bool Send( IBlock block )
```

Tries to send the transaction which is ready to exit from the block, to another block. It returns *true* if the transaction was successfully sent; otherwise, *false*. When using the *Send* method, you have to meet the following conditions:

- For *TQueue*, this method can be called only on handling the events *OnRouting* and *AfterTimeEnded*.
- The block should have a transaction to send.
- The receiving block should belong to the same model as sending.
- The receiving block cannot be the same as sending.

If any of these conditions are not satisfied, Delsi Runtime Engine will throw *DelsiException*.

```
public virtual void SetTime( double time )
```

Sets maximum waiting time for the transaction entering the block. Use this method while handling the *OnEnter* event.

```
public double Usage()
```

Returns the relative part of simulation time when the block was containing at least one transaction.

```
public long ZeroEntries()
```

Returns the number of transaction entries into the block, when the block was empty.

Properties

```
public int Capacity{ get; set; }
```

Defines the maximum possible number of transactions in the block. When the number of transactions reaches its capacity, the block becomes *not "Ready to Get"*. If *Capacity* is 0, the number of transactions in the block is unlimited.

```
public string Caption { get; set;}
```

Gets or sets the text of the item caption.

```
public TModel Model{ get; }
```

Gets the model where the item belongs to.

```
public Object Tag { get; set; }
```

Gets or sets the object associated with the component.

Delegates

```
public delegate void EventHandler( TQueue sender,
                                   Transaction transaction )
```

Represents the method which will handle the event fired by queue *sender* for a specified *transaction*.

Events

```
public virtual event TQueue.EventHandler AfterTimeEnded
```

Occurs when the maximum waiting period for a transaction has expired. Here you can decide what to do with the transaction. One of the options is to send the transaction to another block. If the transaction is not sent to another block, or if you don't handle this event at all, the transaction will be terminated.

```
public virtual event TQueue.EventHandler OnEnter
```

Occurs when a transaction enters the block. This is the place where you may set the maximum waiting time for the transaction using method *SetTime*.

```
public virtual event TQueue.EventHandler OnExit
```

Occurs when a transaction exits from the block.

```
public virtual event TQueue.EventHandler OnRouting
```

Fired when the block has a transaction ready to exit from the block. This is the place where the transaction will be sent to another block (see method *Send*). You can apply any sophisticated logic to determine where to send the transaction. If the transaction is not sent, it will remain in the block until the next *OnRouting* event will be fired by Delsi Runtime Engine.

2.8. TStack

This component represents a stack which handles transactions in accordance with the LIFO rule "Last In – First Out". This is the only difference between *TStack* and *TQueue*.

Syntax

```
public class TStack : TQueue, IBlock, IModelItem
```

Methods

See *TQueue.*

Properties

See *TQueue.*

Delegates

```
public delegate void EventHandler( TStack sender,
                                   Transaction transaction )
```

Represents the method which will handle the event fired by stack *sender* for a specified *transaction*.

Events

See *TQueue*.

2.9. TQueuePrty

This component represents a queue with priorities. It differs from the regular queue (*TQueue*) by the way it handles the transactions. It handles them in accordance with the rule "First In – First Out within the same priority level". So, the transactions with higher priority will leave the queue first.

Another distinguishing feature of *TQueuePrty* is the ability of an incoming higher priority transaction to displace a lower priority transaction contained in the block. Such displacement is also called interruption. The interruption happens on the following conditions:

- The number of the transactions in the queue has reached its *Capacity*.
- The incoming transaction is interruptive, so its field *IsInterruptive* equals *true*.
- The queue is interruptible. This is defined by the property *isInterruptible*.
- The queue contains at least one transaction with the priority lower than the priority of the incoming transaction.

During the interruption by priority, the incoming high-priority transaction will displace one of the transactions with the lowest priority (in no particular order). The interruption fires the *OnInterruption* event for the displaced transaction.

What to do with the displaced (interrupted) transaction? One of the options is to send the transaction to another block while handling the *OnInterruption* event. If the transaction is not sent to another block, or if you don't handle this event at all, the transaction will be terminated.

When a transaction enters the queue, you may limit the waiting time in the queue (within the event *OnEnter*). Specify the maximum waiting time in the queue by using the *SetTime* method. The maximum waiting time may be described by a probability distribution. By assigning random numbers to the value of maximum waiting time we can simulate its statistical properties.

When the maximum waiting period for the transaction is ended, Delsi fires the event *AfterTimeEnded*. Within this event you can decide what to do with the transaction. One of the options is to send the transaction to another block. If the transaction is not sent to another block, or if you don't handle this event at all, the transaction will be terminated.

A queue with priorities may have limited capacity. If the number of transactions in the queue is less than its capacity, the queue is in the state of *"Ready to Get"*. If the queue contains at least one transaction, it is in the state of *"Ready to Send"*.

Syntax

```
public class TQueuePrty : TQueue, IBlock, ImodelItem
```

Methods

```
public double AverageCount()
```

Returns the average number of transactions in the block.

```
public double AverageTime()
```

Returns the average time spent by transactions in the block.

```
public double AverageTimeNonZero()
```

Returns the average non-zero time spent in the queue, i.e. the average calculated only for positive values.

```
public void ClearStatistics()
```

Clears simulation statistics accumulated in the block.

```
public virtual long Count()
```

Returns the current number of transactions in the block.

```
public double DeviationTime()
```

Returns the standard deviation of the time spent in the block.

```
public double DeviationTimeNonZero()
```

Returns the standard deviation of non-zero time spent in the queue, i.e. the standard deviation calculated only for positive values.

```
public long Entries()
```

Returns the total number of transactions which have entered the block.

```
public long Exits()
```

Returns the total number of transactions which have exited from the block.

```
public long ExitsByInterruption()
```

Returns the number of transactions which have exited from the queue as a result of interruption.

```
public virtual long ExitsByTimeLimit()
```

Returns the number of transactions which have left the queue because the maximum waiting period has expired.

```
public override bool isReadyToGet( Transaction transaction )
```

Returns *true* if the block is ready to receive a specific transaction. Block *TQueuePrty* is ready to receive an incoming transaction as long as it has not reached its capacity or if all of the following is true:

- The incoming transaction is interruptive, so its field *IsInterruptive* equals *true*.
- The queue is interruptible. This is defined by the property *isInterruptible*.
- The queue contains at least one transaction with a lower priority than the priority of the incoming transaction.

```
public bool isReadyToSend()
```

Returns *true* if the queue has a transaction to send. *TQueuePrty* is always ready to send a transaction if it has one.

```
public long MaxCount()
```

Returns the maximum number of transactions in the queue.

```
public int MaxPriority()
```

Returns the maximum priority of the transactions currently contained in the block.

```
public double MaxTime()
```

Returns the maximum time spent in the block.

```
public int MinPriority()
```

Returns the minimal priority of the transactions currently contained in the block.

```
public double MinTime()
```

Returns the minimal time spent in the block.

```
public override string Report()
```

Returns the text containing the standard statistical report for the block. For *TQueuePrty* the standard report looks like this.

```
Entries: 2444
Zero Entries: 7
Exits: 1993
Exits by time limit: 304
Exits by interruption: 0
Count: 451
Max count: 452
Average count: 238.27951824913
Usage: 0.997079347448628
Min time: 0
Max time: 1380
Average time: 471.552161543352
Deviation time: 549.787596369526
Average non-zero time: 472.500481626898
Deviation non-zero time: 549.932900949267
```

```
public override bool Send( IBlock block )
```

Tries to send the transaction which is ready to exit from the block, to another block. It returns *true* if the transaction was successfully sent; otherwise, *false*. When using the *Send* method, you have to meet the following conditions:

- For *TQueuePrty*, this method can be called only on handling the events *OnRouting*, *AfterTimeEnded* and *OnInterruption*.
- The block should have a transaction to send.
- The receiving block should belong to the same model as sending.
- The receiving block cannot be the same as sending.

If any of these conditions are not satisfied, Delsi Runtime Engine will throw *DelsiException*.

```
public virtual void SetTime( double time )
```

Sets maximum waiting time for the transaction entering the block. Use this method while handling the *OnEnter* event.

```
public double Usage()
```

Returns the relative part of simulation time when the block was containing at least one transaction.

```
public long ZeroEntries()
```

Returns the number of transaction entries into the block when the block was empty.

Properties

```
public int Capacity { get; set; }
```

Defines the maximum possible number of transactions in the block. If *Capacity* is 0, the number of transactions in the block is unlimited.

```
public bool isInterruptible { get; set; }
```

Defines if the block permits high-priority interruptions.

```
public string Caption { get; set; }
```

Gets or sets the text of the item caption.

```
public TModel Model{ get; }
```

Gets the model where the item belongs to.

```
public Object Tag { get; set; }
```

Gets or sets the object associated with the component.

Delegates

```
public delegate void EventHandler( TQueuePrty sender,
                                   Transaction transaction )
```

Represents the method which will handle the event fired by [queue with priorities] *sender* for a specified *transaction*.

Events

```
public virtual event TQueuePrty.EventHandler AfterTimeEnded
```

Occurs when the maximum waiting period for a transaction has expired. Here you can decide what to do with the transaction. One of the options is to send the transaction to another block. If the transaction is not sent to another block, or if you don't handle this event at all, the transaction will be terminated.

```
public virtual event TQueuePrty.EventHandler OnEnter
```

Occurs when a transaction enters the block. This is the place where you may set the maximum waiting time for the transaction using the *SetTime* method.

```
public virtual event TQueuePrty.EventHandler OnExit
```

Occurs when a transaction exits from the block.

```
public virtual event TQueuePrty.EventHandler OnInterruption
```

Occurs when a transaction is displaced by an incoming transaction with a higher priority. In this case the parameter *transaction* of the event handler will contain the reference to the displaced (low-priority) transaction. Within this event you can decide what to do with the transaction. One of the options is to send the transaction to another block. If the transaction is not sent to another block, or if you don't handle this event at all, the transaction will be terminated.

```
public virtual event TQueuePrty.EventHandler OnRouting
```

Fired when the block has a transaction ready to exit from the block. This is the place where the transaction will be sent to another block (see method *Send*). You can apply any sophisticated logic to determine where to send the transaction. If the transaction is not sent, it will remain in the block until the next *OnRouting* event will be fired by Delsi Runtime Engine.

2.10. TServer

The *TServer* component simulates a serving facility (server). Only one transaction at a time can be in service in *TServer.*

When a transaction enters the server, it is time to specify the service time for the transaction (see the *OnEnter*). Specify the service time by using the *SetTime* method.

The service time may be described by a probability distribution. By assigning random numbers to the service time we can simulate its statistical properties.

TServer provides the mechanism for the interruption by priority, which means that the incoming high-priority transaction can displace the low-priority transaction which is standing in service in the server.

The interruption by priority happens on the following conditions:

- The incoming transaction is interruptive, so its field *IsInterruptive* equals to *true*.
- The server is interruptible. This is defined by the property *isInterruptible*.
- The transaction standing in service has the priority lower than the priority of the incoming transaction.

During the interruption by priority, the incoming high-priority transaction will displace one of the transactions with the lowest priority (in no particular order). The interruption fires the *OnInterruption* event for the displaced transaction.

TServer can be paused and resumed. When a server is paused, the transaction staying in service will be displaced (regardless of the values of the transaction field *IsInterruptive* and the server property *IsInterruptible*). Pausing the server fires the *OnInterruption* event for the displaced transaction.

The following options can be applied for the displaced (interrupted) transaction (in cases of both high-priority interruption and the interruption caused by the paused server):

- Send the transaction to another block (using method *Send*) on handling the *OnInterruption* event.
- Suspend the transaction from service (using method *Suspend*) on handling the *OnInterruption* event.
- If the transaction was not suspended, sent to another block, or if you don’t handle this event at all, the transaction will be terminated.

Syntax

```
public class TServer : Component, IBlock, IModelItem
```

Methods

```
public double AverageCount()
```

Returns the average number of transactions in the block (including suspended).

```
public double AverageTime()
```

Returns the average time spent by all transactions in the block (including time suspended).

```
public double AverageTimeServed()
```

Returns the average time spent in the block by served transactions (including time suspended).

```
public void ClearStatistics()
```

Clears simulation statistics accumulated in the block.

```
public long Count()
```

Returns the current number of transactions in the block. For *TServer* this will include the transaction standing in service and the suspended transactions (if any).

```
public double DeviationTime()
```

Returns the standard deviation of the time spent by all transactions in the block (including time in suspension).

```
public double DeviationTimeServed()
```

Returns the standard deviation of the time spent in the block by served transactions (including time in suspension).

```
public long Entries()
```

Returns the total number of transactions which have entered the block.

```
public long Exits()
```

Returns the total number of transactions which have exited from the block.

```
public long ExitsByInterruption()
```

Returns the number of the transactions which have exited from the block as a result of interruption.

```
public long ExitsServed()
```

Returns the number of exited transactions whose service has been completed.

```
public bool isPaused()
```

Returns *true* if the server is paused by the *Pause* method.

```
public bool isReadyToGet( Transaction transaction )
```

Returns *true* if the block is ready to receive a specific transaction. *TServer* is ready to receive an incoming transaction if one of the following two scenarios is true.

Scenario 1: Both of the following conditions are true:

- The server doesn't contain a transaction.
- The server is not paused.

Scenario 2: All of the following conditions are true:

- The incoming transaction is interruptive, so its field *IsInterruptive* equals true.
- The server is interruptible. This is defined by its property *isInterruptible*.
- The transaction standing in service has a lower priority than the priority of the incoming transaction.
- The server is not paused.

```
public bool isReadyToSend()
```

Returns *true* if the block has a transaction to send. For *TServer* it means that the service has been completed for the contained transaction and the server is not paused by the *Pause* method.

```
public long MaxCount()
```

Returns the maximum number of transactions in the block.

```
public double MaxTime()
```

Returns the maximum time spent in the block.

```
public double MaxTimeServed()
```

Returns the maximum time spent in the block by served transactions (including time suspended).

```
public double MinTime()
```

Returns the minimal time spent in the block.

```
public double MinTimeServed()
```

Returns the minimal time spent in the block by served transactions (including time suspended).

```
public void Pause()
```

Pauses the server and displaces the transaction staying in service. The *Pause* method fires the event *OnInterruption* for that transaction. The server becomes unable to get or send transactions.

```
public string Report()
```

Returns the text containing the standard statistical report for the block. For *TServer* the standard report looks like this.

```
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
```

```
public void Resume()
```

Resumes the server. Enables the server to get or send transactions. If the server contains suspended transactions, the last suspended transaction will be restored in service for the rest of its service time.

```
public bool Send( IBlock block )
```

Tries to send the transaction which is ready to exit from the block, to another block. It returns *true* if the transaction was successfully sent; otherwise, *false*. When using the *Send* method, you have to meet the following conditions:

- For *TServer*, this method can be called only on handling the events *OnRouting* and *OnInterruption*.
- The block should have a transaction to send.
- The receiving block should belong to the same model as sending.
- The receiving block cannot be the same as sending.

If any of these conditions are not satisfied, Delsi Runtime Engine will throw *DelsiException*.

```
public void SetTime( double time )
```

Sets the service time for the transaction entering the server. Use this method while handling the *OnEnter* event. If you don't set the service time for an incoming transaction, the transaction will never exit from the server.

```
public void Suspend()
```

Suspends the transaction displaced from service. This method may be called only within the *OnInterruption* event which is caused either by high-priority interruption or by calling the *Pause* method. The suspended transactions are kept in the internal stack. After a high-priority transaction ends the service or after the *Resume* method is called, the last suspended transaction will be automatically restored in service for the rest of its service time. The server can store only one suspended transaction per priority level.

```
public double Usage()
```

Returns the relative part of simulation time when the block was containing at least one transaction.

Properties

```
public bool isInterruptible { get; set; }
```

Defines if the block permits high-priority interruptions. It doesn't affect the way the *Pause* method.

```
public string Caption { get; set;}
```

Gets or sets the text of the item caption.

```
public TModel Model{ get; }
```

Gets the model where the item belongs to.

```
public Object Tag { get; set; }
```

Gets or sets the object associated with the component.

Delegates

```
public delegate void EventHandler( TServer sender,
                                   Transaction transaction )
```

Represents the method which will handle the event fired by server *sender* for a specified *transaction*.

Events

```
public event TServer.EventHandler OnEnter
```

Occurs when a transaction enters the block. This is the place where you should set the service time for the transaction using the *SetTime* method. If you don't specify the service time for the transaction it will never exit from the server.

```
public event TServer.EventHandler OnExit
```

Occurs when a transaction exits from the block.

```
public event TServer.EventHandler OnInterruption
```

Occurs when a transaction is displaced by incoming transaction with higher priority, or when a transaction is displaced as a result of calling the *Pause* method. The reference to the displaced transaction will be passed to the event as a parameter. The following options can be applied for the displaced (interrupted) transaction:

- Send transaction to another block (see method *Send*)
- Suspend transaction from service (see method *Suspend*())
- If the transaction is not suspended, sent to another block, or if you don't handle this event at all, the transaction will be terminated.

```
public event TServer.EventHandler OnRouting
```

Fired when the service is completed and the serviced transaction is ready to leave the block. This is the place where the transaction should be sent to another block (see method *Send*). You can apply any sophisticated logic to determine where to send the transaction.If the transaction is not sent, it will remain in the block until the next OnRouting event will be fired by Delsi Runtime Engine. Note that an incoming transaction cannot displace the transaction which has already been served and is waiting for exit.

2.11. TMultiServer

This component represents simultaneous, multi-channel service of dynamic entities (transactions). *TMultiServer* is very similar to *TServer*. But in contrast to *TServer*, it may have capacity greater than 1.

When a transaction enters the multi-server, it's time to specify the service time for the transaction (see event *OnEnter*). Specify the service time by using the *SetTime* method.

The service time may be described by a probability distribution. By assigning random numbers to the service time we can simulate its statistical properties.

TMultiServer provides the mechanism of the interruption by priority, which means that the incoming high-priority transaction can displace a low-priority transaction which is standing in service in the multi-server. The interruption by priority happens on the following conditions:

- The number of the transactions in the multi-server has reached its *Capacity*.
- The incoming transaction is interruptive, so its field *IsInterruptive* equals true.
- The multi-server is interruptible. This is defined by the property *isInterruptible*.
- At least one transaction standing in service has a lower priority than the priority of the incoming transaction.
- The multi-server is not paused.

During the interruption by priority, the incoming high-priority transaction will displace one of the transactions with the lowest priority (in no particular order). The interruption fires the *OnInterruption* event for the displaced transaction.

TMultiServer can be paused and resumed. When a multi-server is paused, all transactions staying in service will be displaced (regardless of the values of the transaction field *IsInterruptive* and the multi-server property *IsInterruptible*). Pausing the multi-server fires the *OnInterruption* event for each of the displaced transactions.

The following options can be applied for the displaced (interrupted) transaction (in cases of both high-priority interruption and the interruption caused by the paused multi-server):

- Send the transaction to another block (using method *Send*) on handling the *OnInterruption* event.
- Suspend the transaction from service (using method *Suspend*) on handling the *OnInterruption* event.
- If the transaction was not suspended, sent to another block, or if you don’t handle the *OnInterruption* event at all, the transaction will be terminated.

Syntax

```
public class TMultiServer : TQueuePrty, IBlock, IModelItem
```

Methods

```
public double AverageCount()
```

Returns the average number of transactions in the block (including suspended).

```
public double AverageCountOnService()
```

Returns the average number of transactions standing on service in the block (non including suspended).

```
public double AverageTime()
```

Returns the average time spent by all transactions in the block (including time suspended).

```
public double AverageTimeServed()
```

Returns the average time spent in the block by served transactions (including time suspended).

```
public void ClearStatistics()
```

Clears simulation statistics accumulated in the block.

```
public override long Count()
```

Returns the current number of transactions in the block. For *TMultiServer*, the current number of transactions in the block includes the transactions in service, the suspended transactions (if any) and the transactions waiting for the opportunity to exit from the block after the service is done.

```
public double DeviationTime()
```

Returns the standard deviation of the time spent by all transactions in the block (including time suspended).

```
public double DeviationTimeServed()
```

Returns the standard deviation of the time spent in the block by served transactions (including time suspended).

```
public long Entries()
```

Returns the total number of transactions which have entered the block.

```
public long Exits()
```

Returns the total number of transactions which have exited from the block.

```
public long ExitsByInterruption()
```

Returns the number of the transactions which have exited from the block as a result of interruption.

```
public virtual long ExitsByTimeLimit()
```

Although this method is inherited from *TQueue*, for *TMultiServer* it doesn't make sense, so it always returns 0.

```
public long ExitsServed()
```

Returns the number of exited transactions, whose service has been completed.

```
public bool isPaused()
```

Returns *true* if the multi-server is paused by the *Pause* method.

```
public bool isReadyToGet( Transaction transaction )
```

Returns *true* if the block is ready to receive a specific transaction. *TMultiServer* is ready to receive an incoming transaction if one of the following two scenarios is true:

Scenario 1: Both of the following conditions are true:

- The multi-server has not reached its capacity.
- The multi-server is not paused.

Scenario 2: All of the following conditions are true:

- The number of the transactions in the multi-server has reached its *Capacity*.
- The incoming transaction is interruptive, so its field *IsInterruptive* equals *true*.
- The multi-server is interruptible. This is defined by its property *isInterruptible*.
- At least one transaction standing in service has a lower priority than the priority of the incoming transaction.
- The multi-server is not paused.

```
public bool isReadyToSend()
```

Returns *true* if the block has a transaction to send. For *TMultiServer* it means that the service has been completed for some of the contained transactions and the multi-server is not paused by *Pause* method.

```
public long MaxCount()
```

Returns maximum number of transactions in the block.

```
public int MaxPriority()
```

Returns the maximum priority of the transactions currently contained in the block.

```
public double MaxTime()
```

Returns maximum time spent in the block.

```
public double MaxTimeServed()
```

Returns the maximum time spent in the block by served transactions (including time suspended).

```
public int MinPriority()
```

Returns the minimal priority of the transactions currently contained in the block.

```
public double MinTime()
```

Returns minimal time spent in the block.

```
public double MinTimeServed()
```

Returns minimal time spent in the block by served transactions (including time suspended).

```
public void Pause()
```

Pauses the multi-server and displace the transactions staying in service. The *Pause* method fires the *OnInterruption* event for all of the transactions which are currently in service, one by one. The multi-server becomes unable to get or send transactions.

```
public string Report()
```

Returns the text containing the standard statistical report for the block. For *TMultiServer* the standard report looks like this.

```
Entries: 997
Exits: 993
Exits served: 872
Exits by interruption: 121
Count: 4
Count on service: 4
Max count: 8
Average count: 4.30784767622247
Average count on service: 3.62645142238266
Usage: 0.989708034414055
Min time: 0.00252844857936907
Max time: 44.9997848889686
Average time: 4.32542516111157
Deviation time: 3.35080428178412
Average time served: 4.76218252080034
Deviation time served: 3.32871507905152
```

```
public void Resume()
```

Resumes the multi-server. Enable the multi-server to get or send transactions. If the multi-server has suspended transactions, they will be restored in service for the rest of their service time. The transactions will be restored one by one starting with the most recently suspended transactions until the number of restored transactions reaches the capacity of the multi-server.

```
public bool Send( IBlock block )
```

Tries to send the transaction which is ready to exit from the block, to another block. It returns *true* if the transaction was successfully sent; otherwise, *false*. When using the *Send* method, you have to meet the following conditions:

- For *TMultiServer*, this method can be called only on handling the events *OnRouting* and *OnInterruption*.
- The block should have a transaction to send.
- The receiving block should belong to the same model as sending.
- The receiving block cannot be the same as sending.

If any of these conditions are not satisfied, Delsi Runtime Engine will throw *DelsiException*.

```
public void SetTime( double time )
```

Sets the service time for the transaction entering the multi-server. Use this method while handling the *OnEnter* event. If you don't set the service time for the incoming transaction, the transaction will never exit from the multi-server.

```
public void Suspend()
```

Suspends the transaction displaced from service. This method may be called only within the *OnInterruption* event which is caused either by high-priority interruptions or by calling the *Pause* method.

The suspended transactions are kept in the internal stack in LIFO order. After a high-priority transaction ends the service, the last suspended transaction will be automatically restored in service for the rest of its service time. Because *TMultiServer* can contain multiple transactions, the internal stack of suspended transactions can contain multiple transactions per priority.

```
public double Usage()
```

Returns the relative part of simulation time when the block was containing at least one transaction.

Properties

```
public int Capacity{ get; set; }
```

Defines the maximum possible number of transactions in the block. If *Capacity* is 0, the number of transactions in the block is unlimited.

```
public bool isInterruptible { get; set; }
```

Defines if the block permits high-priority interruptions. It doesn't affect the way the *Pause* method.

```
public string Caption { get; set;}
```

Gets or sets the text of the item caption.

```
public TModel Model{ get; }
```

Gets the model where the item belongs to.

```
public Object Tag { get; set; }
```

Gets or sets the object associated with the component.

Delegates

```
public delegate void EventHandler( TMultiServer sender,
                                   Transaction transaction )
```

Represents the method which will handle the event fired by server *sender* for a specified *transaction*.

Events

```
public event TMultiServer.EventHandler OnEnter
```

Occurs when a transaction enters the block. This is the place where you should set the service time for the transaction using the *SetTime* method. If you don't specify the service time for the transaction it will never exit from the multi-server.

```
public event TMultiServer.EventHandler OnExit
```

Occurs when a transaction exits from the block.

```
public event TMultiServer.EventHandler OnInterruption
```

Occurs when a transaction is displaced by incoming transaction with higher priority, or when a transaction is displaced as a result of calling the *Pause* method. The reference to the displaced transaction will be passed to the event as a parameter. The following options can be applied for the displaced (interrupted) transaction:

- Send the transaction to another block.
- Suspend the transaction from service.
- If the transaction was not suspended, sent to another block, or if you don’t handle the event *OnInterruption* at all, the transaction will be terminated.

```
public event TMultiServer.EventHandler OnRouting
```

Fired when the service is completed and the serviced transaction is ready to leave the block. This is the place where the transaction should be sent to another block (see method *Send*). You can apply any sophisticated logic to determine where to send the transaction. If the transaction was not sent, it will remain in the block until the next *OnRouting* event will be fired by Delsi Runtime Engine. Note that an incoming transaction cannot displace the transaction which has already been served and is waiting for exit.

2.12. TDelay

This component delays transactions for the specified duration. The block can contain multiple transactions at a time.

When a transaction enters the block, it is time to specify the delay time for the transaction (see the *OnEnter* event). Specify the delay duration by using the *SetTime* method.

The delay duration may be described by a probability distribution. By assigning random numbers to the delay duration we can simulate its statistical properties.

Practically, *TDelay* would be identical to *TMultiServer* with its property *IsInterruptable* = 1. Why do we need this *TDelay* as a separate component at all? There are two reasons:

- *TDelay* works more efficiently in terms of performance
- It is more logical to use for the cases when we need just to delay transactions.

Syntax

```
public class TDelay : TQueue, IBlock, IModelItem
```

Methods

```
public double AverageCount()
```

Returns the average number of transactions in the block.

```
public double AverageTime()
```

Returns the average time spent by transactions in the block.

```
public void ClearStatistics()
```

Clears simulation statistics accumulated in the block.

```
public virtual long Count()
```

Returns the current number of transactions in the block.

```
public double DeviationTime()
```

Returns the standard deviation of the time spent in the block.

```
public long Entries()
```

Returns the total number of transactions which have entered the block.

```
public long Exits()
```

Returns the total number of transactions which have exited from the block.

```
public virtual long ExitsByTimeLimit()
```

Although this method is inherited from *TQueue*, for *TDelay* it doesn't make sense, so it always returns 0.

```
public virtual bool isReadyToGet( Transaction transaction )
```

Returns *true* if the block is ready to receive a specific transaction. *TDelay* is always ready to get a transaction as long as it has not reached its capacity.

```
public bool isReadyToSend()
```

Returns *true* if the queue has a transaction to send. For *TDelay* it means that it has at least for one transaction the delay period ended.

```
public long MaxCount()
```

Returns the maximum number of transactions in the block.

```
public double MaxTime()
```

Returns the maximum time spent in the block.

```
public double MinTime()
```

Returns the minimal time spent in the block.

```
public virtual string Report()
```

Returns the text containing the standard statistical report for the block. For *TDelay* the standard report looks like this.

```
Entries: 154
Exits: 154
Count: 0
Max count: 6
Average count: 0.618154960830465
Usage: 0.383614163105583
Min time: 3.00967335701216
Max time: 4.99854302452843
Average time: 4.01241866977533
Deviation time: 0.572330914267769
```

```
public virtual bool Send( IBlock block )
```

Tries to send the transaction, which is ready to exit from the block, to another block. It returns *true* if the transaction was successfully sent; otherwise, *false*. When using the *Send* method, you have to meet the following conditions:

- For *TDelay*, this method can be called only on handling the *OnRouting* event.
- The block should have a transaction to send.
- The receiving block should belong to the same model as sending.
- The receiving block cannot be the same as sending.

If these conditions are not satisfied, Delsi Runtime Engine will throw *DelsiException*.

```
public virtual void SetTime( double time )
```

Sets the delay time for the transaction entering the block. Use this method while handling the *OnEnter* event. If you don't set the delay period for the incoming transaction, the transaction will never exit from the block.

```
public double Usage()
```

Returns the relative part of simulation time when the block was containing at least one transaction.

```
public long ZeroEntries()
```

Returns the number of transaction entries into the block, when the block was empty.

Properties

```
public int Capacity{ get; set; }
```

Defines the maximum possible number of transactions in the block. When the number of transactions reaches its capacity, the block becomes *"Not Ready to Get"*. If *Capacity* is 0, the number of transactions in the block is unlimited.

```
public string Caption { get; set;}
```

Gets or sets the text of the item caption.

```
public TModel Model{ get; }
```

Gets the model where the item belongs to.

```
public Object Tag { get; set; }
```

Gets or sets the object associated with the component.

Delegates

```
public delegate void EventHandler( TDelay sender,
                                   Transaction transaction )
```

Represents the method which will handle the event fired by the delay block *sender* for a specified *transaction*.

Events

```
public virtual event TDelay.EventHandler AfterTimeEnded
```

Although inherited, never fired for *TDelay*.

```
public virtual event TDelay.EventHandler OnEnter
```

Occurs when a transaction enters the block. This is the place where you should set the delay time for the transaction using the *SetTime* method.

```
public virtual event TDelay.EventHandler OnExit
```

Occurs when a transaction exits from the block.

```
public virtual event TDelay.EventHandler OnRouting
```

Fired when the block has a transaction ready to exit from the block. This is the place where the transaction will be sent to another block (see method *Send*). You can apply any sophisticated logic to determine where to send the transaction. If the transaction is not sent, it will remain in the block until the next *OnRouting* event will be fired by Delsi Runtime Engine.

2.13. TEmitter

This component creates portions of transactions on demand. After the transactions are created, *TEmitter* becomes *"Ready to Send"* until all of the created transactions exit from the block.

Syntax

```
public class TEmitter : Component, IBlock, IModelItem
```

Methods

```
public void ClearStatistics()
```

Clears simulation statistics accumulated in the block.

```
public virtual long Count()
```

Returns the current number of transactions in the block.

```
public void Emit( int Amount )
```

Immediately creates the specified amount of transactions. After that, the block becomes *"Ready to Send"* and Delsi Runtime Engine tries to push generated transactions through the model.

```
public long Exits()
```

Returns the total number of transactions which have exited from the block.

```
public virtual bool isReadyToGet( Transaction transaction )
```

TEmitter cannot receive a transaction, so this method always return *false*.

```
public bool isReadyToSend()
```

Returns *true* if the block has a transaction to send. If the emitter is not empty, it is ready to send a transaction.

```
public string Report()
```

Returns the text containing the standard statistical report for the block. For *TEmitter* the standard report looks like this.

```
Exits: 24
Count: 2
```

```
public bool Send( IBlock block )
```

Tries to send the transaction, which is ready to exit from the block, to another block. It returns *true* if the transaction was successfully sent; otherwise, *false*. When using the *Send* method, you have to meet the following conditions:

- For *TEmitter*, this method can be called only on handling the *OnRouting* event.
- The block should have a transaction to send.
- The receiving block should belong to the same model as sending.
- The receiving block cannot be the same as sending.

If any of these conditions are not satisfied, Delsi Runtime Engine will throw *DelsiException*.

Properties

```
public string Caption { get; set;}
```

Gets or sets the text of the item caption.

```
public TModel Model{ get; }
```

Gets the model where the item belongs to.

```
public Object Tag { get; set; }
```

Gets or sets the object associated with the component.

Delegates

```
public delegate void EventHandler( TEmitter sender,
                                   Transaction transaction )
```

Represents the method which will handle the event fired by multi-server *sender* for a specified *transaction*.

Events

```
public event TEmitter.EventHandler AfterGeneration
```

Occurs right after a transaction has been generated. This is a good place to setup the transaction properties.

```
public event TEmitter.EventHandler OnExit
```

Occurs when a transaction exits from the block.

```
public event TEmitter.EventHandler OnRouting
```

Fired when the block has a transaction ready to exit from the block. This is the place where the transaction will be sent to another block. (see method *Send*). You can apply any sophisticated logic to determine where to send the transaction. If the transaction is not sent, it will remain in the block until the next *OnRouting* event will be fired by Delsi Runtime Engine.

2.14. TTerminator

The *TTerminator* component terminates incoming transactions according to the termination factor (see *TerminationFactor*). The termination factor defines how many transactions will be selectively terminated in a fixed series of sequentially incoming transactions:

- If the termination factor is 1 then each incoming transaction will be terminated.
- If the termination factor is N (where N>1) then the first N-1 transactions out of N will be terminated and 1 will get ready for exiting from the terminator.

The implication of the termination factor on incoming transactions can be illustrated by the following table.

Termination Factor	Number of transactions to be terminated	Number of transactions to go through	Comment
1	1	0	
2	1	1	
3	2	1	
4	3	1	
N	N-1	1	where N>1

Example 1: If the termination factor is 4, it means that the first 3 out of 4 sequentially incoming transactions will be terminated and 1 will be ready to exit from the terminator. In this case we may think that any four incoming transactions will be assembled into one.

Example 2: If the termination factor is 1, all incoming transactions will be terminated.

Syntax

```
public class TTerminator : Component, IBlock, IModelItem
```

Methods

```
public void ClearStatistics()
```

Clears simulation statistics accumulated in the block.

```
public long Count()
```

Returns the current number of transactions in the block.

```
public long Entries()
```

Returns the total number of transactions which have entered the block.

```
public long Exits()
```

Returns the total number of transactions which have exited from the block.

```
public bool isReadyToGet( Transaction transaction )
```

Returns *true* if the block is ready to receive a specific transaction. *TTerminator* always returns *true*.

```
public bool isReadyToSend()
```

Returns *true* if the block has a transaction to send.

```
public string Report()
```

Returns the text containing the standard statistical report for the block. For *TTerminator* the standard report looks like this.

```
Entries: 872
Exits: 0
Count: 0
```

```
public bool Send( IBlock block )
```

Tries to send the transaction, which is ready to exit from the block, to another block. When using the *Send* method, you have to meet the following conditions:

- For *TTerminator*, this method can be called only on handling the *OnRouting* event.
- The block should have a transaction to send.
- The receiving block should belong to the same model as sending.
- The receiving block cannot be the same as sending.

If any of these conditions are not satisfied, Delsi Runtime Engine will throw *DelsiException*.

Properties

```
public int TerminationFactor { get; set; }
```

Get or sets the termination factor. The termination factor defines how many transactions will be selectively terminated in a fixed series of sequentially incoming transactions:

- If the termination factor is 1 then each incoming transaction will be terminated.
- If the termination factor is N (where N>1) then the first N-1 transactions out of N will be terminated and 1 will get ready for exiting from the terminator

```
public string Caption { get; set;}
```

Gets or sets the text of the item caption.

```
public TModel Model{ get; }
```

Gets the model where the item belongs to.

```
public Object Tag { get; set; }
```

Gets or sets the object associated with the component.

Delegates

```
public delegate void EventHandler( TTerminator sender,
                                   Transaction transaction )
```

Represents the method which will handle the event fired by terminator *sender* for a specified *transaction*.

Events

```
public event TTerminator.EventHandler OnEnter
```

Occurs when a transaction enters the block.

```
public event TTerminator.EventHandler OnExit
```

Occurs when a transaction exits from the block.

```
public event TTerminator.EventHandler OnRouting
```

Fired when the block has a transaction ready to exit from the block. This is the place where the transaction will be sent to another block. (see method *Send*). You can apply any sophisticated logic to determine where to send the transaction. If the transaction is not sent, it will remain in the block until the next *OnRouting* event will be fired by Delsi Runtime Engine.

2.15. TSplitter

This block splits one incoming transaction into several outgoing transactions. The number of outgoing transactions is defined by property *SplitFactor*. All outgoing transactions have the same *Id* as the incoming one, but they have unique values of *CopyId.*

Note: The original transactions generated by *TGenerator* or *TEmitter* always have *CopyId* = 1.

Splitting here actually means cloning the incoming transaction with copying all its properties except *CopyId*. The outgoing transactions exits from *TSplitter* one by one.

Syntax

```
public class TSplitter : Component, IBlock, ImodelItem
```

Methods

```
public void ClearStatistics()
```

Clears simulation statistics accumulated in the block.

```
public long Count()
```

Returns the current number of transactions in the block.

```
public long Entries()
```

Returns the total number of transactions which have entered the block.

```
public long Exits()
```

Returns the total number of transactions which have exited from the block.

```
public bool isReadyToGet( Transaction transaction )
```

Returns *true* if the block is ready to receive a specific transaction. *TSplitter* always return *true*.

```
public bool isReadyToSend()
```

Returns *true* if the block has a transaction to send. *TSplitter* is “ready to send” if it is not empty.

```
public string Report()
```

Returns the text containing the standard statistical report for the block. For *TSplitter* the standard report looks like this.

```
Entries: 32
Exits: 64
Count: 0
```

```
public bool Send( IBlock block )
```

Tries to send the transaction, which is ready to exit from the block, to another block. When using the *Send* method, you have to meet the following conditions:

- For *TSplitter*, this method can be called only on handling the *OnRouting* event.
- The block should have a transaction to send.
- The receiving block should belong to the same model as sending.
- The receiving block cannot be the same as sending.

If any of these conditions are not satisfied, Delsi Runtime Engine will throw *DelsiException*.

Properties

```
public int SplitFactor { get; set; }
```

Gets or sets the split factor, which defines how many outgoing transactions the incoming transaction will be split into. The default value is 1, meaning no splitting will be done.

```
public string Caption { get; set;}
```

Gets or sets the text of the item caption.

```
public TModel Model{ get; }
```

Gets the model where the item belongs to.

```
public Object Tag { get; set; }
```

Gets or sets the object associated with the component.

Delegates

```
public delegate void EventHandler( TSplitter sender,
                                   Transaction transaction )
```

Represents the method which will handle the event fired by splitter *sender* for a specified *transaction*.

Events

```
public event TSplitter.EventHandler OnEnter
```

Occurs when a transaction enters the block.

```
public event TSplitter.EventHandler OnExit
```

Occurs when a transaction exits from the block.

```
public event TSplitter.EventHandler OnRouting
```

Fired when the block has a transaction ready to exit from the block. This is the place where the transaction will be sent to another block. (see method *Send*). You can apply any sophisticated logic to determine where to send the transaction. If the transaction is not sent, it will remain in the block until the next *OnRouting* event will be fired by Delsi Runtime Engine.

2.16. TJoiner

This component joins several incoming transactions with the same *Id* into one outgoing transaction. The number of incoming transactions with the same *Id* which are to be joined into one outgoing transaction is defined be the property *JoinFactor*.

The joining works the way that the incoming transactions with the same *Id* wait in the block *TJoiner* until their number reaches *JoinFactor*. After that the transaction with the smallest *CopyId* becomes ready to exit from the block; all other transactions (of the same *Id*) will be terminated.

Syntax

```
public class TJoiner : Component, IBlock, IModelItem
```

Methods

```
public void ClearStatistics()
```

Clears simulation statistics accumulated in the block.

```
public long Count()
```

Returns the current number of transactions in the block.

```
public long Entries()
```

Returns the total number of transactions which have entered the block.

```
public long Exits()
```

Returns the total number of transactions which have exited from the block.

```
public bool isReadyToGet( Transaction transaction )
```

Returns *true* if the block is ready to receive a specific transaction. *TJoiner* is always ready to get a transaction.

```
public bool isReadyToSend()
```

Returns *true* if the block has a joined transaction to send.

```
public string Report()
```

Returns the text containing the standard statistical report for the block. For *TJoiner* the standard report looks like this.

```
Entries: 12
Exits: 36
Count: 0
```

```
public bool Send( IBlock block )
```

Tries to send the transaction, which is ready to exit from the block, to another block. When using the *Send* method, you have to meet the following conditions:

- For *TJoiner*, this method can be called only on handling the *OnRouting* event.
- The block should have a transaction to send.
- The receiving block should belong to the same model as sending.
- The receiving block cannot be the same as sending.

If any of these conditions are not satisfied, Delsi Runtime Engine will throw *DelsiException*.

Properties

```
public int JoinFactor { get; set; }
```

Gets or sets the join factor, which defines how many incoming transactions of the same *Id* will be joined into one outgoing transaction.

```
public string Caption { get; set;}
```

Gets or sets the text of the item caption.

```
public TModel Model{ get; }
```

Gets the model where the item belongs to.

```
public Object Tag { get; set; }
```

Gets or sets the object associated with the component.

Delegates

```
public delegate void EventHandler( TJoiner sender,
                                   Transaction transaction )
```

Represents the method which will handle the event fired by joiner *sender* for a specified *transaction*.

Events

```
public event TJoiner.EventHandler OnEnter
```

Occurs when a transaction enters the block.

```
public event TJoiner.EventHandler OnExit
```

Occurs when a transaction exits from the block.

```
public event TJoiner.EventHandler OnRouting
```

Fired when the block has a transaction ready to exit from the block. This is the place where the transaction will be sent to another block. (see method *Send*). You can apply any sophisticated logic to determine where to send the transaction. If the transaction is not sent, it will remain in the block until the next *OnRouting* event will be fired by Delsi Runtime Engine.

2.17. TGate

This block works as a simple gate for the transactions. You can *lock*, *unlock* or *inverse* the gate. If the gate is locked, it is not "ready to get" a transaction, so no transaction can go through the gate.

Syntax

```
public class TGate : Component, IBlock, IModelItem
```

Methods

```
public void ClearStatistics()
```

Clears simulation statistics accumulated in the block.

```
public long Count()
```

Returns the current number of transactions in the block.

```
public long Entries()
```

Returns the total number of transactions which have entered the block.

```
public long Exits()
```

Returns the total number of transactions which have exited from the block.

```
public void Inverse()
```

Inverses the gate. Locks the gate if it is unlocked and vice versa.

```
public bool isReadyToGet( Transaction transaction )
```

Returns *true* if the block is ready to receive a specific transaction. The gate is ready to get a transaction as far it is unlocked and doesn't contain any other transaction.

```
public bool isReadyToSend()
```

Returns *true* if the block has a transaction to send. After a transaction entered the gate it becomes ready to send it.

```
public void Lock()
```

Locks the gate.

```
public string Report()
```

Returns the text containing the standard statistical report for the block. For *TGate* the standard report looks like this.

```
Entries: 36
Exits: 36
Count: 0
```

```
public bool Send( IBlock block )
```

Tries to send the transaction, which is ready to exit from the block, to another block. When using the *Send* method, you have to meet the following conditions:

- For *TGate*, this method can be called only on handling event *OnRouting*.
- The block should have a transaction to send.
- The receiving block should belong to the same model as sending.
- The receiving block cannot be the same as sending.

If any of these conditions are not satisfied, Delsi Runtime Engine will throw *DelsiException*.

```
public void Unlock()
```

Unlocks the gate.

Properties

```
public bool Locked { get; }
```

Returns *true* if the gate is locked and vice versa.

```
public string Caption { get; set;}
```

Gets or sets the text of the item caption.

```
public TModel Model{ get; }
```

Gets the model where the item belongs to.

```
public Object Tag { get; set; }
```

Gets or sets the object associated with the component.

Delegates

```
public delegate void EventHandler( TGate sender,
                                   Transaction transaction )
```

Represents the method which will handle the event fired by gate *sender* for a specified *transaction*.

Events

```
public event TJoiner.EventHandler OnEnter
```

Occurs when a transaction enters the block.

```
public event TJoiner.EventHandler OnExit
```

Occurs when a transaction exits from the block.

```
public event TJoiner.EventHandler OnRouting
```

Fired when the block has a transaction ready to exit from the block. This is the place where the transaction will be sent to another block. (see method *Send*). You can apply any sophisticated logic to determine where to send the transaction. If the transaction is not sent, it will remain in the block until the next *OnRouting* event will be fired by Delsi Runtime Engine. The transaction remaining in the gate will jam it.

2.18. TPicker

This block is able to choose the sending block from which it will accept the next incoming transaction. *TPicker* stores the *picking list*, i.e. the list of admissible blocks-senders. In order to maintain the picking list, use the *AddToPicking* and *RemoveFromPicking* methods.

TPicker can use one of the following picking rules:

- **RoundRobin** - The picking algorithm repeatedly runs through a list of source blocks, giving each of them the opportunity to send a transaction.
- **Random** - The source components are to be chosen randomly.
- **Prioritized** - The blocks in the picking list are prioritized by the rule "first added block has the highest priority, last added block has the lowest priority". For this rule it does matter in which order we are adding the blocks into the picking list.

Those picking rules are defined in the *PickingRules* enumeration. In order to set the picking rule, use the *PickingRule* property.

Syntax

```
public class TPicker : Component, IBlock, ImodelItem
```

Methods

```
public void AddToPicking( IBlock block )
```

Adds a specified block to the picking list. There are the following restrictions when adding:

- *TPicker* cannot be added to its own picking list.
- You cannot add the same block to the picking list twice.

If such restrictions are violated, Delsi Runtime Engine throws *DelsiException.*

```
public long Count()
```

Returns the current number of transactions in the block.

```
public void ClearStatistics()
```

Clears simulation statistics accumulated in the block.

```
public long Entries()
```

Returns the total number of transactions which have entered the block.

```
public long Exits()
```

Returns the total number of transactions which have exited from the block.

```
public bool isReadyToGet( Transaction transaction )
```

Returns *true* if the block is ready to receive a specific transaction. The method returns *true* if all of the following conditions are met:

- The block doesn't contain a transaction.
- The picking list is not empty.
- The block where the transaction currently resides satisfies the criteria of the picking rule.

```
public bool isReadyToSend()
```

Returns *true* if the block has a transaction to send. If a picker has a transaction, it is ready to send it.

```
public void RemoveFromPicking( IBlock block )
```

Removes a specified block from the picking list, if it's there.

```
public string Report()
```

Returns the text containing the standard statistical report for the block. For *TPicker* the standard report looks like this.

```
Entries: 128
Exits: 127
Count: 1
```

```
public bool Send( IBlock block )
```

Tries to send the transaction, which is ready to exit from the block, to another block. When using the *Send* method, you have to meet the following conditions:

- For *TPicker*, this method can be called only on handling the *OnRouting* event.
- The block should have a transaction to send.
- The receiving block should belong to the same model as sending.
- The receiving block cannot be the same as sending.

If any of these conditions are not satisfied, Delsi Runtime Engine throws *DelsiException*.

```
public void SetPick( IBlock block )
```

Explicitly sets the source block for the next picking. If the specified block is not in the picking list, Delsi Runtime Engine throws *DelsiException*.

Properties

```
public PickingRules PickingRule { get; set; }
```

Defines the picking rule.

```
public string Caption { get; set;}
```

Gets or sets the text of the item caption.

```
public TModel Model{ get; }
```

Gets the model where the item belongs to.

```
public Object Tag { get; set; }
```

Gets or sets the object associated with the component.

Delegates

```
public delegate void EventHandler( TPicker sender,
                                   Transaction transaction )
```

Represents the method which will handle the event fired by picker *sender* for a specified *transaction*.

Events

```
public event TPicker.EventHandler OnEnter
```

Occurs when a transaction enters the block.

```
public event TPicker.EventHandler OnExit
```

Occurs when a transaction exits from the block.

```
public event TPicker.EventHandler OnRouting
```

Fired when the block has a transaction ready to exit from the block. This is the place where the transaction will be sent to another block. (see method *Send*). You can apply any sophisticated logic to determine where to send the transaction. If the transaction is not sent, it will remain in the block until the next *OnRouting* event will be fired by Delsi Runtime Engine.

2.19. TBatcher

This block packs several incoming transactions into one outgoing transaction called a batch. The number of transactions contained in a batch is defined by a *batch factor*. The transactions packed in the *TBatcher* will travel inside their batch transaction until it enters *TUnbatcher*.

The nesting of transactions inside a batch can be recursive, in other words a batch can contain other batches.

Syntax

```
public class TBatcher : Component, IBlock, ImodelItem
```

Methods

```
public long Count()
```

Returns the current number of transactions in the block. This includes a number of batch transactions waiting for exit and the transaction which is in the process of batching (filling-up) with incoming transactions.

```
public void ClearStatistics()
```

Clears simulation statistics accumulated in the block.

```
public long Entries()
```

Returns the total number of transactions which have entered the block.

```
public long Exits()
```

Returns the total number of transactions which have exited from the block.

```
public bool isReadyToGet( Transaction transaction )
```

Returns *true* if the block is ready to receive a specific transaction. *TBatcher* is always ready to get a transaction.

```
public bool isReadyToSend()
```

Returns *true* if the block has a batch-transaction to send.

```
public string Report()
```

Returns the text containing the standard statistical report for the block. For *TBatcher* the standard report looks like this.

```
                        Entries: 38
                        Exits: 7
                        Count: 1
```

```
public bool Send( IBlock block )
```

Tries to send the transaction, which is ready to exit from the block, to another block. When using the *Send* method, you have to meet the following conditions:

- For *TBatcher*, this method can be called only on handling the *OnRouting* event.
- The block should have a transaction to send.
- The receiving block should belong to the same model as sending.
- The receiving block cannot be the same as sending.

If any of these conditions are not satisfied, Delsi Runtime Engine throws *DelsiException*.

Properties

```
public int BatchFactor { get; set; }
```

Gets or sets the batch factor, which defines defines how many incoming transactions will be packed into one outgoing batch transaction. The default value is 1.

```
public string Caption { get; set;}
```

Gets or sets the text of the item caption.

```
public TModel Model{ get; }
```

Gets the model where the item belongs to.

```
public Object Tag { get; set; }
```

Gets or sets the object associated with the component.

Delegates

```
public delegate void EventHandler( TBatcher sender,
                                   Transaction transaction )
```

Represents the method which will handle the event fired by splitter *sender* for a specified *transaction*.

Events

```
public event TBatcher.EventHandler OnEnter
```

Occurs when a transaction enters the block.

```
public event TBatcher.EventHandler OnExit
```

Occurs when a transaction exits from the block..

```
public event TBatcher.EventHandler OnRouting
```

Fired when the block has a transaction ready to exit from the block. This is the place where the transaction will be sent to another block. (see method *Send*). You can apply any sophisticated logic to determine where to send the transaction. If the transaction is not sent, it will remain in the block until the next *OnRouting* event will be fired by Delsi Runtime Engine.

2.20. TUnbatcher

This block unpacks transactions from an incoming batch transaction. The unpacking is not recursive. So, the component unpacks transactions only one level down. If the incoming transaction is not a batch, it will be seamlessly forwarded to exit.

Syntax

```
public class TUnatcher : Component, IBlock, ImodelItem
```

Methods

```
public long Count()
```

Returns the current number of transactions in the block. The transactions contained in *TUnbatcher* are those which were extracted from a batch and waiting to exit from the block.

```
public void ClearStatistics()
```

Clears simulation statistics accumulated in the block.

```
public long Entries()
```

Returns the total number of transactions which have entered the block.

```
public long Exits()
```

Returns the total number of transactions which have exited from the block.

```
public bool isReadyToGet( Transaction transaction )
```

Returns *true* if the block is ready to receive a specific transaction. *TUnbatcher* is always ready to get a transaction.

```
public bool isReadyToSend()
```

Returns *true* if the block has a transaction to send.

```
public string Report()
```

Returns the text containing the standard statistical report for the block. For *TUnbatcher* the standard report looks like this.

```
Entries: 7
Exits: 33
Count: 2
```

```
public bool Send( IBlock block )
```

Tries to send the transaction, which is ready to exit from the block, to another block. When using the *Send* method, you have to meet the following conditions:

- For *TUnbatcher*, this method can be called only on handling the *OnRouting* event.
- The block should have a transaction to send.
- The receiving block should belong to the same model as sending.
- The receiving block cannot be the same as sending.

If any of these conditions are not satisfied, Delsi Runtime Engine throws *DelsiException*.

Properties

```
public string Caption { get; set;}
```

Gets or sets the text of the item caption.

```
public TModel Model{ get; }
```

Gets the model where the item belongs to.

```
public Object Tag { get; set; }
```

Gets or sets the object associated with the component.

Delegates

```
public delegate void EventHandler( TUnbatcher sender,
                                   Transaction transaction )
```

Represents the method which will handle the event fired by splitter *sender* for a specified *transaction*.

Events

```
public event TUnbatcher.EventHandler OnEnter
```

Occurs when a transaction enters the block.

```
public event TUnbatcher.EventHandler OnExit
```

Occurs when a transaction exits from the block..

```
public event TUnbatcher.EventHandler OnRouting
```

Fired when the block has a transaction ready to exit from the block. This is the place where the transaction will be sent to another block. (see method *Send*). You can apply any sophisticated logic to determine where to send the transaction. If the transaction is not sent, it will remain in the block until the next *OnRouting* event will be fired by Delsi Runtime Engine.

2.21. TAssembler

This block assembles incoming transactions into one outgoing transaction according to an assembling scheme. The assembling scheme defines how many of different parts needed to assemble a final product. The scheme can be imagined as a list or records, where each record contains part ID and its corresponding amount. The direct analogy of the assembling scheme in manufacturing is a Bill of Materials (or BOM in short form).

Part ID	Amount
1789	3
5432	1
2345	10

When a transaction enters into an assembler, it is your responsibility to designate this transaction with a certain part ID chosen by you. This has to be done using the *Designate* method in the *OnEnter* event. If you didn't call *Designate* in the *OnEnter* event, the incoming transaction will be designated with 0.

After an assembler has all the necessary parts (according to the assembling scheme), it creates a new transaction which represents an assembly. The transactions designated with a certain part ID will be taken for the assembling in FIFO or LIFO order depending on the value of the *AssemblingMode* property. The *LIFO* and *FIFO* options are defined in the *AssemblingModes* enumeration.

When exactly does the actual assembling take place?

The key for answering this question is in the *MaxAssemblies* property. This property defines how many assemblies (exiting transactions) can the assembler block hold.

If the value of the property is positive, Delsi Runtime Engine will try to conduct the assembling any time an incoming transaction enters the block. If at this time all necessary parts are available, the assembling will take place and the resulting transaction will be ready to exit from the block. The only condition that can intervene the assembling at this moment is that the number of transactions (assemblies) which are ready for exiting has already reached the limit set in *MaxAssemblies*. For instance, if you set the *MaxAssemblies* property to 1, then the next assembling can take place only after the previous assembly exited from the block. If you wish the assembling capacity to be unlimited, set this property with the admittedly large number.

If the value of the property equals *0*, then the block can hold zero assemblies. It means that the assembling will take place right before an assembly is exiting from the block.

What to do with the transactions participated in the assembling as parts?

There are two options defined by the *BatchingMode* property:

- *Batched* - the transactions-parts will be packed (batched) into an outgoing transaction.
- *Unbatched* - the transactions-parts will be terminated. The outgoing transaction will not be a batch.

These options are defined in *BatchingModes* enumeration.

Syntax

```
public class TAssembler : Component, IBlock, ImodelItem
```

Methods

```
public void AddToScheme( int PartId, int Amount )
```

Adds a record to the assembling scheme of the storage. If the assembling scheme already contains a record with the specified part ID then Delsi Runtime Engine throws *DelsiException*.

```
public void ClearStatistics()
```

Clears simulation statistics accumulated in the block.

```
public long Count(int PartId)
```

Determines how many transactions designated with the specified part ID are waiting to be assembled.

```
public long CountExiting()
```

Returns the current number of assemblies (the transactions which are ready to exit from the block as a result of assembling).

public long **CountParts()**

Returns the current number of the transactions stored in the assembler as parts and waiting to be assembled.

```
public long Designate(int PartId)
```

Designates incoming transaction with a part ID. This method should be called in the *OnEnter* event of *TAssembler*. If you don't call the *Designate* method in the *OnEnter* event, the incoming transaction will be designated with 0.

```
public long Entries()
```

Returns the total number of transactions which have entered the block.

```
public long Exits()
```

Returns the total number of transactions which have exited from the block.

```
public bool isReadyToGet( Transaction transaction )
```

Returns *true* if the block is ready to receive a specific transaction. *TAssembler* is always ready to get a transaction.

```
public bool isReadyToSend()
```

Returns *true* if the block has a transaction to send.

```
public long RemoveFromScheme(int PartId)
```

Removes the record with the specified part ID from the assembling scheme. If the assembling scheme does not contain a record with the specified part ID then Delsi Runtime Engine throws *DelsiException*.

```
public string Report()
```

Returns the text containing the standard statistical report for the block. For *TAssembler* the standard report looks like this.

```
Entries: 543
Exits: 46
Count exiting: 1
Count (parts): 404
```

```
public bool Send( IBlock block )
```

Tries to send the transaction, which is ready to exit from the block, to another block. When using the *Send* method, you have to meet the following conditions:

- For *TAssembler*, this method can be called only on handling the *OnRouting* event.
- The block should have a transaction to send.
- The receiving block should belong to the same model as sending.
- The receiving block cannot be the same as sending.

If any of these conditions are not satisfied, Delsi Runtime Engine throws *DelsiException*.

```
public void UpdateScheme( int PartId, int Amount )
```

Updates the record with the specified part ID in the assembling scheme. If the assembling scheme does not contain a record with the specified part ID then Delsi Runtime Engine throws *DelsiException*.

Properties

```
public AssemblingModes AssemblingMode { get; set; }
```

Gets or sets the assembling mode, which can take the following values:

- *FIFO* - The transaction of a certain part Id will be retrieved for assembling in FIFO order.
- *LIFO* - The transaction of a certain part Id will be retrieved for assembling in LIFO order.

The default mode is *FIFO*.

```
public BatchingModes BatchingMode { get; set; }
```

Gets or sets the batching mode, which can take the following values:

- *Batched* - The transactions participated in the assembling will be packed (batched) into outgoing transaction.
- *NonBatched* - The transactions participated in the assembling will be terminated.

The default mode is *NonBatched*.

```
public int MaxAssemblies { get; set; }
```

Defines how many assemblies (exiting transactions) can the assembler block hold. This property answers the question: when does the actual assembling take place?

If the value of the property is positive, Delsi Runtime Engine will try to conduct the assembling any time an incoming transaction enters the block. If at this time all necessary parts are available, the assembling will take place and the resulting transaction will be ready to exit from the block. The only condition that can intervene the assembling at this moment is that the number of transactions (assemblies) which are ready for exiting has already reached the limit set in *MaxAssemblies*. For instance, if you set the *MaxAssemblies* property to 1, then the next assembling can take place only after the previous assembly exited from the block. If you wish the assembling capacity to be unlimited, set this property with the admittedly large number.

If the value of the property equals *0*, then the block can hold zero assemblies. It means that the assembling will take place right before an assembly is exiting from the block.

```
public string Caption { get; set;}
```

Gets or sets the text of the item caption.

```
public TModel Model{ get; }
```

Gets the model where the item belongs to.

```
public Object Tag { get; set; }
```

Gets or sets the object associated with the component.

Delegates

```
public delegate void EventHandler( TAssembler sender,
                                   Transaction transaction )
```

Represents the method which will handle the event fired by splitter *sender* for a specified *transaction*.

Events

```
public event TAssembler.EventHandler OnEnter
```

Occurs when a transaction enters the block.

```
public event TAssembler.EventHandler OnExit
```

Occurs when a transaction exits from the block..

```
public event TAssembler.EventHandler OnRouting
```

Fired when the block has a transaction ready to exit from the block. This is the place where the transaction will be sent to another block. (see method *Send*). You can apply any sophisticated logic to determine where to send the transaction. If the transaction is not sent, it will remain in the block until the next *OnRouting* event will be fired by Delsi Runtime Engine.

2.22. TStorage

This block stores transactions. In order to exit from the storage, the transactions should be first selected and then posted for exiting. Transactions can be selected from a storage in three ways:

- by transaction *Id*, using method *Select(int Id)*
- by transaction *Id* and *CopyId*, using method *Select(int Id, int CopyId)*
- by a criteria implemented in a custom method exposed via the *SelectionHandler* delegate, using method *Select(SelectionHandler Criteria, int Top)*

There are two scenarios to be considered for the transactions contained in a storage:

Scenario A:

Any transaction in the storage can be uniquely identified by its *Id*. In this scenario, in order to select some known transaction you should use method *Select(long Id)*. To make this method work in the most efficient way, set the storage property *SearchMode* equal to *ById*.

Scenario B:

You assume the possibility of having several transactions with the same *Id* in the storage. This can happen if your model contains a *TSplitter* block, which splits transaction into several clones with the same *Id* but different *CopyId*. In this scenario, in order to select some known transaction you should use method *Select(long Id, long Copy).* To make this method work in the most efficient way, set the storage property *SearchMode* equal to *ByIdAndCopyId* .

The default value for the *SearchMode* property is *ById*. Changing value of the *SearchMode* property is allowed only when the storage is empty.

Why do we have three overloads of the *Select* method?

The *Select(SelectionHandler Criteria, int Top)* method should be used when we we are intended to directly analyze transactions contained in a storage. In this case, transactions will be analyzed one by one with help of some custom method written by you. This custom method checks a transaction and returns *true* or *false.* You will specify this customer method via the *Criteria* parameter for the *Select* method.

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).*

What to do with selected transactions?

In order to select transactions you can make several calls of the *Select* methods. As a result, the selected transactions become "candidates" for exiting from the block.

If the results of the selections satisfy your logic, you can "post" (or forward) all selected transactions to exit from the storages using the *Post* method. When you post the selected transactions, you have the option to pack them into one batch-transaction by specifying parameter *Batched* in the call of the *Post* method.

If you want to roll-back all the selections, call the *Unselect* method. Here is the illustration.

```
GetProductsFromWarehouse()
{
        storage.Select(Criteria1);
        storage.Select(Critetia2);
        ...
        storage.Select(Id1);
        storage.Select(Id2);
        ...
        if (satisfied)
        {
            storage.Post(true);  // Post and pack into a batch
        }
        else
        {
            storage.Unselect();
        }
}
```

Syntax

```
public class TStorage : Component, IBlock, ImodelItem
```

Methods

```
public void ClearStatistics()
```

Clears simulation statistics accumulated in the block.

```
public long CountExiting()
```

Returns number of transactions ready to exit from the storage.

```
public long CountSelected()
```

Returns number of transactions selected by the *Select* methods.

```
public long CountStored()
```

Returns number of transactions stored in the storage. This includes the number of selected transactions.

```
public long Entries()
```

Returns the total number of transactions which have entered the block.

```
public long Exits()
```

Returns the total number of transactions which have exited from the block.

```
public bool isReadyToGet( Transaction transaction )
```

Returns *true* if the block is ready to receive a specific transaction. *TStorage* is always ready to get a transaction.

```
public bool isReadyToSend()
```

Returns *true* if the block has a transaction to send.

```
public void Post( bool Batched )
```

Posts (forwards) selected transactions to exit from the storage.

Parameters:

Batched - If *true*, all previously selected transactions will be forwarded to exit inside one batch transaction. If *false*, all previously selected transactions will be forwarded to exit one by one.

```
public string Report()
```

Returns the text containing the standard statistical report for the block. For *TAssembler* the standard report looks like this.

```
Entries: 872
Exits: 366
Count stored: 487
Count selected: 13
Count exiting: 6
```

```
public int Select( long Id )
```

Selects transactions by transaction *Id.* Returns a number of selected transactions.

```
public int Select( long Id, long CopyId )
```

Selects a transaction by transaction *Id* and *CopyId*. Returns 1 if the transaction is selected; 0 otherwise.

```
public int Select( SelectionHandler Criteria, int Top )
```

Selects transactions by the criteria defined by the custom method. Transactions are selected in no particular order. The method returns the number of selected transactions.

Parameters:

Criteria - Custom method determining the criteria for selection.
Top - Defines how many transactions conforming to the criteria will be selected. If *Top* equals 0, all transactions conforming to the criteria will be selected.

```
public bool Send( IBlock block )
```

Tries to send the transaction, which is ready to exit from the block, to another block. When using the *Send* method, you have to meet the following conditions:

- For *TAssembler*, this method can be called only on handling the *OnRouting* event.
- The block should have a transaction to send.
- The receiving block should belong to the same model as sending.
- The receiving block cannot be the same as sending.

If any of these conditions are not satisfied, Delsi Runtime Engine throws *DelsiException*.

```
public void Unselect()
```

Unselects all previously selected transactions.

Properties

```
public SearchModes SearchMode { get; set; }
```

Gets or sets the preferred searching mode, which can take the following values:

- *ById* – Optimizes the search by transaction *Id*; see method *Select(long Id)*;
- *ByIdandCopyId* - Optimizes the search by transaction *Id* and Copy*Id*; see method *Select(long Id, long CopyId)*;

If this property is set to the *ById* option, then only one transaction with certain *Id* can be in the storage at any given time. On attempt of a clone transaction to enter the storage Delsi Runtime Engine will throw *DelsiException*. The default mode is *ById*.

```
public string Caption { get; set;}
```

Gets or sets the text of the item caption.

```
public TModel Model{ get; }
```

Gets the model where the item belongs to.

```
public Object Tag { get; set; }
```

Gets or sets the object associated with the component.

Delegates

```
public delegate void EventHandler( TStorage sender,
                                   Transaction transaction )
```

Represents the method which will handle the event fired by splitter *sender* for a specified *transaction*.

Events

```
public event TStorage.EventHandler OnEnter
```

Occurs when a transaction enters the block.

```
public event TStorage.EventHandler OnExit
```

Occurs when a transaction exits from the block..

```
public event TStorage.EventHandler OnRouting
```

Fired when the block has a transaction ready to exit from the block. This is the place where the transaction will be sent to another block. (see method *Send*). You can apply any sophisticated logic to determine where to send the transaction. If the transaction is not sent, it will remain in the block until the next *OnRouting* event will be fired by Delsi Runtime Engine.

2.23. TMultiRand

This component generates random numbers with different continues distributions. The generation of numbers of different distributions is based on uniform distribution from 0 to 1. In order to get values of this distribution the component uses built-in prime modulus multiplicative congruent random number generator (*PMMCG*).

Syntax

```
public class TMultiRand : Component
```

Methods

```
public double Beta( double ShapeAlpha, double ShapeBeta, double Min, double Max )
```

Returns random number distributed with *Beta* distribution.

Parameters:

ShapeAlpha - Alpha-shape parameter of the Beta distribution
ShapeBeta - Beta-shape parameter of the Beta distribution
Min - Minimal value of the Beta distribution
Max - Maximum value of the Beta distribution

```
public double Erlang( double Mean, int Alpha )
```

Returns random number distributed with *Erlang* distribution.

Parameters:

Mean - mean value of the Erlang distribution
Alpha - shape parameter of the Erlang distribution

```
public double Exponential( double Mean )
```

Returns random number distributed with *Exponential* distribution.

Parameters:

Mean - mean value of the Exponential distribution

```
public double Gamma( double Mean, double Alpha )
```

Returns random number distributed with *Gamma* distribution.

Parameters:

Mean - mean value of the Gamma distribution
Alpha - parameter determining the shape of the Gamma distribution

```
public void InitializeCDF( double[] X, double[] P )
```

Initializes cumulative distribution function (*CDF*) for non-parametric random generation. Cumulative distribution function is also known as probability distribution function or just distribution function. It can be defined by two arrays of numbers. One array represents values of some random variable *X*. Another array defines cumulative probabilities *P* for those values. The cumulative probabilities are ranked from 0.0 to 1.0. Both arrays should be of the same size and ordered in ascending order.

Parameters:

X - array of values
P - array of cumulative probabilities

Example:

```
double[] X = { 4.0, 5.0, 6.0, 7.0, 8.0, 9.0, 10.0, 11.0};
double[] P = { 0.0, 0.23, 0.36, 0.52, 0.77, 0.84, 0.93, 1.0 };

tMultiRand1.InitializeCDF(X, P);
...
// The initialized CDF will be used for generating random numbers
// with non-parametric distribution.
double x = tMultiRand1.NonParametric();
```

```
public double Lognormal( double Mean, double Deviation )
```

Returns random number distributed with *Lognormal* distribution.

Parameters:

Mean - mean value of the Lognormal distribution
Deviation - standard deviation of the Lognormal distribution

```
public double NonParametric()
```

Returns random value according to cumulative distribution function (CDF) defined by method *InitializeCDF*. If cumulative distribution function is not define, the method will throw *DelsiException*.

```
public double Normal( double Mean, double Deviation )
```

Returns random number distributed with *Normal* distribution.

Parameters:

Mean - mean value of the Normal distribution
Deviation - standard deviation of the Normal distribution

```
public void Reset()
```

Resets seed of PMMCG random generator to initial value. The initial value of the seed is the default value assigned by the constructor, or the value previously assigned using the *Seed* property.

```
public double Triangular( double Min, double Max, double Mode )
```

Returns random number distributed with *Triangular* distribution.

Parameters:

Min - minimal value of the Triangular distribution
Max - maximum value of the Triangular distribution
Mode - mode of the Triangular distribution

```
public double Uni01()
```

Returns random number with uniform distribution from 0.0 to 1.0.

```
public double Uniform( double Min, double Max )
```

Returns random number uniformly distributed between *Min* and *Max*.

```
public double Weibull( double Alpha, double Scale )
```

Returns random number distributed with *Weibull* distribution.

Parameters:

Alpha - shape parameter of the Weibull distribution
Scale - scale parameter of the Weibull distribution

Properties

```
public uint Seed { get; set; }
```

Gets or sets the seed for *PMMCG* random generator.

2.24. TTabulator

This component gathers statistical data suitable for building a histogram. *TTabulator* collects number of hits of some random parameter into specified numeric intervals. The component also calculates the average value and the quadratic mean (estimation of a standard deviation) of the tabulated parameter.

Syntax

```
public class TTabulator : Component
```

Methods

```
public double Average()
```

Returns the average (arithmetic mean) of tabulated values.

```
public long Count()
```

Returns total number of hits.

```
public double Deviation()
```

Returns estimated standard deviation (quadratic mean) of tabulated values.

```
public double getHits( int Index )
```

Returns a number of hits for the specified interval. If *Index* is equal to 0, the method returns the number of the hits to the section lower than the first interval. If Index is equal to *IntervalCount+1* then the method returns the number of hits to the section higher than the last interval.

```
public void Reset()
```

Resets all accumulated data in the tabulator.

```
public double MaxValue()
```

Return maximum value.

```
public double MinValue()
```

Return minimal value.

```
public void Tabulate( double Value )
```

Tabulates the value, which means it determines the numeric interval which fits the value and increments the number of hits for this interval.

Properties

```
public double Interval { get; set; }
```

Width of the numeric interval.

```
public int IntervalCount { get; set; }
```

Number of intervals (can range from 1 to 254).

```
public double LowerBound { get; set; }
```

Lower bound (beginning) of the first interval.