November 23, 2011

OPC UA Client and Server Development Tutorial

Tetris Server and Client with the OPC Foundation C#.NET SDK

Artesis | WebCom

Institute:	Artesis University College of Antwerp	website: www.artesis.be
Project:	WebCom	website: www.webcom-eu.org
Project Coordinator:	Dirk van der Linden	e-mail: dirk.vanderlinden@artesis.be
Author:	Vincent Vanderputten	e-mail: vincent.vanderputten@artesis.be
Version | Date:	v2.6.0 | November 23, 2011	

Table of Contents

0 Introduction

This tutorial was made at Artesis University College as part of the WebCom project (www.webcom-eu.org). Its purpose is to introduce the OPC UA .NET SDK developed by the OPC Foundation (www.opcfoundation.com) and learn how to work with it. The reader is assumed to have at least *some* knowledge of:

- The basic OPC UA concepts:
  - information modeling with nodes, references and attributes
  - data transport over secure channel, session, subscription by means of monitored items and notifications
  - the most important services
  - …
- The .NET language C# and Visual Studio

For literature on OPC UA, I will refer to the following recourses:

- The Unified Architecture section of the OPC Foundation website: www.opcfoundation.com/ua
- The book "OPC Unified Architecture" by Wolfgang Mahnke, Stefan-Helmut Leitner and Matthias Damm *(a pdf version of this book is available for the project partners on our website www.webcom-eu.org)*
- The book "OPC - From Data Access to Unified Architecture" by Jürgen Lange, Frank Iwanitz, Thomas J. Burke

The idea in this tutorial is to start from a simple UA client and a UA server program (in one Visual Studio 2008 Solution) and step by step implement more functionality. Once you get to a certain step, you can either choose to continue working towards the next step with your own solution, or use the solution for the corresponding step provided in this download. You could choose the latter option in case your solution is not working or you just want to skip a certain step.

Before you start, you need to have Visual Studio 2008 installed. You should also install:

- OPC UA SDK 1.01 **DotNet SDK**
- OPC UA SDK 1.01 **Quickstarts**
- OPC UA SDK 1.01 Test Applications
  (This is optional, but the Sample Client has a lot and good generic capabilities, useful for debugging etc. Another option is Unified Automation's UAExpert found at http://www.unified-automation.com/opc-ua-clients, similar in concept to Matrikon's MatrikonOPC Explorer for the Classic OPC specs)

Although these tutorial projects were made with an older version of the SDK, they do run with **SDK version 1.01.329**. All this is available for download on the WebCom website (www.webcom-eu.org) for WebCom partners or on the OPC Foundation website (should you be a corporate member of the OPC Foundation).

Once you have installed all the software, *copy/paste* the different tutorial folders, corresponding to the different steps of the tutorial, in the folder where you also find the other installed quickstarts:
C:\Users\*user*\AppData\Local\OPC Foundation\UA SDK\v1.1\Quickstarts\Source\Workshop
(in my case on a windows 7 machine)

Open the folder "Tetris 0" and open the solution: "Quickstart Tetris Solution.sln". This will be our starting point, our starting client and server to which we will add more features. In the Solution Explorer on the right you will see:

- The **SDK** with four projects:
  - Core Library (stack)
  - Configuration Library
  - Client SDK Library
  - Server SDK Library

These are the tools with which one builds UA clients and UA servers (in .NET).
- The **Quickstart Library**:
  a layer on top of the SDK with some helpful functionality, a standard server form, etc. used in clietns and servers.
- The **Tetris Client**
- The **Tetris Server**

We will only change and add code to the last two projects.

There is a possible problem on running the Tetris Server program initially. After you have built it once, you need to run the built .exe once from the command line:
- Go to: All Programs => Accessories => **Command prompt**: right-click and **run as administrator**
- Browse to folder:
  C:\Users\*user*\AppData\Local\OPC Foundation\UA SDK\v1.1\Quickstarts\Source\Workshop\Tetris\Server\bin\Debug
- Run the .exe with "install" option: **quickstarts.tetrisserver.exe /install**

Now you should be able to build and run without errors.

The initial program, "Tetris 0", will contain a simple UA server, that has an Address Space containing just two hardcoded test nodes, and a UA client with simple browse, monitor, etc. functionality. Try running both and connect with the client to the server: copy the endpoint URL from the server window into the client window and click 'connect'. You will now be able to browse the server Address Space.

Get to the test property ("test_PropertyNode" is a component of the "test_ObjectNode"), right click it and select Monitor. In the bottom window the monitored item is visible and we can see the value of the property: five integer values in an int array. It is static since the values are hard coded and no actual or simulated process is updating these values.

When a node is selected in the browse window, the right window shows the attributes belonging to that node as well as the values and data types of these attributes. We can see that the "test_PropertyState" property has a NodeClass of '2', indicating it is a Variable Node. There are eight different but fixed NodeClasses: see Figure 1 below.

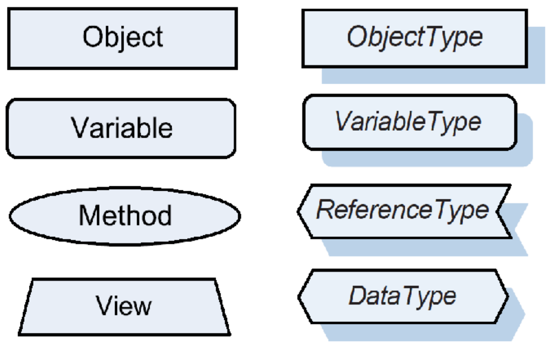
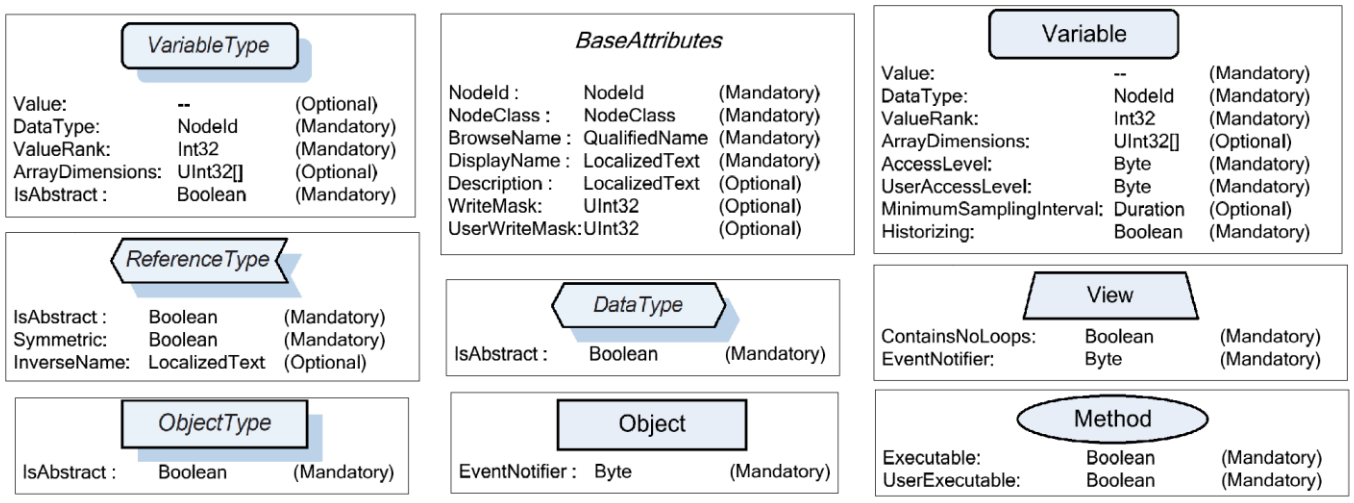


Figure 1:
The eight different NodeClasses.

The *DataType attribute* is a NodeId (this node's data type is a NodeId: see Figure 2), pointing to a node in the DataType Hierarchy (see Figure 6 below: in this case it is the NodeId of the Int32 DataType node). The *ValueRank attribute* is '1', meaning the value is one dimensional, i.e. an array. So we have a Property node that has an array of Int32 integers as its value. A property is always a VariableNode, see Figure 4 (PropertyState inherits from and is a subtype of BaseVariableState). It is a special kind of VariableNode that is only child of one other node and thus tightly coupled to it. When monitoring the node's *Value attribute* by creating a monitored item, as we did above, we can see that this array is {7,-18,56,0,3}. All these different attributes are fixed for each one of the eight different nodes classes, although some attributes, the BaseAttributes, are common among all nodes: see Figure 2 below.



VariableType

Value:	--	(Optional)
DataType:	NodeId	(Mandatory)
ValueRank:	Int32	(Mandatory)
ArrayDimensions:	UInt32[]	(Optional)
IsAbstract :	Boolean	(Mandatory)

BaseAttributes

NodeId :	NodeId	(Mandatory)
NodeClass :	NodeClass	(Mandatory)
BrowseName :	QualifiedName	(Mandatory)
DisplayName :	LocalizedText	(Mandatory)
Description :	LocalizedText	(Optional)
WriteMask:	UInt32	(Optional)
UserWriteMask:	UInt32	(Optional)

Variable

Value:	--	(Mandatory)
DataType:	NodeId	(Mandatory)
ValueRank:	Int32	(Mandatory)
ArrayDimensions:	UInt32[]	(Optional)
AccessLevel:	Byte	(Mandatory)
UserAccessLevel:	Byte	(Mandatory)
MinimumSamplingInterval:	Duration	(Optional)
Historizing:	Boolean	(Mandatory)

ReferenceType

IsAbstract :	Boolean	(Mandatory)
Symmetric:	Boolean	(Mandatory)
InverseName:	LocalizedText	(Optional)

DataType

IsAbstract :	Boolean	(Mandatory)

View

ContainsNoLoops:	Boolean	(Mandatory)
EventNotifier:	Byte	(Mandatory)

ObjectType

IsAbstract :	Boolean	(Mandatory)

Object

EventNotifier :	Byte	(Mandatory)

Method

Executable:	Boolean	(Mandatory)
UserExecutable:	Boolean	(Mandatory)

Figure 2:
All the common and different Attributes of the eigth NodeClasses with their data types.

Let us implement some extra features, step-by-step, to make the server and client more interesting. If at any point you encounter any issues, problems, questions, doubts, etc. please feel free to e-mail me at vincent.vanderputten@artesis.be. Since I am not a professional programmer, I do not claim the following code to be perfect or even to be considered decent programming by any standard. But I do think it serves well for illustrative purposes.

In the download of this package, there should also be a feedback form: please fill this in and send it back to the same e-mail address. This would be *greatly* appreciated!

1 TetrisSimulation on Server

In this step, only changes to the Server side are made. Open the “Tetris 0” solution, like you already did during the introduction. This time we will make some additions. More specific: we will add nodes to our Address Space that represent values from a simulated Tetris game.

We will start by creating the nodes that will represent our simulated Tetris data. We will hard code the nodes as predefined nodes, in contrast to an approach where nodes are created dynamically, as is the case in the “DataAccess Quickstart” where nodes are only created (based on data from an Underlying System) whenever a client browses the node, needs data from it, etc. I think this better illustrates some basic UA concepts like different nodes, attributes, references, etc. We will use a TetrisSimulation class already implemented to serve as Underlying System and provide data for our nodes. Then the Underlying System and our nodes will be linked by periodically updating the node’s value attribute with data from our TetrisSimulation class by means of a Timer. But first, implement a few nodes.

Open the *TetrisNodeManager.cs* file. This contains the TetrisNodeManager class. The TetrisNodeManager, or more general a ‘CustomNodeManager’, manages the part of the Address Space made and controlled by you. Below is a figure, see Figure 3, of part of the architecture of a UA server built with the server SDK. The MasterNodeManager manages all the other NodeManagers, including our custom NodeManager: TetrisNodeManager in our case, DataAccessNodeManager in the case of Figure 3. One server could contain multiple CustomNodeManagers, each with their own Address Space. The MasterNodeManager also manages the DiagnosticsNodeManager, which manages the default server diagnostic nodes representing the server status etc., as well as the CoreNodeManager, which manages other standard nodes like the Root, Objects, Types and Views Nodes.

The Underlying System is not actually part of the UA SDK and is a general name for whatever infrastructure, functionality, classes, code, etc. you create that provides data to the Address Space. Your NodeManager and your Underlying System are the two elements that you, as developer of a server, will have to make and/or modify.

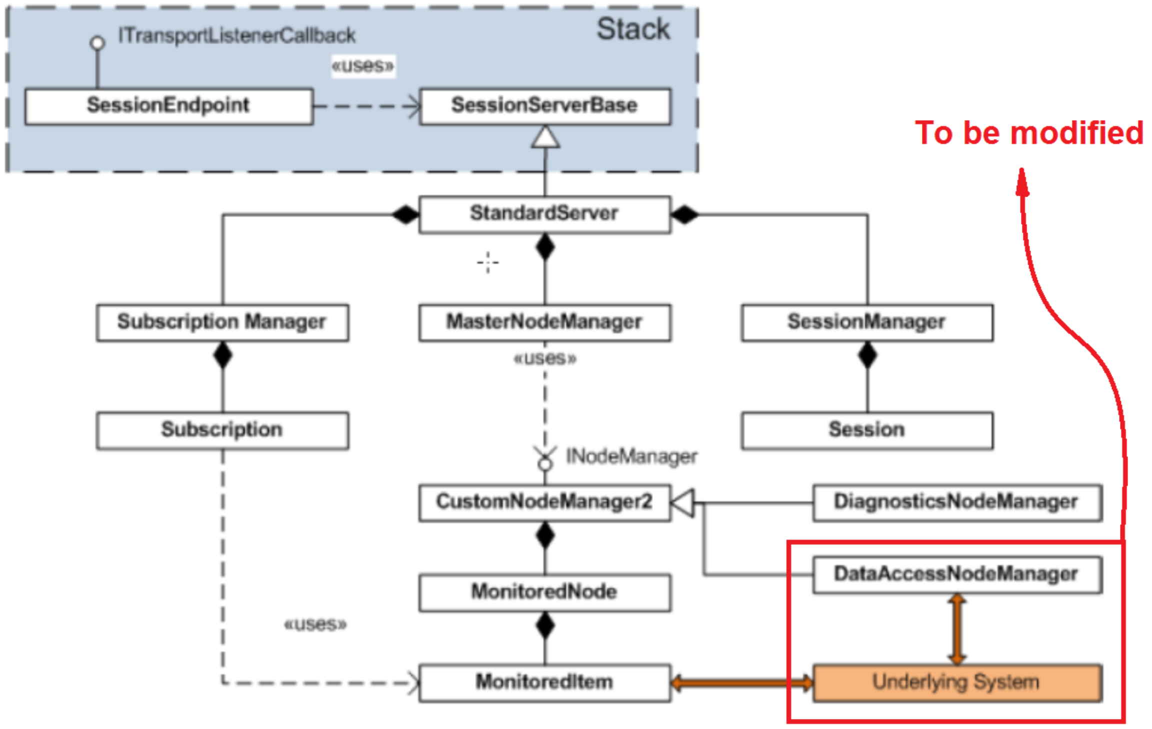


Figure 3:
The OPC Foundation .NET Server SDK architecture.

In the TetrisNodeManager class, you will find the "CreateAddressSpace(…)" method in the "INodeManager Members" region. Here the initialization of the Address Space is done. There is already some code in here, namely the code for the two nodes we encountered when running the initial client and server during the introduction. You can try and figure out what is done there. If it is not clear, don't worry, the next part will deal with adding the Tetris nodes in the same way as the two already coded ones.

Note that in the first parts of this tutorial, when we refer to "methods", we are referring to C#.NET methods, i.e. functions, not to "UA methods": OPC UA also has the concept of methods and Method nodes. Later on we will implement actual "UA methods".

First we will add an object node, one under which the nodes containing data will be organized. This can be done by adding the following code to the "CreateAddressSpace(…)" method:

```
//------------------------------ The nodes for the tetris simulation --------------------------

// The object node under wich the other nodes containing the data will be organized:
BaseObjectState tetrisSimulationNode = new BaseObjectState(null);
tetrisSimulationNode.NodeId = new NodeId(id++, NamespaceIndex);
tetrisSimulationNode.BrowseName = new QualifiedName("Tetris_Simulation", NamespaceIndex);
tetrisSimulationNode.DisplayName = tetrisSimulationNode.BrowseName.Name;
tetrisSimulationNode.SymbolicName = tetrisSimulationNode.BrowseName.Name;
tetrisSimulationNode.TypeDefinitionId = ObjectTypeIds.BaseObjectType;

// ensure tetrisSimulationNode can be found via the server object: add references to/from it!
references = null;
if (!externalReferences.TryGetValue(ObjectIds.ObjectsFolder, out references))
{
       externalReferences[ObjectIds.ObjectsFolder] = references = new List<IReference>();
}
// Forward Reference FROM the ObjectsFolder node TO our tetris node:
references.Add(new NodeStateReference(ReferenceTypeIds.Organizes, false, tetrisSimulationNode.NodeId));

// Inverse Reference FROM our tetris node TO the ObjectsFolder node:
tetrisSimulationNode.AddReference(ReferenceTypeIds.Organizes, true, ObjectIds.ObjectsFolder);

// FOLLOWING CODE SHOULD COME HERE
// ...

AddPredefinedNode(SystemContext, tetrisSimulationNode);
//-----------------------------------------------------------------------------------------------
```

In the first line a new BaseObjectState is instantiated, which is the in code representation of an Object node. The hierarchy of C# classes representing nodes is shown in Figure 4 (the BaseObjectState used in our code corresponds to the ObjectState in Figure 4).

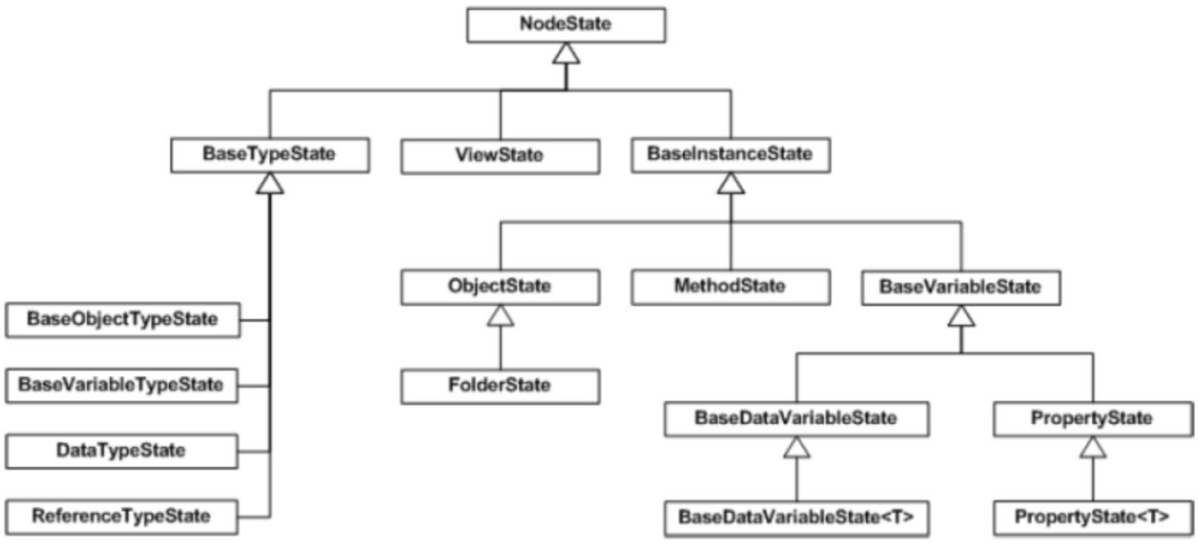


Figure 4:
The hierarchy of classes in the C#.NET SDK representing UA nodes in code. This is *not* the UA Type Hierarchy.

Its attributes are set in the following lines. A NodeId consists of an identifier (GUID, opaque, string, or in our case numeric) and a namespaceindex. The namespaceindex is a short integer and is the index in the server namespacetable at which the actual namespace can be found. Namespaceindexes are used instead of full namespace strings to reduce overhead in communication. Our TetrisNodeManager Address Space has its own namespace and corresponding namespaceindex. The TypeDefinitionId and SymbolicName are not actually UA attributes of a node (see Figure 2). The TypeDefinitionId is a NodeId that refers to a certain Type node in the UA Type Hierarchy, which in UA terms would translate as a "HasTypeDefinition" reference from our Object Node to an ObjectType node, the BaseObjectType node, in the UA Type Hierarchy. Note that this UA Hierarchy does not correspond to the hierarchy of classes representing nodes in Figure 4.

Our nodes should also be found by a client. Every client enters a server's Address Space through the Root Object. This Root Object itself has three Object nodes (FolderType nodes, see Figure 4) organized under it: Objects, Views and Types. These four nodes are four standard nodes managed by the CoreNodeManager. The whole Type Hierarchy is organized under the Types Object node, so also the DataTypes depicted in Figure 6 and the EventTypes from Figure 9 below are part of this big UA Type Hierarchy. Our nodes have to be organized under the Objects Object node if we want clients to be able to browse them. How to achieve this?

As an argument of the "CreateAddressSpace(…)" method, *externalReferences* is passed, which is a dictionary that contains lists of references that belong to well-known Nodes, like the Objects Object node. In the code above we first fetch the list that belongs to the Objects Object node (the ObjectsFolder), which is done in the 'if'-statement. Then an "Organizes" reference is added to this list with the "Add(…)" method called on this list, in which we pass the kind of reference we want and the target NodeId as arguments: an "Organizes" reference and our tetrisSimulationNode's NodeId respectively. From now on there is an "Organizes" reference from the ObjectsFolder (the Objects Object node) to our tetrisSimulationNode, but the inverse direction is not known anywhere yet: browsing from the ObjectsFolder to our tetrisSimulationNode will work, but browsing from our tetrisSimulationNode to the ObjectsFolder following the inverse "Organizes" reference will not work. So in the next line we do exactly that: add the inverse "Organizes" reference from our node to the ObjectsFolder.

The last line of code above adds the tetrisSimulationNode node to our NodeManagers list of Predefined Nodes. This works recursively: in the next part we will add Variable nodes and add them to the Address Space by organizing them under our tetrisSimulationNode with "HasComponent" references, but we just have to call the "AddPredefinedNode(…)" method once on the parent node, since the method is called recursively on the node's children as well.

Now run the server and connect with the client: in the browse window you should be able to see our Tetris_Simulation node. Now let us add three Variable nodes that will later contain our simulated data, so that our nodes are organized like depicted in Figure 5 below.

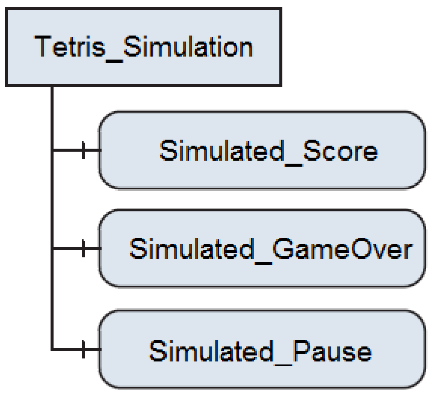


Figure 5:
The Tetris_Simulation Object node with its child Variable nodes that contain the actual data, related to a simulated Tetris game in this case.

This is achieved by adding the following code, before the "AddPredefinedNode(…)" method, at the line of comment "`// FOLLOWING CODE SHOULD COME HERE`" :

```
// -------------------- Nodes containing data -------------------
// The nodes containing some simulated tetris data:

//Score:
BaseDataVariableState simulatedScoreNode = new BaseDataVariableState(tetrisSimulationNode);
simulatedScoreNode.NodeId = new NodeId(id++, NamespaceIndex);
simulatedScoreNode.BrowseName = new QualifiedName("simulated_Score", NamespaceIndex);
simulatedScoreNode.DisplayName = simulatedScoreNode.BrowseName.Name;
simulatedScoreNode.TypeDefinitionId = VariableTypeIds.BaseDataVariableType;
simulatedScoreNode.ReferenceTypeId = ReferenceTypeIds.HasComponent;
simulatedScoreNode.DataType = DataTypeIds.Int32;
simulatedScoreNode.ValueRank = ValueRanks.Scalar;
//add the reference from the Tetris_Simulation Object Node to this one:
tetrisSimulationNode.AddChild(simulatedScoreNode);

//Game Over status:
BaseDataVariableState simulatedGameOverNode = new BaseDataVariableState(tetrisSimulationNode);
simulatedGameOverNode.NodeId = new NodeId(id++, NamespaceIndex);
simulatedGameOverNode.BrowseName = new QualifiedName("simulated_GameOver", NamespaceIndex);
simulatedGameOverNode.DisplayName = simulatedGameOverNode.BrowseName.Name;
simulatedGameOverNode.TypeDefinitionId = VariableTypeIds.BaseDataVariableType;
simulatedGameOverNode.ReferenceTypeId = ReferenceTypeIds.HasComponent;
simulatedGameOverNode.DataType = DataTypeIds.Boolean;
simulatedGameOverNode.ValueRank = ValueRanks.Scalar;
//add the reference from the Tetris_Simulation Object Node to this one:
tetrisSimulationNode.AddChild(simulatedGameOverNode);

//Paused status:
BaseDataVariableState simulatedPausedNode = new BaseDataVariableState(tetrisSimulationNode);
simulatedPausedNode.NodeId = new NodeId(id++, NamespaceIndex);
simulatedPausedNode.BrowseName = new QualifiedName("simulated_Paused", NamespaceIndex);
simulatedPausedNode.DisplayName = simulatedPausedNode.BrowseName.Name;
simulatedPausedNode.TypeDefinitionId = VariableTypeIds.BaseDataVariableType;
simulatedPausedNode.ReferenceTypeId = ReferenceTypeIds.HasComponent;
simulatedPausedNode.DataType = DataTypeIds.Boolean;
simulatedPausedNode.ValueRank = ValueRanks.Scalar;
simulatedPausedNode.AccessLevel = AccessLevels.CurrentReadOrWrite;
simulatedPausedNode.Value = false;
//add the reference from the Tetris_Simulation Object Node to this one:
tetrisSimulationNode.AddChild(simulatedPausedNode);
//--------------------------------------------------------------
```

Much the same is done here. A new node is initialized, but now a Variable node instead of an Object node (a BaseDataVariableState instead of a BaseObjectState). As an argument we can pass the parent node. Since the tetrisSimulationNode is the one we want to add this new node to as a component (see Figure 5), we indeed pass it as the constructor argument. NodeId, BrowseName and Displayname attributes are set in the same manner as was previously done. The TypeDefinition is now another NodeId: now it points to the BaseDataVariableType node in the TypeHierarchy. The ReferenceTypeId is set to the kind of reference we want between this node and its parent node (the tetrisSimulationNode), the "HasComponent" reference in this case. The DataType is again a NodeId, a NodeId that refers to a DataType node in the DataType Hierarchy (see Figure 6). The ValueRank attribute determines whether the Value attribute is a Scalar, one dimensional, two dimensional, etc.
For the simulatedPausedNode a default value is set, as well as the Acceslevel. The Acceslevel is set to 'Read' *and* 'Write', so a client can also write the Value attribute, the 'Paused' Boolean, and thus pause the simulated Tetris game.

We passed the parent node as an argument in the constructor and set the kind of reference we want between the two through the ReferenceTypeId setting. So the inverse reference, from child to parent, is known, but not the forward reference from parent to child: *tetrisSimulationNode "HasComponent" simulatedScoreNode*. This is handled by calling the "AddChild(…)" method on the parent node and pass the child node as argument (the last line of code for each of the three blocks).

Now compile, run and again connect with the client to the server: the three new nodes can be found as components of the Tetris_Simulation node.

Now let us actually simulate some values and put them into the variable node's value attribute. In the region "Private Fields" you will find already some instances defined:

```
//------------------- The underlying system: Tetris --------------------
TetrisServerControlsForm m_TetrisControlsForm;
TetrisGame m_TetrisGame;
TetrisSimulation m_TetrisSimulation;
```

We will use "m_TetrisSimulation" now, the others are for later steps. Initialization takes place in the constructor by the already coded line: `m_TetrisSimulation = new TetrisSimulation();`

The class will generate a few simulated values in the background. We will use these as Underlying System and feed this data to our nodes. To do this we will make a timer and periodically pull the data from the TetrisSimulation class and push it into the nodes' value attribute. So in the "Private Fields" region, add a Timer:

```
Timer m_UpdateTimer;
```

From the moment the nodes are initialized, the Timer should be initialized. This means that at the end of the "CreateAddressSpace(…)" method you should code something like this:

```
//------------------------ Initialize UpdateTimer---------------------------
m_UpdateTimer = new Timer(UpdateTimerCallback, null, 1000, 500);
//------------------------------------------------------------------------
```

And we implement the timer callback 'UpdateTimerCallback':

```
#region UpdateTimer
private void UpdateTimerCallback(object state)
{
        #region Update Tetris Simulation Nodes
        foreach (NodeState nodestate in PredefinedNodes.Values)
        {
                if (nodestate.SymbolicName == "Tetris_Simulation")
                {
                        //When the Tetris_Simulation Node is found, get its children:
                        List<BaseInstanceState> children = new List<BaseInstanceState>();
                        nodestate.GetChildren(SystemContext, children);
                        //Iterate through the children and update the necessary nodes:
                        foreach (BaseInstanceState child in children)
                        {
                                BaseDataVariableState childvariable = child as BaseDataVariableState;
                                if (childvariable != null)
                                {
                                        switch (childvariable.BrowseName.Name)
                                        {
                                                case "simulated_Score":
                                                        childvariable.Value = m_TetrisSimulation.Score;
                                                        break;
                                                case "simulated_GameOver":
                                                        childvariable.Value = m_TetrisSimulation.GameOver;
                                                        break;
                                                case "simulated_Paused":
                                                        if (childvariable.Value != null)
                                                        {
                                                        m_TetrisSimulation.Paused = (bool)childvariable.Value;
                                                        }
                                                        break;
                                                default:
                                                        break;
                                        }
                                }
                        }
                        // make known that this nodes and its child nodes have changed:
                        nodestate.ClearChangeMasks(SystemContext, true);
                        // Tetris node found, no further need to iterate:
                        break;
                }
        }
        #endregion
}
#endregion
```

Let us see what we do here. We first go through all the predefined nodes and look for the Tetris_Simulation node, since we know our data nodes are organized hierarchically under it. Once we have that node, we go through all its children, check their browse paths and update the necessary data variable nodes: some data ("score" and "game over" status) goes from the Underlying System (the Tetris simulation) to the nodes in the Address Space, other data ("paused") flows from the node to Underlying System. This way a client has the capability to pause the game by setting the "paused" node to "true". Of course a more "UA-like approach" would be to add a Method node that can be called to pause the Tetris game, but UA methods are handled in a later step, "step 4".

Note that a client cannot write to the "score" or "game over" nodes, since we did not set their AccessLevel attribute to "write"!

Also note that after all nodes have been updated, the "ClearChangeMasks(…)" method is called on the parent node. This notifies any changes to the SDK, so that notifications with the new values are sent to the client.

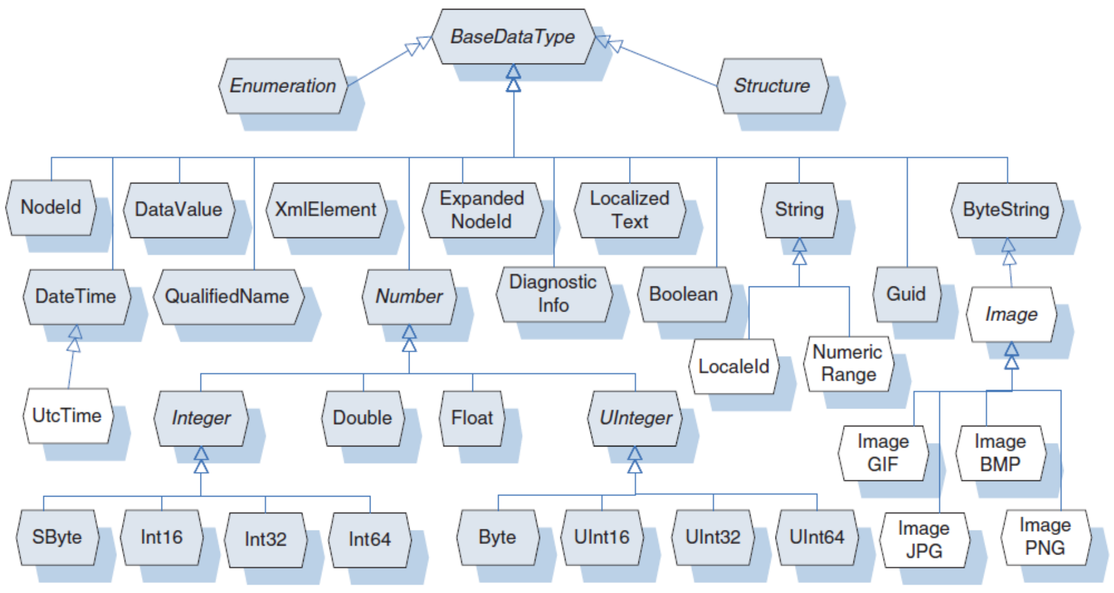


Figure 6:
The UA DataType Hierarchy.

A recap of what was done:

- Create a few nodes and connect them with references
- Create a simulated Tetris game that serves as Underlying System
- Implement a Timer to periodically update the nodes and link the Underlying System with the Address Space (this approach is only one of many, there are other, mostly more preferable, ways to link your Underlying System with the UA Address Space)

This is all we intended to do in this step, so try to compile, run and connect with the client to the server. Right-click the three data nodes in the browse menu and monitor all three: the score will go up every few seconds (might take more ten a few seconds!) and every random interval the game will get into Game Over status, the score will be reset as well as the "game". You can right click the "Paused" node or "Paused" monitored item en write "true" or "false": try setting it to "true" and the game will be paused indefinitely, until it is set to "false" again.

In the next step we will have an actual Tetris game running on the server side.

2 TetrisGame on Server

The classes to run and play a Tetris game are already implemented. We will do more or less the same as in the first step, but provide the nodes with data from an actual Tetris game.

Go to the *TetrisNodeManager.cs* file (and class) again. In the constructor you will see the code:

```
//----------------- Initialize the Tetris Functionality --------------------
m_TetrisSimulation = new TetrisSimulation();
m_TetrisGame = new TetrisGame();
m_TetrisControlsForm = new TetrisServerControlsForm(m_TetrisGame);
//m_TetrisControlsForm.Show();
//--------------------------------------------------------------------------
```

Now uncomment the last line of code:

```
m_TetrisControlsForm.Show();
```

When running the server, you will notice a GUI for the Tetris game popping up. You can play (use the four buttons on the form or numpad's '1', '2', '3' and 'space' to move the block left, down, right and rotate respectively), pause or pause for a number of second and reset the game.

We will repeat the same procedure as above to link nodes to the Tetris game data.

There will be a lot of hard coded nodes in the "CreateAddressSpace(…)" method, so it might make things more clear if you put the code in different regions which are collapsible.

The new nodes for the Tetris game, to be added in the "CreataAddressSpace(…)" method:

```
#region teris game nodes
//------------------------------ The nodes for the tetris game --------------------------------

// The object node under wich the other nodes containing the data will be organized:
BaseObjectState tetrisGameNode = new BaseObjectState(null);
tetrisGameNode.NodeId = new NodeId(id++, NamespaceIndex);
tetrisGameNode.BrowseName = new QualifiedName("Tetris_Game", NamespaceIndex);
tetrisGameNode.DisplayName = tetrisGameNode.BrowseName.Name;
tetrisGameNode.SymbolicName = tetrisGameNode.BrowseName.Name;
tetrisGameNode.TypeDefinitionId = ObjectTypeIds.BaseObjectType;

// ensure tetrisGameNode can be found via the server object: add references to/from it!
references = null;
if (!externalReferences.TryGetValue(ObjectIds.ObjectsFolder, out references))
{
       externalReferences[ObjectIds.ObjectsFolder] = references = new List<IReference>();
}
// Forward Reference FROM the ObjectsFolder node TO our tetris node:
references.Add(new NodeStateReference(ReferenceTypeIds.Organizes, false, tetrisGameNode.NodeId));

// Inverse Reference FROM our tetris node TO the ObjectsFolder node:
tetrisGameNode.AddReference(ReferenceTypeIds.Organizes, true, ObjectIds.ObjectsFolder);

// -------------------- Nodes containing data --------------------
// The nodes containing some simulated tetris data:

//Score:
BaseDataVariableState scoreNode = new BaseDataVariableState(tetrisGameNode);
scoreNode.NodeId = new NodeId(id++, NamespaceIndex);
scoreNode.BrowseName = new QualifiedName("Score", NamespaceIndex);
scoreNode.DisplayName = scoreNode.BrowseName.Name;
scoreNode.TypeDefinitionId = VariableTypeIds.BaseDataVariableType;
scoreNode.ReferenceTypeId = ReferenceTypeIds.HasComponent;
scoreNode.DataType = DataTypeIds.Int32;
scoreNode.ValueRank = ValueRanks.Scalar;
//add the reference from the Tetris_Game Object Node to this one:
tetrisGameNode.AddChild(scoreNode);
```

```
        //Game Over status:
        BaseDataVariableState gameOverNode = new BaseDataVariableState(tetrisGameNode);
        gameOverNode.NodeId = new NodeId(id++, NamespaceIndex);
        gameOverNode.BrowseName = new QualifiedName("GameOver", NamespaceIndex);
        gameOverNode.DisplayName = gameOverNode.BrowseName.Name;
        gameOverNode.TypeDefinitionId = VariableTypeIds.BaseDataVariableType;
        gameOverNode.ReferenceTypeId = ReferenceTypeIds.HasComponent;
        gameOverNode.DataType = DataTypeIds.Boolean;
        gameOverNode.ValueRank = ValueRanks.Scalar;
        //add the reference from the Tetris_Game Object Node to this one:
        tetrisGameNode.AddChild(gameOverNode);

        //Paused status:
        BaseDataVariableState pausedNode = new BaseDataVariableState(tetrisGameNode);
        pausedNode.NodeId = new NodeId(id++, NamespaceIndex);
        pausedNode.BrowseName = new QualifiedName("Paused", NamespaceIndex);
        pausedNode.DisplayName = pausedNode.BrowseName.Name;
        pausedNode.TypeDefinitionId = VariableTypeIds.BaseDataVariableType;
        pausedNode.ReferenceTypeId = ReferenceTypeIds.HasComponent;
        pausedNode.DataType = DataTypeIds.Boolean;
        pausedNode.ValueRank = ValueRanks.Scalar;
        //add the reference from the Tetris_Game Object Node to this one:
        tetrisGameNode.AddChild(pausedNode);

        //SecondsTillUnpause:
        BaseDataVariableState secondsTillUnpauseNode = new BaseDataVariableState(tetrisGameNode);
        secondsTillUnpauseNode.NodeId = new NodeId(id++, NamespaceIndex);
        secondsTillUnpauseNode.BrowseName = new QualifiedName("Seconds_Till_Unpause", NamespaceIndex);
        secondsTillUnpauseNode.DisplayName = secondsTillUnpauseNode.BrowseName.Name;
        secondsTillUnpauseNode.TypeDefinitionId = VariableTypeIds.BaseDataVariableType;
        secondsTillUnpauseNode.ReferenceTypeId = ReferenceTypeIds.HasComponent;
        secondsTillUnpauseNode.DataType = DataTypeIds.Int32;
        secondsTillUnpauseNode.ValueRank = ValueRanks.Scalar;
        //add the reference from the Tetris_Game Object Node to this one:
        tetrisGameNode.AddChild(secondsTillUnpauseNode);

        //SecondsTillUnpause:
        BaseDataVariableState fieldNode = new BaseDataVariableState(tetrisGameNode);
        fieldNode.NodeId = new NodeId(id++, NamespaceIndex);
        fieldNode.BrowseName = new QualifiedName("Field", NamespaceIndex);
        fieldNode.DisplayName = fieldNode.BrowseName.Name;
        fieldNode.TypeDefinitionId = VariableTypeIds.BaseDataVariableType;
        fieldNode.ReferenceTypeId = ReferenceTypeIds.HasComponent;
        fieldNode.DataType = DataTypeIds.Int32;
        fieldNode.ValueRank = ValueRanks.TwoDimensions;
        //fieldNode.ArrayDimensions = ???
        // => get the matrix (twodimensional array) dimension lengths:
        uint x = (uint)m_TetrisGame.DynamicField.GetLength(0);
        uint y = (uint)m_TetrisGame.DynamicField.GetLength(1);
        fieldNode.ArrayDimensions = new ReadOnlyList<uint>(new uint[] { x, y });
        //add the reference from the Tetris_Game Object Node to this one:
        tetrisGameNode.AddChild(fieldNode);
        //----------------------------------------------------------------

        AddPredefinedNode(SystemContext, tetrisGameNode);
        //----------------------------------------------------------------------------------------------
        #endregion
```

This is the same approach as for the simulation nodes: make a parent node (tetrisGameNode) and add child nodes that will contain the actual Tetris Game (same approach as in Figure 5). The fieldNode will contain a matrix that holds the tetris field data. To correctly fill in the “ArrayDimensions” attribute, we check the dimensions of the Tetris field of our Tetris game:

*(***repeated code!****)*

```
        //fieldNode.ArrayDimensions = ???
        // => get the matrix (twodimensional array) dimension lengths:
        uint x = (uint)m_TetrisGame.DynamicField.GetLength(0);
        uint y = (uint)m_TetrisGame.DynamicField.GetLength(1);
        fieldNode.ArrayDimensions = new ReadOnlyList<uint>(new uint[] { x, y });
```

*(***repeated code!****)*

At this point the nodes are visible in the Address Space, which you can check when you connect with a client, but no data changes take place yet. To Achieve this, add again a “foreach” loop to the “UpdatTimerCallback(…)” method:

```
#region Update Tetris Games Nodes
foreach (NodeState nodestate in PredefinedNodes.Values)
{
    if (nodestate.SymbolicName == "Tetris_Game")
    {
        //When the Tetris_Game Node is found, get its children:
        List<BaseInstanceState> children = new List<BaseInstanceState>();
        nodestate.GetChildren(SystemContext, children);
        //Iterate through the children and update the necessary nodes:
        foreach (BaseInstanceState child in children)
        {
            BaseDataVariableState childvariable = child as BaseDataVariableState;
            if (childvariable != null)
            {
                switch (childvariable.BrowseName.Name)
                {
                    case "Score":
                        childvariable.Value = m_TetrisGame.Score;
                        break;
                    case "GameOver":
                        childvariable.Value = m_TetrisGame.GameOver;
                        break;
                    case "Paused":
                        childvariable.Value = m_TetrisGame.Paused;
                        break;
                    case "Seconds_Till_Unpause":
                        childvariable.Value = m_TetrisGame.SecondsTillUnpause;
                        break;
                    case "Field":
                        childvariable.Value = m_TetrisGame.DynamicField;
                        break;
                    default:
                        break;
                }
            }
        }
        // make known that this nodes and its child nodes have changed:
        nodestate.ClearChangeMasks(SystemContext, true);
        // Tetris node found, no further need to iterate:
        break;
    }
}
#endregion
```

(Again: this code should be added within the callback for the UpdateTimer!)

This is the same approach as used in “step 1” for updating the Tetris_Simulation nodes, so for an explanation of this code, see above.

This is basically what we wanted to achieve in this step: doing more or less the same as in “step 1”, but now for an actual Tetris Game.

In “step 3” we will take a look at how to show the server data in a simple Tetris GUI at the client side, besides the generic monitoring.

3 TetrisViewer on Client

All the work on the client is done in the *MainForm.cs* file: right-click on it and select “view code”.

As we saw before, when browsing the Address Space in the client you can right click on nodes and click on “monitor” to create a monitored item, to periodically receive notifications and thus data updates. We now want to link some of these monitored items and their periodically updated data to a newly created form that will display this data.

The existing EventHandler for creating a monitored item is “Browse_MonitorMI_Click”. We will make a new EventHandler that will create a monitored item as well, but also add the monitored item to a list. This list will be checked whenever a notification is received and the data corresponding to monitored items in this list, will also be uploaded to our Form, our TetrisViewer form.

So in the “Private Fields” region, add these two new instances:

```
//----------------- Tetris Fields ----------------------
private List<MonitoredItem> m_MonitoredItem_list_for_Tetris;
private TetrisFieldViewer m_TetrisViewer;
//------------------------------------------------------
```

Now for our new event handler, we want to trigger it via a new *”right-click-on-node”-menu-item*, much in the same way as is done to create a regular monitored item. How?

In the client project, go to *MainForm.cs (Design)*, click on the BrowsingMenu (bottom of screen), and in the “Type Here” box add a menu item by typing “Monitor for TetrisViewer”. After creating this new BrowsingMenu entry, select this item, go to the Properties menu on the right side of the screen, click on the “Events” icon, the yellow lightning bolt. We will add an event handler to the “Click” event: in the “click” event box, write “Browse_MonitorMI_Tetris_Click” and press enter. This will create the method stub related to this event, so we’re just left with implementing this method:

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

}
```

(note: you could cut it and paste it under the already existing “Browse_MonitorMI_Click” event handler, which handles the normal event when one wants to create a monitored item)

Now this event handler will be called whenever you click the “Monitor for TetrisViewer” menu-item in the menu that pops up when right-clicking a node in the browse window, the left window on the client form. What to implement here?

On the one hand, as mentioned before, we will want to create a normal monitored item, just as if we had clicked on the “monitor” menu-item instead of the “Monitor for TetrisViewer” menu-item. This is easily done by just calling the event handler related to clicking the “monitor” menu-item: “Browse_MonitorMI_Tetris_Click(…)”.

On the other hand, *if* calling this event handler resulted in the successful creation of a monitored item, we will add this monitored item to our special list called “m_MonitoredItem_list_for_Tetris”. Should the TetrisViewer form not be initialized yet or have been disposed of before, initialize the form and show it.

This kind of functionality could be implemented like this:

```
/// <summary>
/// Handles the Click event of the Browse_MonitorMI_Tetris control.
/// </summary>
/// <param name="sender">The source of the event.</param>
/// <param name="e">The <see cref="System.EventArgs"/> instance containing the event data.</param>
private void Browse_MonitorMI_Tetris_Click(object sender, EventArgs e)
{
    try
    {
        // Check the number of ListViewItems (= number of Monitored Items (= MI)):
        int countBefore = MonitoredItemsLV.Items.Count;

        // Call the event handler for creating a normal MI (so it is also listed in the ListView):
        Browse_MonitorMI_Click(sender, e);

        // The last Item added to the ListView for MI's i the one created in Browse_MonitorMI_Click.
        // But first check whether Browse_MonitorMI_Click did not abort before creating the new MI:
        int countAfter = MonitoredItemsLV.Items.Count;
        if (countAfter > countBefore) //this means a monitored item was succesfully created
        {
            // get the last added item and get its related Monitored Item (via its 'Tag'):
            MonitoredItem monitoredItem = (MonitoredItem)MonitoredItemsLV.Items[countAfter - 1].Tag;
            // add this item to the list of MI's that are related to our Tetris Game and TetrisViewer,
            // and also initialize and show the form if not yet done:
            if (monitoredItem.NodeClass == NodeClass.Variable) // only Variable Nodes can provide data
            {
                if (m_MonitoredItem_list_for_Tetris == null)
                { m_MonitoredItem_list_for_Tetris = new List<MonitoredItem>(); }
                m_MonitoredItem_list_for_Tetris.Add(monitoredItem);

                if (m_TetrisViewer == null || m_TetrisViewer.IsDisposed)
                {
                    m_TetrisViewer = new TetrisFieldViewer();
                    m_TetrisViewer.Show();
                }
            }
        }
    }
    catch (Exception exception)
    {
        MessageBox.Show(exception.Message, this.Text, MessageBoxButtons.OK, MessageBoxIcon.Error);
    }
}
```

The above explanation and the comments in the code should explain this event handler well enough.

Whenever a notification is received, it is processed in the event handler “MonitoredItem_Notification(…)”. Value, StatusCode, Timestamp, etc. are extracted from the notification and the ListViewItem corresponding to the notification’s monitored item is updated.

We also want to update the TetrisViewer form, which has to be done *if* the monitored item is in our list “m_MonitoredItem_list_for_Tetris”. So at the end of the “try” block, in the “MonitoredItem_Notification(…)” event handler (just after updating the ListViewItem), add the code:

```
// ------------------------ Update the TetrisViewer (if needed): ----------------------------
if (m_MonitoredItem_list_for_Tetris != null)
{
    if (m_MonitoredItem_list_for_Tetris.Contains(monitoredItem))
    {
        UpdateTetrisViewer(notification, monitoredItem.DisplayName);
    }
}
// ------------------------------------------------------------------------------------------
```

Thus, if the list is initialized and if the monitored item is in the list, update the form:

```
/// <summary>
/// Updates the TetrisViewer (a new notification has arrived).
/// </summary>
private void UpdateTetrisViewer(MonitoredItemNotification notification, string displayName)
{
    try
    {
        // Update a field of the TetrisViewer according to the passed displayName string:
        switch (displayName)
        {
            case "Score":
                m_TetrisViewer.Score = (int)notification.Value.Value;
                break;
            case "GameOver":
                if ((bool)notification.Value.Value) m_TetrisViewer.MyMessage = "GAME OVER!!!";
                else m_TetrisViewer.MyMessage = "Game Reset!";
                break;
            case "Paused":
                m_TetrisViewer.GameActive = !(bool)notification.Value.Value;
                break;
            case "Seconds_Till_Unpause":
                m_TetrisViewer.SecsRemaining = Convert.ToUInt32(notification.Value.Value);
                break;
            case "Field":
                m_TetrisViewer.Field = (int[,])((Matrix)notification.Value.Value).ToArray();
                break;
            default:
                break;
        }
    }
    catch (Exception exception)
    {
        MessageBox.Show(exception.Message, this.Text, MessageBoxButtons.OK, MessageBoxIcon.Error);
    }
}
```

The argument “notification” is a “MonitoredItemNotification”, and “notification.Value.Value” is the C# object that has to be cast into the correct format for the TetrisViewer form.

You should now be able to run the client and connect to the server. Right click the VariableNodes under Tetris_Game and select “Monitor for TetrisViewer”. This will pop up the Tetris Viewer form. Do this for each variable and you can visually monitor the “Tetris Process” at the server side. Also try clicking on the *“Pause for …”* (…seconds) button: you will get a message that this functionality is not yet implemented. In the next step we will add a method to our server, a method to pause the tetris game for a certain amount of second, and add the functionality to call it from the client side.

Extra note:
To make more use of the Information Modeling of OPC UA, another possible implementation could make use of a complex ObjectType, for example a TetrisGameType (Figure 7). The Server would have to initialize an instance of this ObjectType and keep the values of the Variable Node components up-to-date. The client could – once it encounters an instance of this complex ObjectType, by checking in the DataType Hierarchy of the server that it is indeed an instance of the TetrisGameType – know at what browsepaths it can get the data it needs, to keep its TetrisViewer form up to date in this case.

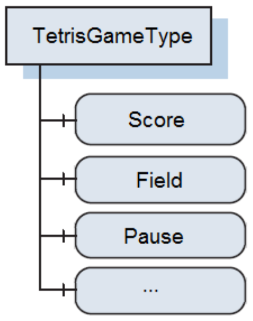


Figure 7:
A possible Type definition for a TetrisGame data structure.

4 Methods

We will add functionality on both client and server side this time, functionality to call methods. Methods will now (generally) refer to “UA Methods”, i.e. nodes in the server Address Space that have specific functionality related to them when called.

4.1 *Method implementation on Server*

Let us start at the **server side**. We basically need to do two things:

- Add a Method node to the Tetris_Game node (+ InputArguments and OutputArguments as properties of this Method node)
- Implement an event handler that is called whenever the Method is called by the client

So the Tetris_Game node will have a new component (via a “HasComponent” reference): the Pause_Method node. This Method node will have the two standard properties: an InputArguments and an OutputArguments property. In “CreateAddressSpace(…)” the added code is shown on the next page. What is done in that code?

First the Method node is added in much the same way as previous nodes were added. The InputArguments and OutputArguments properties are both standard properties for a Method node. A Property node is a special kind of Variable node. The property’s Value attribute is an array of the DataType ‘Argument’ and the Argument DataType is a structure of again multiple fields (Name, DataType, ValueRank, ArrayDimensions, Description: these are all initialized in the code above!). Figure 8 illustrates this (as well as the similarities with a method in a programming language) with a generic example.

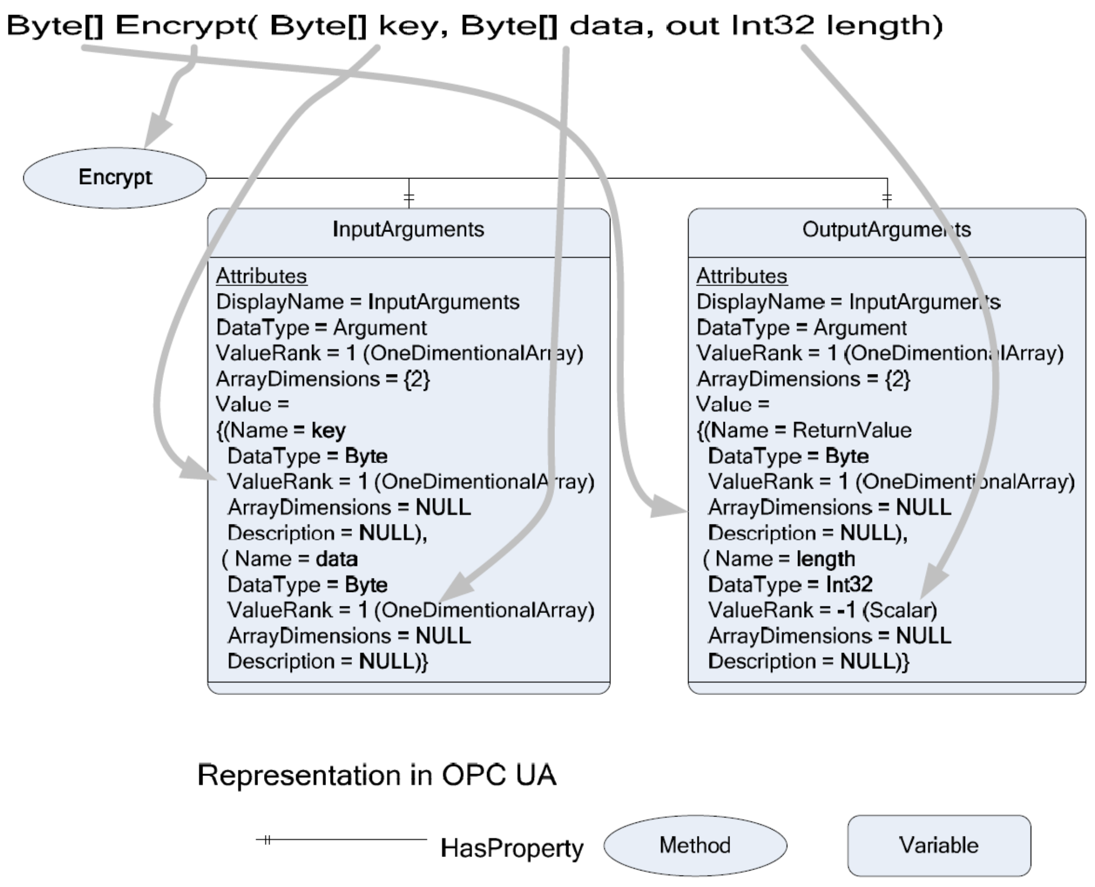


Figure 8:
The mapping of a method in pseudo code to an OPC UA Method.

The code for the Method node and the argument nodes:

```
#region tetris method node
//------------------------------ The method node for the tetris game --------------------------

// ------ The  method state itself: ------
MethodState pauseMethodNode = new MethodState(tetrisGameNode);
pauseMethodNode.NodeId = new NodeId(id++, NamespaceIndex);
pauseMethodNode.BrowseName = new QualifiedName("Pause_Method", NamespaceIndex);
pauseMethodNode.DisplayName = pauseMethodNode.BrowseName.Name;
pauseMethodNode.SymbolicName = pauseMethodNode.BrowseName.Name;
pauseMethodNode.Description = new LocalizedText("A method to pause the TetrisGame for X seconds");
pauseMethodNode.ReferenceTypeId = ReferenceTypeIds.HasComponent;
pauseMethodNode.UserExecutable = true;
pauseMethodNode.Executable = true;

// ------ Add the InputArguments: ------
pauseMethodNode.InputArguments = new PropertyState<Argument[]>(pauseMethodNode);
pauseMethodNode.InputArguments.NodeId = new NodeId(id++, NamespaceIndex);
pauseMethodNode.InputArguments.BrowseName = BrowseNames.InputArguments;
pauseMethodNode.InputArguments.DisplayName = new LocalizedText("InputArgs_PauseMethod");
pauseMethodNode.InputArguments.SymbolicName = pauseMethodNode.InputArguments.DisplayName.Text;
// These are properties of the method state (closer relation with parent than "HasComponent"):
pauseMethodNode.InputArguments.TypeDefinitionId = VariableTypeIds.PropertyType;
pauseMethodNode.InputArguments.ReferenceTypeId = ReferenceTypeIds.HasProperty;
//Attributes related to the Value (= Argument):
pauseMethodNode.InputArguments.DataType = DataTypeIds.Argument;
pauseMethodNode.InputArguments.ValueRank = ValueRanks.Scalar;
// Create the Argument (= Value of PropertyNode 'InputArguments'):
Argument[] args_in = new Argument[1]; // only one inputargument
args_in[0] = new Argument();
args_in[0].Name = "SecondsToPause";
args_in[0].Description = "The number of seconds to pause the Tetris Game";
args_in[0].DataType = DataTypeIds.UInt32;
args_in[0].ValueRank = ValueRanks.Scalar;
// Add the Argument as the Value of the method Node:
pauseMethodNode.InputArguments.Value = args_in;

// ------ Add the OutputArguments: ------
pauseMethodNode.OutputArguments = new PropertyState<Argument[]>(pauseMethodNode);
pauseMethodNode.OutputArguments.NodeId = new NodeId(id++, NamespaceIndex);
pauseMethodNode.OutputArguments.BrowseName = BrowseNames.OutputArguments;
pauseMethodNode.OutputArguments.DisplayName = new LocalizedText("OutputArgs_PauseMethod");
pauseMethodNode.OutputArguments.SymbolicName = pauseMethodNode.OutputArguments.DisplayName.Text;
// These are properties of the method state (closer relation with parent than "HasComponent"):
pauseMethodNode.OutputArguments.TypeDefinitionId = VariableTypeIds.PropertyType;
pauseMethodNode.OutputArguments.ReferenceTypeId = ReferenceTypeIds.HasProperty;
//Attributes related to the Value (= Argument):
pauseMethodNode.OutputArguments.DataType = DataTypeIds.Argument;
pauseMethodNode.OutputArguments.ValueRank = ValueRanks.Scalar;
// Create the Argument (= Value of PropertyNode 'InputArguments'):
Argument[] args_out = new Argument[1]; // only one outputargument
args_out[0] = new Argument();
args_out[0].Name = "SecondsToPause_StringResult";
args_out[0].Description = "A string telling the Nr. of seconds the game is paused";
args_out[0].DataType = DataTypeIds.String;
args_out[0].ValueRank = ValueRanks.Scalar;
// Add the Argument as the Value of the method Node:
pauseMethodNode.OutputArguments.Value = args_out;

// ------ set up myPauseMethod method handler: ------
pauseMethodNode.OnCallMethod = new GenericMethodCalledEventHandler(OnPauseMethod);
// (From now on OnPauseMethod(...) will be called whenever this method is called by the client!)

// ------ Add the Method Node as child to the Tetris_Game parent node... ------
tetrisGameNode.AddChild(pauseMethodNode);
// ------ ... and add it to the predefined nodes: ------
AddPredefinedNode(SystemContext, pauseMethodNode);
//--------------------------------------------------------------------------------------------
#endregion
```

The line `pauseMethodNode.OnCallMethod = new GenericMethodCalledEventHandler(OnPauseMethod);` adds an event handler to the event when the method is called from the client. This event handler can look like this:

```
        #region OnPauseMethod
        public ServiceResult OnPauseMethod(
                    ISystemContext context,
                    MethodState method,
                    IList<object> inputArguments,
                    IList<object> outputArguments)
        {
            // All arguments must be provided:
            if (inputArguments.Count < 1) return StatusCodes.BadArgumentsMissing;

            // Check the data type of the input arguments:
            uint? secondsToPause = inputArguments[0] as uint?;
            if (secondsToPause == null) return StatusCodes.BadTypeMismatch;

            // Do the useful things this method is supposed to do:
            lock (m_lock)
            {
                // Call the pause function on the tetris game:
                m_TetrisGame.pause((uint)secondsToPause);

                // The calling function sets default values for all output arguments,
                // you only need to update them here:
                uint checkSeconds = m_TetrisGame.SecondsTillUnpause;
                // Did the "Pause for ... second" work?:
                if (checkSeconds == secondsToPause) // yes!
                {
                    outputArguments[0] = "Success: Game was paused for " + checkSeconds.ToString() + " seconds";
                    return ServiceResult.Good;
                }
                else // no!
                {
                    outputArguments[0] = "NO Success: Game was already manually paused at serverside!";
                    // The method was succesfully finished, but the result might not be what the client wanted, so
try again later:
                    return StatusCodes.GoodCallAgain;
                }
            }
        }
        #endregion
```

(***note**: in the "Private Fields" region, add:* `object m_lock = new object();` *)*

It does little more than what we already said we wanted this method to do: it calls the "Pause(…)" method (not a "UA Method" here!) on the Tetris Game and returns a StatusCode Good.

This should already work fine: if you also installed the OPC Foundation "Sample Client" (or "UAExpert"), open it, connect to the server and browse till you see the Pause_Method Method node. Right-click it and select "Call…". In the pop-up window double-click on the inputargument and set the initial '0' to any 'uint' value (for example '20') and press the "Call" button. If your tetris game is running, it should now be paused for 20 seconds and the outputargument will be "Success: Game was paused for X seconds". If it was not yet running or already paused at the server side, you will get another outputargument: "NO Success: Game was already manually paused at serverside!".

The next paragraph will deal with the functionality at our clients' side to call this UA method via the TetrisViewer GUI.

4.2 Method Calling from Client

Now for the **Client side**. First of all, you will notice that when browsing the Address Space with our own client, we don’t get to see the Pause_Method method node! There is a reason for this. Go to the code of *MainForm.cs*, go to “PopulateBranch(…)”. In this method you will see the line:

```
nodeToBrowse1.NodeClassMask = (uint)(NodeClass.Object | NodeClass.Variable);
```

This means any browsing done with this client will only return Object and Variable Nodes, none of the other six NodeClasses (see Figure 1). But we also want Method nodes returned, so change the previous line to:

```
nodeToBrowse1.NodeClassMask = (uint)(NodeClass.Object | NodeClass.Variable | NodeClass.Method);
```

Browsing now will also return the Pause_Method Method node as component of the Tetris_Game Object node.

Concerning the callinf of the method, a few things need to be done:

- The TetrisFieldViewer class has a “PauseMethodClickedEventHandler” field that has to be initialized in the client (*MainForm.cs*): “m_TetrisViewer.OnPauseClicked = …” . When the *“Pause for …”* button is pressed, this event is raised. So a callback for this event needs to be implemented (in *MainForm.cs*) in which the Pause_Method on the server side is called.
- To call a method, we need the MethodNode’s NodeId and its parent node’s NodeId: these are two arguments that need to be passed in “m_session.Call(…)”. So we will have to make two new private fields (two NodeId’s) and implement functionality to find those nodes in the Address Space and initialize the two NodeId fields.

The two new entries in the “Private Fields” region:

```
private NodeId m_pauseMethod_NodeId;
private NodeId m_pauseMethod_Parent_NodeId;
```

How and when to initialize these? Let us make another *”right-click-on-node”-menu-item* for the Browsing window that will raise an event on click with an event handler that will check whether the selected node is a valid Method node to be coupled with the *“Pause for …”* button in the TetrisViewer form. Go to *MainForm.cs (design)*, click on the “BrowsingMenu” on the bottom of the screen and make a new menu item in “Type Here” called “Method for *“Pause for …”*”. Then select this new item and go to its event handlers (Properties window => yellow lightening icon) and for the “click” event, add “Browse_SetPauseMethod_Click” and press enter (which will bring you to the code for this event handler).

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

}
```

I copied and pasted this event handler under the “Browse_MonitorMI_Click(…)” event handler. In here we have to check whether the selected node is a method node and has a valid BrowseName (“Pause_Method”). Since we also need its parent node, we do the same checks for that node. Store the NodeId’s if the checks were passed. We’re left with initializing the “m_TetrisViewer.OnPauseClicked” event handler. This looks like this:

```
        /// <summary>
        /// Handles the Click event of the Browse_SetPauseMethod control.
        /// </summary>
        /// <param name="sender">The source of the event.</param>
        /// <param name="e">The <see cref="System.EventArgs"/> instance containing the event data.</param>
        private void Browse_SetPauseMethod_Click(object sender, EventArgs e)
        {
            try
            {
                // Check if operation is currently allowed:
                if (m_session == null || BrowseNodesTV.SelectedNode == null) return;

                // Get the tag (= reference description) of the selected node:
                ReferenceDescription methodReferenceDescription =
                    (ReferenceDescription)BrowseNodesTV.SelectedNode.Tag;

                // Check whether it's indeed a Method node + check whether it's the correct method:
                if (methodReferenceDescription.NodeClass != NodeClass.Method
                    || methodReferenceDescription.BrowseName.Name != "Pause_Method") return;

                // Also check whether the parent is a Tetris_Game Object Node:
                ReferenceDescription parentReferenceDescription =
                    (ReferenceDescription)BrowseNodesTV.SelectedNode.Parent.Tag;
                if (parentReferenceDescription.NodeClass != NodeClass.Object
                    || parentReferenceDescription.BrowseName.Name != "Tetris_Game") return;

                // We got this far without returning, so those two nodes are the correct nodes:
                m_pauseMethod_NodeId = (NodeId)methodReferenceDescription.NodeId;
                m_pauseMethod_Parent_NodeId = (NodeId)parentReferenceDescription.NodeId;

                // Just left with enabling the calling of the method through the "Pause for ..." button of the
TetrisViewer form
                // (but first initialize the TetrisViewer form if not yet done)
                if (m_TetrisViewer == null || m_TetrisViewer.IsDisposed)
                {
                    m_TetrisViewer = new TetrisFieldViewer();
                    m_TetrisViewer.Show();
                }
                m_TetrisViewer.OnPauseClicked =
                    new TetrisFieldViewer.PauseMethodClickedEventHandler(PauseMethod_in_TetrisViewer_Click);

                // Indicate succesful coupling of the pause method to the TetrisViewer form:
                MessageBox.Show("Method Succesfully coupled to TetrisViewer Form \"Pause for ...\" button!");
            }
            catch (Exception exception)
            {
                MessageBox.Show(exception.Message, this.Text, MessageBoxButtons.OK, MessageBoxIcon.Error);
            }
        }
```

We are now left with the implementation of "PauseMethod_in_TetrisViewer_Click(…)":

```
        /// <summary>
        /// Handles the Click event of the "Pause for ..." button in the Tetris Viewer form.
        /// </summary>
        /// <param name="secondsToPause">The number of seconds to pause the "Tetris process" on the server.</param>
        private string PauseMethod_in_TetrisViewer_Click(uint secondsToPause)
        {
            //Simply just call the method (necessary arguments (two NodeId's) were previously determined):
            IList<object> outputArguments = m_session.Call(
                m_pauseMethod_Parent_NodeId,
                m_pauseMethod_NodeId,
                secondsToPause);
            // return the returned outputarguments:
            return outputArguments[0].ToString();
        }
```

The "m_session.Call(…)" needs the two previously determined NodeId's as argument, as well as the InputArguments (= number of seconds to pause in this case). So from the moment we have set the correct Pause_Method Method node (i.e. set its and its parent's NodeId), whenever the *"Pause for …"* button is clicked in the TetrisViewer form, this event is dispatched to the event handler above in which the Pause Method on the server will be called.

Run the client and connect to the tetris server: you are able to set the Pause_Method Method node as the Method to be linked with the *"Pause for …"* button on the TetrisViewer form and pause the server's Tetris game.

5 Events

We will create our own EventType, the server will generate this type of events and the client will subscribe to it at the client side. The event will be the “TetrisLinesMadeEventType” and will be raised whenever one or more tetris lines are made in the Underlying System TetrisGame.

5.1 Generating Events on Server

First, let us implement the **Server-side** part. Take a look at part of the EventType hierarchy in figure below. What we want to do is add our own EventType in here, as a subtype of the BaseEventType. Type hierarchies, like the ones for DataTypes (see Figure 6), Complex Object– and VariableTypes, ReferenceTypes, etc., can always be extended to the needs of the UA user. As you can see from the node icons (see Figure 1) used in Figure 9 below: EventTypes are ObjectTypes.

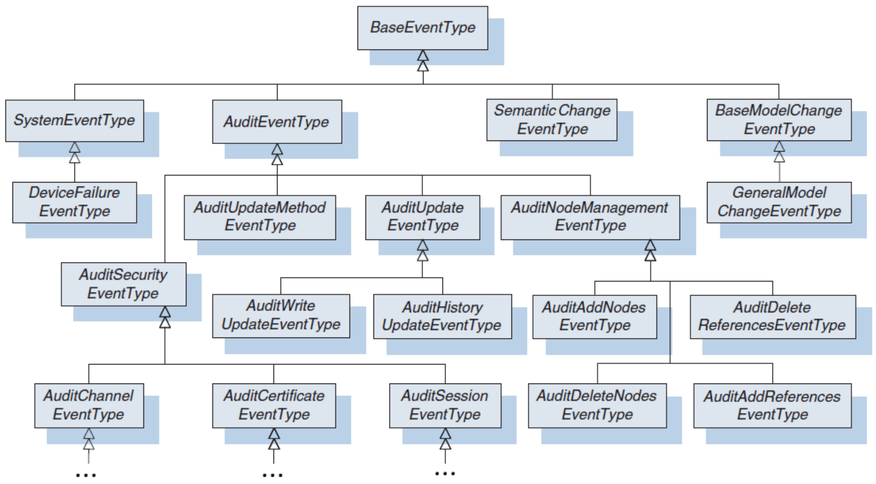

BaseEventType
SystemEventType
AuditEventType
Semantic Change EventType
BaseModelChange EventType
DeviceFailure EventType
AuditUpdateMethod EventType
AuditUpdate EventType
AuditNodeManagement EventType
GeneralModel ChangeEventType
AuditSecurity EventType
AuditWrite UpdateEventType
AuditHistory UpdateEventType
AuditAddNodes EventType
AuditDelete ReferencesEventType
AuditChannel EventType
AuditCertificate EventType
AuditSession EventType
AuditDeleteNodes EventType
AuditAddReferences EventType
...
...
...

Figure 9:
The UA EventType Hierarchy.

Just defining a new EventType – as a new ObjectType that is a subtype of the BaseEventType – will not do: there is event data related to EventTypes, called the event fields, in the form of Properties. An example of this in Figure 10: the standard UA (blue) EventTypes and their Properties (= event fields: EventId, EventType, Message, etc.) can be extended by new EventTypes with new event fields. In our case the new TetrisLinesMadeEventType with two new properties, “NrOfLines” and “PointsScored”, is added to the hierarchy as a subtype of the BaseEventType. Its properties will also include the BaseEventType properties (inheritance as in OOP)!

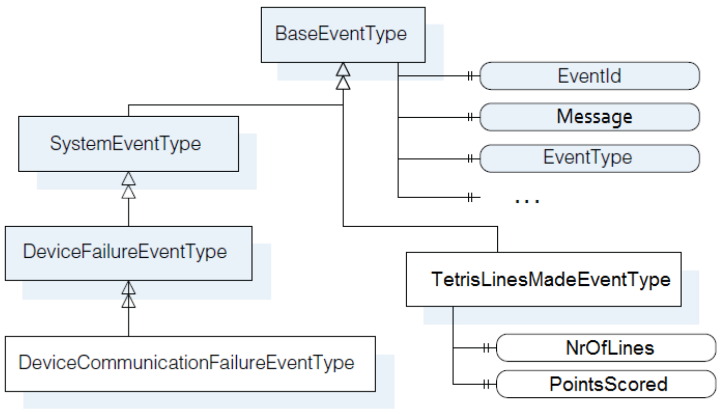


Figure 10:
New EventTypes with new event fields and Inheritance in an extract from the UA EventType Hierarchy.

Before adding these nodes to the EventType Hierarchy in our NodeManager’s “CreateAddressSpace(…)”, a few **constants** have to be defined, constants found in the project’s *TetrisEventsConstants.cs* file:

- The Object and Variables **identifiers** (for the event and its properties (= event fields) respectively): These fixed identifiers will be used to create the fixed NodeId's of the new Nodes in the Type Hierachry. Remember that a NodeId uniquely defines a Node with an *identifier* and a *namespaceindex* (that refers to a namespace in which the identifier is unique). The identifiers will be *uint*'s and we will need three (one for the TetrisLinesMadeEventType and two for the two properties).
- The **NodeId**'s for the Object (EventType) and Variables (Properties of events): Creation of these will be based on the previously defined constants *identifiers*, as noted above. Actually **ExpandedNodeId**'s will be used. These do not just contain a *namespaceindex*, but the full *namespace*. These can be translated to normal NodeId's in client or server with the help of the *Server NamespaceTable* (which links multiple *namespaceindices* with their *namespace*).
- Finally also standard **BrowseNames** will be defined for the EventType and its Properties.

Go over the code, you will find all the constants explained above. We will need these to create our nodes in our NodeManager and add them to the EventType Hierarchy. Open the TetrisNodeManager, go to (again!) "CreateAddressSpace(…)" and add:

```
#region TetrisLinesMadeEventType
//----------------Add the TetrisLinesMadeEventType to the EventTYpe Hierarchy -----------------

// Create the Type information                        (i.e. Object type with attached properties)
// to represent our TetrisLinesMadeEventType          (with its 2 properties),
// in the OPC UA type model's EventType Hierarchy     (which is accessible in the server Address Space)
BaseObjectTypeState newEventType = new BaseObjectTypeState(); // remeber EventTypes are ObjectTypes!
newEventType.NodeId = ExpandedNodeId.ToNodeId(
       ObjectTypeIds_TetrisEvents.TetrisLinesMadeEventType,
       SystemContext.NamespaceUris); // the fixed (in 'constants' defined) NodeId is used here!
newEventType.SuperTypeId = ObjectTypeIds.BaseEventType; // our new EventType will inherit from
BaseEventType
newEventType.BrowseName = new QualifiedName(BrowseNames_TetrisEvents.TetrisLinesMadeEventType,
NamespaceIndex);
newEventType.DisplayName = new LocalizedText(BrowseNames_TetrisEvents.TetrisLinesMadeEventType);
newEventType.IsAbstract = true; // events are NOT nodes in the Address Space (no instances can be made)s

// Create the two Properties.
// NrOfLinesMade:
PropertyState<uint> nrOfLines = new PropertyState<uint>(newEventType);
nrOfLines.NodeId = ExpandedNodeId.ToNodeId(
                      VariableIds_TetrisEvents.TetrisLinesMadeEventType_NrOfLines,
                      SystemContext.NamespaceUris);
nrOfLines.BrowseName = new QualifiedName(BrowseNames_TetrisEvents.NrOfLines, NamespaceIndex);
nrOfLines.DisplayName = new LocalizedText(BrowseNames_TetrisEvents.NrOfLines);
nrOfLines.Description = new LocalizedText("An unsigned Integer representing the number of lines made in
Tetris Game");
nrOfLines.ReferenceTypeId = ReferenceTypeIds.HasProperty;
nrOfLines.TypeDefinitionId = VariableTypeIds.PropertyType;
nrOfLines.DataType = DataTypeIds.UInt32;
nrOfLines.ValueRank = ValueRanks.Scalar;
// PointsScored:
PropertyState<double> pointsScored = new PropertyState<double>(newEventType);
pointsScored.NodeId = ExpandedNodeId.ToNodeId(
                      VariableIds_TetrisEvents.TetrisLinesMadeEventType_PointsScored,
                      SystemContext.NamespaceUris);
pointsScored.BrowseName = new QualifiedName(BrowseNames_TetrisEvents.PointsScored, NamespaceIndex);
pointsScored.DisplayName = new LocalizedText(BrowseNames_TetrisEvents.PointsScored);
pointsScored.Description = new LocalizedText("A double representing the points scored when making
'NrOfLines' lines");
pointsScored.ReferenceTypeId = ReferenceTypeIds.HasProperty;
pointsScored.TypeDefinitionId = VariableTypeIds.PropertyType;
pointsScored.DataType = DataTypeIds.Double;
pointsScored.ValueRank = ValueRanks.Scalar;

// The constants defined in "TetrisEventsConstants.cs" were used above
// for the creation of the NodeId's, BrowseNames and DisplayNames!

//Add these two properties as children to the newEventType (= TetrisLinesMadeEventT:ype)
newEventType.AddChild(pointsScored);
newEventType.AddChild(nrOfLines);

// ...And add newEventType (= TetrisLinesMadeEventType) it to the predefined nodes:
AddPredefinedNode(SystemContext, newEventType); // will also add the new type to the Type Hierarchy!
//-------------------------------------------------------------------------------------------
#endregion
```

In the first block the EventType is created as an ObjectType – a *BaseObject**Type**State* is used now instead of *BaseObjectState* like before – and its attributes are set. The NodeId is derived from the constant ExpandedNodeId defined in *TetrisEventsConstants.cs*. The SuperTypeId is set to the NodeId of the BaseEventType node: this information will be used in "AddPredefinedNodes(…)", which is called later, to make the reference (forward and backward) between the BaseEventType node (already existent in the EventType Hierarchy) and our newly created EventType (= TetrisLinesMadeEventType) node.

In the second and third block the two properties – the event fields of our event – are initialized and its attributes are set in much the same way as either the newEventType above or as nodes in the previous steps (not much new here). The two properties are added as children of the newEventType, which itself is added to the predefined nodes (+ it is added to the EventType Hierarchy).

The new Type should have been added to UA's type model now. To check this, connect to the Tetris Server with the OPC Foundation Sample client (copy/paste the server-endpoint to the Sample Client and click 'connect'). Unfortunately this client starts browsing from the "Objects" node, but to access the server's type model (= standard UA Type Hierachy + custom Types added), we need the "Root" node, which organizes the "Views", "Objects" and "Types". The latter contains the server's type model. What to do? Right-click on any node in the browse window, select "Browse options …" and in the menu that pops up choose 'Both' instead of 'forward' for the "Browse Direction" option. You could also select "Show References" on Right-clicking a node. Now you can Browse to the EventType Hierachy and see our new EventType in the type model (see Figure 11).

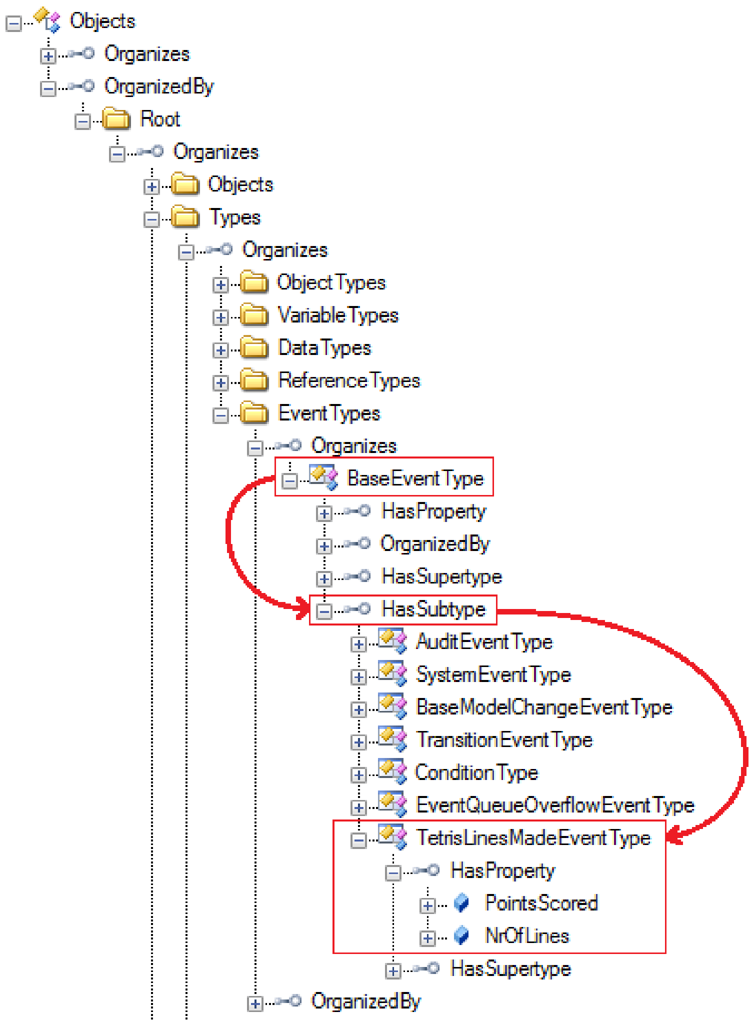
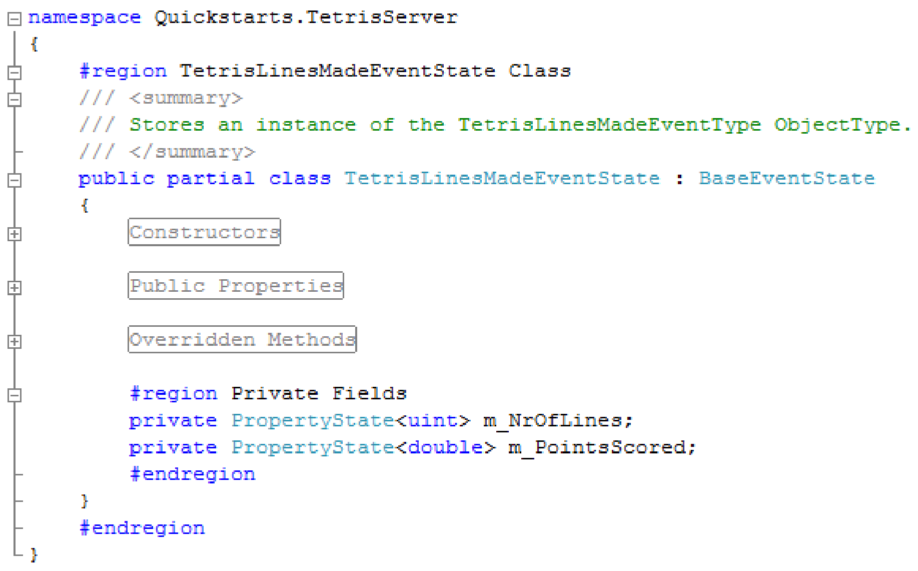


Figure 11:
Our new EventType (TetrisLinesMadeEventType) added to the UA EventType Hierarchy.

Besides having our EventType in the UA type model, we also need to have a class of which the instantiations will represent TetrisLinesMadeEvents. When we created nodes before, classes were used like MethodState for Method nodes, PropertyState for Property nodes, BaseObjectState to instantiate the most general Object node, etc. The same is true for events: if we want to make a basic event based on the BaseEventType, we use the BaseEventState class. But keep in mind that even though EventTypes ***are*** nodes in a type hierarchy in the Address Space, the events itself ***are not*** actual nodes in the Address Space! Likewise we have to make a class called TetrisLinesMadeEventState that will inherit from the BaseEventState class. This class is defined in the *TetrisEventsClasses.cs* file found in the Tetris project folder. The class includes: fields for the two properties – “NrOfLines” and “PointsScored” – as instantiations of the PropertyState class, functionality to retrieve our EventType’s fixed NodeId (see the paragraph on the defined constants above) and functionality to get and find the event’s ‘children’, i.e. the event fields in the form of properties. Going through the code, you will recognize the above mentioned content and functionality (see snapshot Figure 12):


```
namespace Quickstarts.TetrisServer
{
    #region TetrisLinesMadeEventState Class
    /// <summary>
    /// Stores an instance of the TetrisLinesMadeEventType ObjectType.
    /// </summary>
    public partial class TetrisLinesMadeEventState : BaseEventState
    {
        Constructors

        Public Properties

        Overridden Methods

        #region Private Fields
        private PropertyState<uint> m_NrOfLines;
        private PropertyState<double> m_PointsScored;
        #endregion
    }
    #endregion
}
```

Figure 12:
Snapshot of the TetrisLinesMadeEventState Class, already defined in the *TetrisEventsClasses.cs* file.

Now that all the ‘infrastructure’ is in place, the events still have to be made and sent to the client. One option could be to implement a callback in our NodeManager and pass it to the underlying TetrisGame class in the *TetrisGame.cs* file. Upon creating lines and scoring points the the TetrisGame the game could push its event data (nr of lines made and points scored) to the NodeManager through the callback. But so far it has been the TetrisNodeManager that has done all the pushing and pulling: periodically ‘pull’ underlying game data from the TetrisGame and update the nodes, and upon a call to the Pause_Method ‘push’ this pause to the underlying TetrisGame.

So to refrain from getting into the TetrisGame code, let us choose another option. In the TetrisNodeManager, in the periodically called “UpdateTimerCallback(…)”, during the update of the Score node in one of the cases of the switch-statement, before actually updating the Value of the Score node: pass the Score node’s current Value and the new Score value of the TetrisGame – m_TetrisGame.Score – to a new method “Check_for_TetrisLinesMadeEvent(…)”. This method will create and send an event to the client in the case of an increment in score.

Go to “UpdateTimerCallback(…)” and in the “Update Tetris Game Nodes” region, go to the switch statement and add a call to “Check_for_TetrisLinesMadeEvent(…)” in the first case (case “Score”):

```
...
case "Score":
 if (childvariable.Value != null)
 { Check_for_TetrisLinesMadeEvent((int)childvariable.Value, m_TetrisGame.Score, nodestate as BaseObjectState); }
 // arguments: (old score value, new score value, source ObjectNode that generates the Event)
 childvariable.Value = m_TetrisGame.Score;
 break;
...
```

The implementation of “Check_for_TetrisLinesMadeEvent(…)”:

```
        #region Check_for_TetrisLinesMadeEvent

        private void Check_for_TetrisLinesMadeEvent(int oldScore, int newScore, BaseObjectState sourceNode)
        {
            //---------- determine whether actual lines were made and score increased -----------
            if (!(newScore > oldScore) || sourceNode == null) return;

            // determine score inscrease:
            int pointsScored = newScore - oldScore;
            // ... and the nr of lines (slightly dirty implementation: TetrisGame knowledge assumed...)
            int nrOfLinesMade = 0;
            switch (pointsScored)
            {
                case 10:
                    nrOfLinesMade = 1;
                    break;
                case 30:
                    nrOfLinesMade = 2;
                    break;
                case 60:
                    nrOfLinesMade = 3;
                    break;
                case 100:
                    nrOfLinesMade = 4;
                    break;
                default:
                    return;
            }

            //------------------------------ Create the event -----------------------------------
            //TetrisLinesMadeEventState linesMadeEvent = new TetrisLinesMadeEventState(null);
            TetrisLinesMadeEventState linesMadeEvent = new TetrisLinesMadeEventState(null);

            // construct the message of the event (on of the event fields):
            TranslationInfo message = new TranslationInfo(
                "TetrisLinesMade",
                "en-US",
                "In the TetrisGame {0} lines where made for a total of {1} points",
                nrOfLinesMade,
                pointsScored);

            // initialize the event:
            linesMadeEvent.Initialize(
                SystemContext,
                sourceNode,
                EventSeverity.Low,
                new LocalizedText(message));

            // set our two new fields:
            linesMadeEvent.SetChildValue(SystemContext, BrowseNames_TetrisEvents.NrOfLines, nrOfLinesMade, false);
            linesMadeEvent.SetChildValue(SystemContext, BrowseNames_TetrisEvents.PointsScored, pointsScored,
false);

            //----------------------------- Report the event ------------------------------------
            sourceNode.ReportEvent(SystemContext, linesMadeEvent);
        }

        #endregion
```

This alone will not work yet: Events are generated at certain nodes, but have to propagate upwards in the Notifier hierarchy of the Address Space, follow a chain of “HasNotifier” references, up to the Server Object node, which collects and reports *all* events. A client subscription for events from the Server Object should result in the client receiving all events – of the type(s) he subscribed for – generated in the Address Space. So our TetrisGame Object node that will generate events has to be connected with a “HasNotifier” reference to the server object. Go to “CreateAddressSpace(…)” again and add the following three lines of code in the “tetris game nodes” region:

(before the comment `// -------------------- Nodes containing data --------------------` *and just after the initialization of the Tetris_Game node and its attributes and references)*

```
//------------- Enable tetris node to report events ---------------
//set the tetrisGameNode as EventNotifier:
tetrisGameNode.EventNotifier = EventNotifiers.SubscribeToEvents;
// Add a HasNotifier reference from Server object to our tetris node:
Server.CoreNodeManager.AddReference(ObjectIds.Server, ReferenceTypeIds.HasNotifier, false, tetrisGameNode.NodeId,
false);
// Also do the inverse:
// connect the tetrisGame node to the Server object by making it a Root Notifier (of our Node Manager)
AddRootNotifier(tetrisGameNode); // an inverse reference from our tetris node to the Server object is added in here
// (Note: A node added as a root notifier will report its events directly to the server object.
//  This is not UA specific, but an SDK specific implementation.)
//-----------------------------------------------------------------
```

The comments explain the code. This should be all the work on the Server-side: try and run the server and connect to it with the “Generic Event Client” you will find in the “Extra” folder under “step 5” (The “Quickstarts.SimpleEventsClient.exe”). Once connected, whenever you make a line in the tetris game, the Cient will receive an event. The limitation here is that this client is subscribed to all BaseEventTypes, which means it will receive all subtypes and thus all types of events, but it will only receive the BaseEventType fields (which includes Source Node, Event Type, Time and Message, among others), even if the event is a subtype of the BaseEventType with more fields of its own. The client we will see next will be able to dynamically handle different events with different fields.

5.2 Receiving Events at Client

Now let us do some work at the **Client-side**, so that the events sent by the server are received by the client and all information is processed.

First of all, copy the three files of a new TetrisFieldViewer form – a new form that also includes a textbox to display the events – from the folder “TetrisFieldViewer_With_Events” in the “Extra” folder under “step 5”, and paste these into the “client” folder of your project, overriding the previous three *TetrisFieldViewer* files (*.cs*, *.Designer.cs* and *.resx*). Now, back to events in UA.

Basically OPC UA just gives you the building blocks and basic functionality. How you want to use and combine these is partially up to the developer and can be as basic or as complex as is needed for the practical application. In our case we also have different options on how to cope with the events generated by the client.

One basic approach was already discussed at the end of the previous step, where a generic event client subscribes to all BaseEventTypes and basic event fields and thus receives all events from the server, but misses out on specific event information.

The other extreme approach would be to browse the entire EventType Hierarchy of the server and subscribe to all events’ specific event fields: the client wouldn’t miss out on any information, but at the cost of a big overhead.

Besides these two extreme approaches – which would be used in situations where the client has no clue as to what kind of events it will receive – another approach could be used in the situation where there is advance knowledge of what kind of events the clients will receive (the server will only send certain specific types of events) or where the client is only interested in certain specific events (the server generates all kinds of events, but the client is interested in only a few types of events). A subscription could be made with an event filter that would only ask for a few specific event types with specific event fields. This is depicted in Figure 13. This would require either hardcoded knowledge of the event types and their constants and event fields (ExpandedNodeId's, identifiers, BrowseNames etc.: we could copy some server files – see 5.1 – to the client), or some other knowledge that would allow the client to go get all the just mentioned information at the server side.

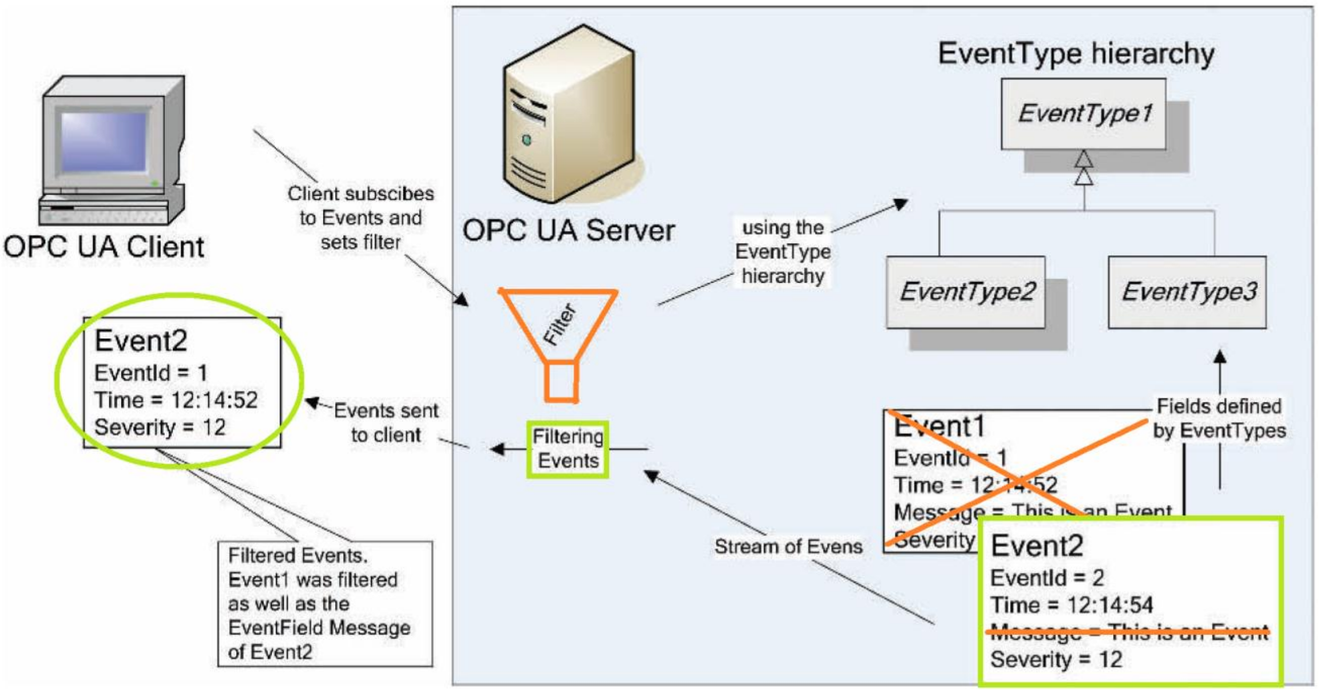


Figure 13:
Event subscriptions with event type and event field filtering.

The approach taken in the tutorial is somewhat in between. Initially we will just subscribe to the BaseEventTypes (so we receive all types of events) and its event fields (only basic event fields and thus limited information). Whenever we receive an event, we will process it and display its event field Values, but we will also check whether the event is of the BaseEventType or a subtype of it. This check is based on the EventType NodeId that is one of the BaseEventType event fields. If this NodeId differs from the BaseEventType NodeId we know the event is a subtype. Based on this NodeId we can look up the corresponding EventType node in the EventType Hierarchy, get its specific event fields and add these event fields to the subscription's event filter. From then on, whenever an EventType of this type is received, its specific event fields will also be included and returned in an event notification, and can also be processed and displayed.

While this approach does make the event subscription dynamic and adaptable to the kind of events that are received by the client – and serves as a nice example for this tutorial on how to subscribe to events and change the subscription later on – it might not be an appropriate approach for practical situations: the first event *of each type* received will always be incomplete since its specific event fields were not yet added to the subscription's monitored item filter and are only added to it after the first reception and thus only from the second event on will that type of event be reported with all of its specific event fields!

To the UA client project now. From the moment we click the connect button and a connection to the server is established, we want to start receiving events, so in the event handler for clicking the connect button, "Server_ConnectMI_Click(…)", we will add a call to a function – which we will implement – that will create the monitored item for the events we want to receive and add it to the subscription. So at the end of the try-block, in the just mentioned event handler, add:

```
                // ------------------ Start Receiving Events --------------------
                StartReceivingEvents();
                // -------------------------------------------------------------
```

Below this event handler, we will implement “StartReceivingEvents(…)”:

```
        /// <summary>
        /// Make a monitored item on a subscription for Events
        /// </summary>
        private void StartReceivingEvents()
        {
            // Create the subscription is not yet created:
            if (m_subscription == null)
            {
                m_subscription = new Subscription(m_session.DefaultSubscription);
                m_subscription.PublishingEnabled = true;
                m_subscription.PublishingInterval = 1000;
                m_subscription.KeepAliveCount = 10;
                m_subscription.LifetimeCount = 30;
                m_subscription.MaxNotificationsPerPublish = 1000;
                m_subscription.Priority = 100;

                m_session.AddSubscription(m_subscription);

                m_subscription.Create();
            }

            // Create the Event filter:
            FilterDefinition filter = new FilterDefinition();
            // Choose the Sever Object for AreaId: we want to receive events from the whole server
            // (choosing another Node will result in getting events only from that section of the Address Space)
            filter.AreaId = ObjectIds.Server;
            filter.Severity = EventSeverity.Min;
            // We want to receive all types of events (later new fields will be dynamically added):
            filter.EventTypes = new NodeId[] { ObjectTypeIds.BaseEventType };
            filter.SelectClauses = filter.ConstructSelectClauses(m_session, (NodeId[])filter.EventTypes);

            // Create a monitored item based on the current filter settings:
            MonitoredItem monitoredItem_TetrisEvent = filter.CreateMonitoredItem(m_session);
            // Set the callback: this event handler will process the event notification:
            monitoredItem_TetrisEvent.Notification +=
                new MonitoredItemNotificationEventHandler(MonitoredItem_EventNotification);

            // Add the monitored item to the subscription:
            m_subscription.AddItem(monitoredItem_TetrisEvent);
            m_subscription.ApplyChanges();
        }
```

So we are just left with the implementation of the event handler “MonitoredItem_EventNotification(…)”, which will handle the reception of a notification of our newly created monitored item for events. But first declare two new variables in the “Private Fields” region:

```
private List<string> m_eventFieldsToDisplay =
       new List<string> { BrowseNames.EventType, BrowseNames.SourceName, BrowseNames.Time, BrowseNames.Message };
private List<NodeId> m_eventTypes_Received = new List<NodeId> { ObjectTypeIds.BaseEventType };
```

m_eventsFieldsToDisplay is a list of strings representing the event fields we want display: we don’t necessarily want to display all BaseEventType event fields. Also, whenever new event fields are added to the monitored item’s filter, they are also added to this list. The m_eventTypes_Received list will contain all NodeId’s whose event fields were already added to the subscriptions, and will be used to check incoming EventType NodeId’s against. These two fields will be used in the event handler for our events:

```
/// <summary>
/// Handles the reception of a notification for an Event
/// </summary>
private void MonitoredItem_EventNotification(MonitoredItem monitoredItem, MonitoredItemNotificationEventArgs e)
{
   if (this.InvokeRequired)
   {
      this.BeginInvoke(new MonitoredItemNotificationEventHandler(MonitoredItem_EventNotification), monitoredItem,
      e);
      return;
   }

   try
   {
      // Get the event fields:
      EventFieldList eventFieldList = e.NotificationValue as EventFieldList;
      if (eventFieldList == null) return;

      // Get the EventTypeNodeId:
      NodeId eventTypeId = FormUtils.FindEventType(monitoredItem, eventFieldList);
      if (NodeId.IsNull(eventTypeId)) return;

      // Create string with the fields of the event:
      // first get the filter with its select clauses and then add the fields to the string.
      string eventString = "";
      EventFilter filter = monitoredItem.Filter as EventFilter;
      if (filter != null)
      {
         // Iterate through all the select clauses and respective received event fields:
         for (int ii = 0; ii < filter.SelectClauses.Count; ii++)
         {
            string fieldstring = "";
            // The SelectClause must contain a BrowsePath representing the field
            // AND: only display the field if it is in the list of fields we WANT to display
            // AND: also the event field must contain a value (must not be null):
            if (filter.SelectClauses[ii].BrowsePath.Count != 0
               && m_eventFieldsToDisplay.Contains(filter.SelectClauses[ii].BrowsePath[0].Name)
               && eventFieldList.EventFields[ii].Value != null)
            {
               // First set the field's name:
               fieldstring = filter.SelectClauses[ii].BrowsePath[0].Name;
               // Then add the value to the string representing the field:
               switch (fieldstring)
               {
                  case BrowseNames.EventType:
                     // Special case: we want the EventType NAME, not the actual NodeId, to be displayed:
                     fieldstring += ": " + (m_session.NodeCache.Find(eventTypeId)).DisplayName;
                     break;
                  default:
                     fieldstring += ": " + eventFieldList.EventFields[ii].Value.ToString();
                     break;
               }
            }else continue; // get going with the next event field = next iteration!

            // Add the fieldstring to the eventString that we'll later display:
            eventString += fieldstring + Environment.NewLine;
         }
         // After all the event fields added: Report Event to TetrisViewer!
         // (but not if it is not yet initialized!)
         if(m_TetrisViewer != null) m_TetrisViewer.AddEvent(eventString);
      }


      // If the received eventTYpeId is not the BaseEventType nor an evenType already received before and added to
      the list:
      // * Find the EventType node, get its specific fields,
      //   and add them to the SelectClauses of the subscription and also to the list of fields to display.
      // * At the end of the loop: also set the "evenTypeId" variable to its superType NodeId:
      //   the while loop will be iterated till an EventType node is encoutered
      //   that has already been added to the the list of received events.
      //   (this way also the fields of the eventTypes in between the received EventType
      //   and the BaseEventType in the EventType Hierarchy are added to the subscription)
      while (!m_eventTypes_Received.Contains(eventTypeId))
      {
         // Find the EventType node in the Address Space's Type Hierarchy ...
         Node eventTypeNode = m_session.NodeCache.FetchNode(eventTypeId);
         if (eventTypeNode == null) return; // (or generate an error if you want)
         // ... and get the eventType node's properties (= event fields)...
         IList<IReference> EventPropertyNodes = eventTypeNode.Find(ReferenceTypeIds.HasProperty, false);
         // ... and extract the event fields' names (as a List of strings) from this list of property nodes:
         List<string> newEventFields = new List<string>();
         foreach (IReference referenceToPropertyNode in EventPropertyNodes)
         {
            string eventFieldBrowseName =
            m_session.NodeCache.Find(referenceToPropertyNode.TargetId).BrowseName.Name;
            newEventFields.Add(eventFieldBrowseName);
         }

         // Add the new event fields to the SelectClauses of the current filter ...
         EventFilter eventFilter = monitoredItem.Filter as EventFilter;
         if (eventFilter != null)
         {
            foreach (string fieldName in newEventFields)
            {
               eventFilter.AddSelectClause(eventTypeId, new QualifiedName(fieldName, eventTypeId.NamespaceIndex));
            }
         }
         // ... and apply the changes:
         monitoredItem.Filter = eventFilter;
         m_subscription.ApplyChanges();

         // Add the new event fields to the List of fields to be displayed:
         m_eventFieldsToDisplay.AddRange(newEventFields);

         // Add the Event to our list of received EventTypes:
         m_eventTypes_Received.Add(eventTypeId);

         // Set the "eventTypeId" variable to its superType:
         // this way the while loop will be iterated untill all superTypes are also added
         // (the received event's superTypes can also have event fields relevant to their subtypes!)
         eventTypeId = (NodeId)eventTypeNode.GetSuperType(m_session.TypeTree);
      }
   }
   catch (Exception exception)
   {
      MessageBox.Show(exception.Message, this.Text, MessageBoxButtons.OK, MessageBoxIcon.Error);
   }
}
```

Most of the code is explained in the comments and the overall idea is described in the paragraphs above.

Run the server and connect to it with our new client. The first time a line is made in the TetrisGame on the server the client will receive an event, but only the BaseEvent fields will be sent. One of the BaseEvents fields is the EventType NodeId of the actual event. This is the TetrisLinesMadeEventType NodeId in our case and based on this information the client can adapt the subscription’s event monitored item and add that EventType’s specific fields. From the second event on – of the same type – the event’s specific fields are also sent by the server and received and processed by the client.

Further features could be added, for example the logging of the received events. In its most simple form, in this case in our code, this could mean adding a few lines and writing the generated eventstrings not only to the TetrisViewer, but also to a file on the hard disk. More complex logging system will be used in practice.

Again, note that UA provides the event functionality, but our choices – of for example subscribing only to certain events or making it dynamic, etc. – are not specifically articulated by the OPC UA specs.