DynamicDOM Tutorial

Draft Version 0.43.2, November 13, 2005

Abstract

This document describes the features, APIs, and use scenarios of the DynamicDOM (DDOM). The DDOM is an extended version of the W3C DOM (Document Object Model) that can dynamically update and validate its XML document based on rules defined in an XRules document.

Status of this document

This document is a work in progress. Software implementations and test tools of the XRules specification are available for download from www.xrules.org. These implementations are used as a proof of concept and are updated regularly as the XRules specification evolves.

If you like to participate in developing XRules or have feedback or questions, please refer to the XRules web site (http://www.xrules.org). All contributions are welcome and credit will be given where credit is due.

Table of Contents

DynamicDOM Tutorial 1

1 Introduction 3

  1.1 The DynamicDOM Implementation 3

2 DynamicDOM Tutorial 4

  2.1 Load the XML Document 5

  2.2 Attach the XRules Document 5

  2.3 Test the Validity of the XML Document 6

  2.4 The <calculate> rule 7

  2.5 The <validate> Rule 9

  2.6 The XRules Report 11

  2.7 XRules/DDOM Properties 12

    2.7.1 XRules Automatic Properties 12

    2.7.2 XRules User-Defined Properties 13

    2.7.3 DDOM Properties 13

  2.8 DDOM Events 15

  2.9 Where to Go from Here 16

1 Introduction

The DynamicDOM (or, DDOM) is an extension of the W3C Document Object Model with the added capability to enforce and execute XRules rules at run time. The DDOM adds intelligence (in the form of rules) to the DOM to expand its capabilities beyond parsing XML documents to processing and enforcing business rules.

The DynamicDOM is a dynamic XRules processor. You can download the software, detailed API documentation, and source code from http://www.xrules.org.

Just as C++ adds intelligence to C `struct` in the form of member methods, DDOM adds intelligence to the DOM in the form of rules. An example of a rule might be an expression such as `nodeX = nodeA + nodeB`, or a condition like `nodeX > nodeY`. Once rules are attached to the DDOM, they are executed and enforced automatically as appropriate. So, changing the value of a node can cause the values of other nodes in the DDOM to change automatically. And, if a node is set to a value that violates one of the attached rules, the error is detected and an appropriate error message is generated.

1.1 The DynamicDOM Implementation

This document describes the C# implementation in the DLL file: XRules.`DynamicDom.dll` version 0.43.2. The DynamicDOM is implemented in the `DXmlDocument` class, which is inherited from the .NET Framework class `XmlDocument`. Therefore, `DXmlDocument` has the same properties and methods as the original DOM object and adds a few more.

2 DynamicDOM Tutorial

The best way to explain the features and behavior of the DynamicDOM is by using an example. We'll use a sample XML Purchase Order and XRules document to demonstrate the use of the DynamicDOM to validate the Purchase Order, update calculated nodes automatically, and attach properties to various XML nodes.

Please refer to the *XRules Tutorial* document available at http://www.xrules.org for details about the syntax of the XRules language. This document assumes familiarity with XRules.

XML Document: PurchaseOrder.xml

```
<PurchaseOrder>
    <Item>
        <Name>Digital Camera</Name>
        <Quantity>1</Quantity>
        <UnitPrice>150</UnitPrice>
        <ItemTotal>150</ItemTotal>
    </Item>
    <Item>
        <Name>Battery</Name>
        <Quantity>4</Quantity>
        <UnitPrice>30</UnitPrice>
        <ItemTotal>120</ItemTotal>
    </Item>
    <SubTotal>270</SubTotal>
    <Tax>21.6</Tax>
    <GrandTotal>291.6</GrandTotal>
</PurchaseOrder>
```

XRules Document: PurcahseOrder.xr

```
<xr:rules xmlns:xr="http://www.xrules.org/2003/11">

   <xr:ruleset context="/PurchaseOrder">
      <xr:calculate target="SubTotal">
         <xr:value>sum(Item/ItemTotal)</xr:value>
      </xr:calculate>
      <xr:calculate target="Tax">
         <xr:value>SubTotal * 0.08</xr:value>
      </xr:calculate>
      <xr:calculate target="GrandTotal">
         <xr:value>SubTotal + Tax</xr:value>
      </xr:calculate>
      <xr:bind target="GrandTotal">
         <xr:property name="InEuros" dvalue="GrandTotal * 0.8" />
      </xr:bind>
   </xr:ruleset>

   <xr:ruleset context="/PurchaseOrder/Item">
      <xr:bind target="Quantity" min="1" max="100"
```

```
              errorMessage.min="Quantity must be positive." />
       <xr:validate test="UnitPrice &gt; 0"
                 errorMessage="Unit Price cannot be negative."/>
       <xr:calculate target="ItemTotal">
          <xr:value>UnitPrice * Quantity</xr:value>
       </xr:calculate>
    </xr:ruleset>

</xr:rules>
```

2.1 Load the XML Document

The easiest way to load an XML document is to create a `DXmlDocument` object and then use the `Load()` method as shown below. Don't forget to add a reference to `XRules.DynamicDom.dll` in your project, and to import the `XRules` namespace in which the `DXmlDocument` class is defined.

C#:

```
using XRules;

// Open the XML document.
DXmlDocument dxml = new DXmlDocument();
dxml.Load("PurchaseOrder.xml");
```

2.2 Attach the XRules Document

Just like loading an XML document, loading an XRules document requires creating an `XRulesDocument` object, then using the `Load()` method as shown below:

C#:

```
// Open the XRules document.
XRulesDocument xrules = new XRulesDocument();
xrules.Load("PurchaseOrder.xr");
```

Then, we attach the XRules document to the `DXmlDocument` object using the `Add()` method of the `XRulesDocuments` collection. The `XRulesDocuments` collection provides methods such as `Add()` and `Remove()` to allow you to easily attach and detach XRules documents.

C#:

```
// Attach the XRules document to the DDOM.
dxml.XRulesDocuments.Add(xrules);
```

Internally, attaching an XRules document to the DDOM is performed in two phases:

1. **Binding Phase**: in this phase the rules of the XRules document are compiled and attached to the appropriate nodes in the XML document.
2. **Application Phase**: in this phase the rules are executed against the XML document. Based on the result of applying the rules some nodes in the XML document might have new values, and some errors might be generated if the XML document doesn't satisfy all the rules. The errors are accessible through the `Errors` collection of the `DXmlDocument` as we'll see later.

2.3 Test the Validity of the XML Document

Once an XRules document is attached to the DDOM, validation and rule enforcement is already in place. To check if any errors are generated from the validation, we'll check the `Errors` collection of the `DXmlDocument` class.

```
// Check the validty of the XML document.
if (dxml.Errors.Count == 0)
{
   Console.WriteLine("XML document is valid.");
}
else
{
   Console.WriteLine("Errors found in the XML document.");
}
```

And, this is how the code looks like so far:

```
using System;
using XRules;
using System.Xml;
using System.IO;

namespace ConsoleTestCS
{
   class ConsoleTest
   {
      /// <summary>
      /// The main entry point for the application.
      /// </summary>
      [STAThread]
      static void Main(string[] args)
      {
         // Open the XML document.
         DXmlDocument dxml = new DXmlDocument();
         dxml.Load("PurchaseOrder.xml");
```

```
          // Open the XRules document.
          XRulesDocument xrules = new XRulesDocument();
          xrules.Load("PurchaseOrder.xr");

          // Attach the XRules document to the DDOM.
          dxml.XRulesDocuments.Add(xrules);

          // Check the validty of the XML document.
          if (dxml.Errors.Count == 0)
          {
             Console.WriteLine("XML document is valid.");
          }
          else
          {
             Console.WriteLine("Errors found in the XML document.");
          }
       }
    }
}
```

Running this code generates the following simple output, which tells us that our XML document is valid:

```
C:\ConsoleTestCS>ConsoleTestCs.exe
XML document is valid.
```

2.4 The <calculate> rule

Now, let's demonstrate the use of the `<calculate>` rule. Our XRules document contains four of these rules to determine how the `ItemTotal`, `SubTotal`, `Tax`, and `GrandTotal` nodes are calculated. Let's change the quantity of the second item in the purchase order from 4 to 2 and see how this affects the values of the calculated nodes. We'll use the regular way of changing the `innerText` of the `Quantity` node just like we'd do it when using the standard `XmlDocument` class:

C#:

```
// Print XML document.
Console.WriteLine("\nXML Before Updating Quantity value:");
WriteXml(dxml);

// Update the 'Quanity' node of the second item.
dxml.SelectSingleNode("/PurchaseOrder/Item[2]/Quantity").InnerText="2";

// Print XML document.
Console.WriteLine("XML After Updating Quantity value:");
```

```
WriteXml(dxml);
```

The `WriteXml()` function is a helper function that writes the XML document to the console:

```
static void WriteXml(XmlDocument xml)
{
   StringWriter writer = new StringWriter();
   xml.Save(writer);
   Console.WriteLine("\n" + writer.ToString() + "\n");
}
```

And this is the output of running this code:

```
XML Before Updating Quantity value:

<?xml version="1.0" encoding="utf-16"?>
<PurchaseOrder>
  <Item>
    <Name>Digital Camera</Name>
    <Quantity>1</Quantity>
    <UnitPrice>150</UnitPrice>
    <ItemTotal>150</ItemTotal>
  </Item>
  <Item>
    <Name>Battery</Name>
    <Quantity>4</Quantity>
    <UnitPrice>30</UnitPrice>
    <ItemTotal>120</ItemTotal>
  </Item>
  <SubTotal>270</SubTotal>
  <Tax>21.6</Tax>
  <GrandTotal>291.6</GrandTotal>
</PurchaseOrder>


XML After Updating Quantity value:

<?xml version="1.0" encoding="utf-16"?>
<PurchaseOrder>
  <Item>
    <Name>Digital Camera</Name>
    <Quantity>1</Quantity>
    <UnitPrice>150</UnitPrice>
    <ItemTotal>150</ItemTotal>
  </Item>
  <Item>
    <Name>Battery</Name>
    <Quantity>2</Quantity>
    <UnitPrice>30</UnitPrice>
    <ItemTotal>60</ItemTotal>
  </Item>
  <SubTotal>210</SubTotal>
  <Tax>16.8</Tax>
  <GrandTotal>226.8</GrandTotal>
</PurchaseOrder>
```

As the example shows, changing the value of the `Quantity` node caused the `ItemTotal` value to update, which, in turn, caused the `SubTotal`, `Tax`, and `GrandTotal` nodes to update.

2.5 The <validate> Rule

Now let's demonstrate the use of the `<validate>` rule. We'll start by setting the value of the `UnitPrice` node of the first item to -150, which is an invalid value because of the following rule in our XRules document:

```
       <xr:validate test="UnitPrice &gt; 0" />
```

The code we'll use is just like the code you'd use to change the value of a node in the regular `XmlDocument` class except for the use of the `Errors` collection to check for validation errors.

```
// Set UnitPrice to -150.
Console.WriteLine("Setting UnitPrice to -150:");
dxml.SelectSingleNode("/PurchaseOrder/Item[1]/UnitPrice").InnerText = "-150";

// Print XML document.
Console.WriteLine("XML after setting UnitPrice to -150:\n");
WriteXml(dxml);

// Print errors if any.
if (dxml.Errors.Count > 0)
{
   Console.WriteLine("Errors found in XML document:");
   foreach (RuleError error in dxml.Errors)
   {
      Console.WriteLine("* " + error.ErrorMessage);
      foreach (XmlNode node in error.RuleInstance.ParticipatingParticles)
      {
         Console.WriteLine("  - node: " + node.Name);
      }
   }
}
else
{
   Console.WriteLine("No Errors in the XML document.");
}
```

Notice that we're checking `Errors.Count` to see if there are validation errors. And if so, we iterate through the errors and print their details to the console. The `Errors` collection contains error objects of the type `RuleError`, which has two properties that we're interested in:

- **`ErrorMessage`**: This is either the error message specified in the rule using the `errorMessage` attribute (as is the case in this example), or an automatically generated error message if the `errorMessage` attribute is missing.
- **`RuleInstance`**: This property returns an object of type `RuleInstance` which represents an instance of the rule that generated the error. I won't describe this class in detail here except for what we need for our example, i.e., the `ParticipatingParticles` collection. This collection contains the list of the particles (elements, attributes, or properties) that participate in this rule instance. And, since we know that we're not using DynamicDOM properties in this example yet, we can safely assume that the collection contains objects of type `XmlNode`, and we loop through those nodes and write their names.

```
        foreach (XmlNode node in error.RuleInstance.ParticipatingParticles)
        {
            Console.WriteLine("  - node: " + node.Name);
        }
```

Running the code above produces the following output:

```
Setting UnitPrice to -150:

XML after setting UnitPrice to -150:

<?xml version="1.0" encoding="utf-16"?>
<PurchaseOrder>
  <Item>
    <Name>Digital Camera</Name>
    <Quantity>1</Quantity>
    <UnitPrice>-150</UnitPrice>
    <ItemTotal>-150</ItemTotal>
  </Item>
  <Item>
    <Name>Battery</Name>
    <Quantity>2</Quantity>
    <UnitPrice>30</UnitPrice>
    <ItemTotal>60</ItemTotal>
  </Item>
  <SubTotal>-90</SubTotal>
  <Tax>-7.2</Tax>
  <GrandTotal>-97.2</GrandTotal>
</PurchaseOrder>

Errors found in XML document:
* Unit Price cannot be negative.
  - node: UnitPrice
```

2.6 The XRules Report

Another way to see the result of validating the XML document is to retrieve the XRules report, which contains an XML representation of the validation errors. To do that, we'll use the `GenerateReportXml()` method of the `DXmlDocument` class. This method returns an `XmlDocument` object that contains the report.

```
// Print the XRules report.
Console.WriteLine("\nXRules Report:");
WriteXml(dxml.GenerateReportXml());
```

This code generates the following output:

```
XRules Report:

<?xml version="1.0" encoding="utf-16"?>
<xrr:report xmlns:xr="http://www.xrules.org/2003/11"
xmlns:xrr="http://www.xrules.org/2003/11/report">
  <xrr:errors>
    <xrr:error context="/PurchaseOrder/Item">
      <xrr:message>Unit Price cannot be negative.</xrr:message>
      <xrr:node path="/PurchaseOrder/Item[1]/UnitPrice[1]" />
    </xrr:error>
  </xrr:errors>
</xrr:report>
```

As you can see, we got the same information that we got from inspecting the Errors collection, but this time in XML format. This is mostly useful for applications that use XSLT to transform the error details into, say, an HTML report.

The Errors collection and the XRules report are dynamic entities. They update automatically with every change that happens in the XML document. So, as we change values of nodes, new errors might show up in the Errors collection and others might disappear. For now, let's set the value of `UnitPrice` to 150 again to bring our XML to a valid state:

```
// Reset UnitPrice to 150 and print the XRules report.
Console.WriteLine("Reset UnitPrice to 150.");
dxml.SelectSingleNode("/PurchaseOrder/Item[1]/UnitPrice").InnerText = "150";
Console.WriteLine("\nXRules Report:");
WriteXml(dxml.GenerateReportXml());
```

This code will print an empty XRules report, as shown below:

```
Reset UnitPrice to 150.

XRules Report:

<?xml version="1.0" encoding="utf-16"?>
<xrr:report xmlns:xr="http://www.xrules.org/2003/11"
xmlns:xrr="http://www.xrules.org/2003/11/report">
  <xrr:errors />
</xrr:report>
```

2.7 XRules/DDOM Properties

XRules and DDOM allow you to add properties to elements and attributes in the XML Infoset at run time. These properties are stored in the DDOM object model and can be accessed using the `Properties` collection on all element and attribute objects.

Generally, we can classify properties into three categories:

1. **XRules Automatic Properties**: Created automatically from the execution of certain XRules rules.
2. **XRules User-Defined Properties:** Created using the `<xr:property>` rule.
3. **DDOM Properties:** Attached to XML nodes using the `Properties` collection.

2.7.1 XRules Automatic Properties

These are properties generated automatically by the DDOM when it applies XRules rules such as `<bind>`. These properties provide information about the target node such as its data type, minimum value, maximum value, minimum length …etc. In our XRules document, we have this `<bind>` rule:

```
<xr:bind target="Quantity" min="1" max="100"
         errorMessage.min="Quantity must be positive." />
```

This rule, when applied, generates the two properties on the `Quantity` node: `xr:min` and `xr:max`, which have the values 1 and 100 respectively. And, we can read the values of these properties in code as follows:

```
// Print automatic properties of the 'Quantity' node.
Console.WriteLine("Properties of the 'Quantity' node");
IDXmlNode quantity = (IDXmlNode)dxml.SelectSingleNode("//Item[1]/Quantity");
```

```
Console.WriteLine("xr:min = " + quantity.Properties["xr:min"].Value);
Console.WriteLine("xr:max = " + quantity.Properties["xr:max"].Value);
```

This code generates this output:

```
Properties of the 'Quantity' node
xr:min = 1
xr:max = 100
```

Notice that we used the `IDXmlNode` interface instead of the regular `XmlNode` class. This interface allows access to the new methods and properties that the DDOM provides. All `XmlElement` and `XmlAttribute` objects of the `DXmlDocument` class implement the `IDXmlNode` interface.

Also, keep in mind that all XRules automatic properties has the "`xr:`" prefix and that all of them are read-only.

2.7.2 XRules User-Defined Properties

These properties are created using the `<property>` child rule of the `<bind>` rule and attached to selected elements or attributes. In our XRules document, we have the following property:

```
<xr:bind target="GrandTotal">
   <xr:property name="InEuros" dvalue="GrandTotal * 0.8" />
</xr:bind>
```

The rule above generates a property named `InEuros`, to hold the value of `GrandTotal` converted to Euros, and attaches it to the `GrandTotal` element in our XML purchase order. Furthermore, it provides the XPath expression used to calculate the value of this property in the `dvalue` attribute.

2.7.3 DDOM Properties

In addition to using the `<property>` rule, we can attach properties to any element or attribute using the `Properties` collection of the `IDXmlNode` interface. So, let's add another property to the `GrandTotal` element and call it `InCanadian`. We'll use this property to store the value of `GrandTotal` in Canadian Dollars.

```
// Create InCanadian property on the GrandTotal element.
XmlNode grandTotal = dxml.SelectSingleNode("/PurchaseOrder/GrandTotal");
IDXmlNode dGrandTotal = (IDXmlNode)grandTotal;
Double inCanadian = Double.Parse(grandTotal.InnerText) * 1.2;
dGrandTotal.Properties["InCanadian"].Value = inCanadian;
```

As you see in the code above, creating a new property (the last line) is simply a matter of setting it to a value. But don't forget to cast your node to the `IDXmlNode` interface to get access to the `Properties` collection.

Now, let's print the values of all properties attached to the `GrandTotal` node. We should expect to find two: the one we just created; and the one created by the `<property>` rule in the previous section. This time we'll iterate through the Properties collection and print everything we find:

```
// Print all properties on the GrandTotal node.
Console.WriteLine("\nProperties of the 'GrandTotal' node");
foreach (DXmlProperty property in dGrandTotal.Properties)
{
   Console.Write(property.Name + " = " + property.Value);
   if (property.IsCalculated == true)
      Console.WriteLine("\t(Calculated)");
   else
      Console.WriteLine("\t(Editable)");
}
```

This code generates this output:

```
Properties of the 'GrandTotal' node
InCanadian = 272.16     (Editable)
InEuros = 181.44        (Calculated)
```

Now let's examine the code above. You'll notice that all properties belong to the `DXmlProperty` class. We're using the following properties of this class:

- **Name**: The name of the property.
- **Value**: The value of the property.
- **IsCalculated**: A Boolean value that indicates weather a property is a calculated one. A property is considered a calculated property if it's an XRules automatic property, or generated by the `<property>` rule. Those properties are read-only because their values are determined by the DDOM. On the other hand, properties created using the `Properties` collection are editable and their `IsCalculated` value is False.

2.8 DDOM Events

The `DXmlDocument` class raises two new events that the regular `XmlDocument` class doesn't: `DNodeChanged` and `DPropertyChanged`. As the name implies, these two events fire after a node or a property has changed its value, respectively.

So let's see a code sample that captures these events. We'll attach two event handlers to our `DXmlDoncument` object.

```
// Attach event handlers.
dxml.DNodeChanged +=
      new DXmlNodeChangedEventHandler(dxml_DNodeChanged);
dxml.DPropertyChanged +=
      new DXmlPropertyChangedEventHandler(dxml_DPropertyChanged);
```

And here are our two event handler functions. We'll just print out a message and the name of the node or property that caused the event to fire:

```
private static void dxml_DNodeChanged(object sender,
                                      DXmlNodeChangedEventArgs e)
{
   Console.WriteLine(" - DNodeChanged event fired. Node = " + e.Node.Name);
}


private static void dxml_DPropertyChanged(object sender,
                                          DXmlPropertyChangedEventArgs e)
{
   Console.WriteLine(" - DPropertyChanged event fired. Property = " +
                     e.Property.Name);
}
```

And now let's change the value of a node in our XML purchase order to see what events fire and in what order:

```
// Update the 'Quanity' node of the second item.
Console.WriteLine("\nUpdate Quanity of the second item.");
dxml.SelectSingleNode("/PurchaseOrder/Item[2]/Quantity").InnerText="5";
Console.WriteLine("Finished updating Quanity.");
```

We're all set, now let's run the code and see the output:

```

Update Quanity of the second item.
 - DNodeChanged event fired. Node = Quantity
 - DNodeChanged event fired. Node = ItemTotal
 - DNodeChanged event fired. Node = SubTotal
 - DNodeChanged event fired. Node = Tax
 - DNodeChanged event fired. Node = GrandTotal
 - DPropertyChanged event fired. Property = InEuros
Finished updating Quanity.
```

The output above shows that changing the `Quantity` element caused the `ItemTotal` element to change, which in turn caused the `SubTotal` to change and so on. Also we see that the value of the `InEuros` property changed after the `GrandTotal` value was updated. The DDOM has determined the dependencies among the nodes and automatically recalculated all nodes that are affected by the change.

An interesting point to note here is that, unlike the `InEuros` property, the `InCanadian` property that we attached earlier to the `GrandTotal` node didn't change. That's because it's not defined in the XRules document as a calculated property, and the DDOM has no way of determining when to recalculate it and how. Properties that we create in our code will have to be updated by our code.

Another point to keep in mind is that the regular `NodeChanged` and `NodeChanging` events of the `XmlDocument` class are disabled in the DDOM. That's because they work in a different manner and conflict with the inner workings of the `DXmlDocument` class.

2.9 Where to Go from Here

I hope this document has provided a good starting point for using XRules and DynamicDOM. You can find the source code of the example we used in this tutorial in the download package available at http://www.xrules.org along with other tools, samples, and utilities. These other tools are described in detail in the XRules Tutorial available from the same web site.