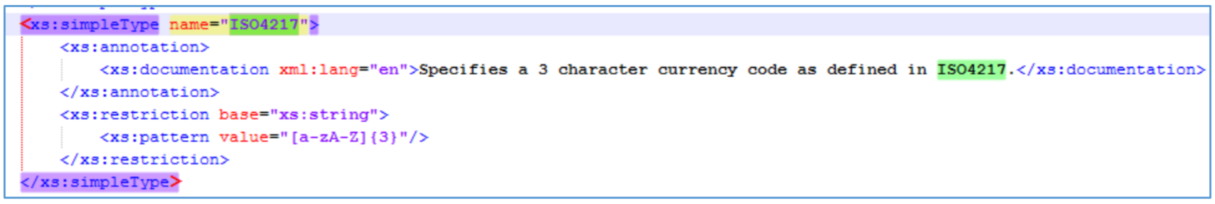
Strong Typing

To explore the topic of Strong Typing, let's look at two `SimpleTypes` from the Open Travel schema `OTA_SimpleTypes.xsd`.

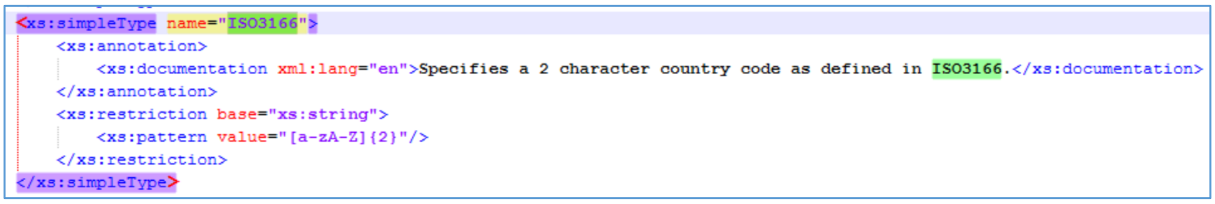
In Figure 1 we see the schema definition for an `ISO4217`, it's a restriction of a 'string' base type using a business-relevant pattern which provide the *very specific* structure of the entity.



```
<xs:simpleType name="ISO4217">
    <xs:annotation>
        <xs:documentation xml:lang="en">Specifies a 3 character currency code as defined in ISO4217.</xs:documentation>
    </xs:annotation>
    <xs:restriction base="xs:string">
        <xs:pattern value="[a-zA-Z]{3}"/>
    </xs:restriction>
</xs:simpleType>
```

Figure 1 ISO4217 Type Definition

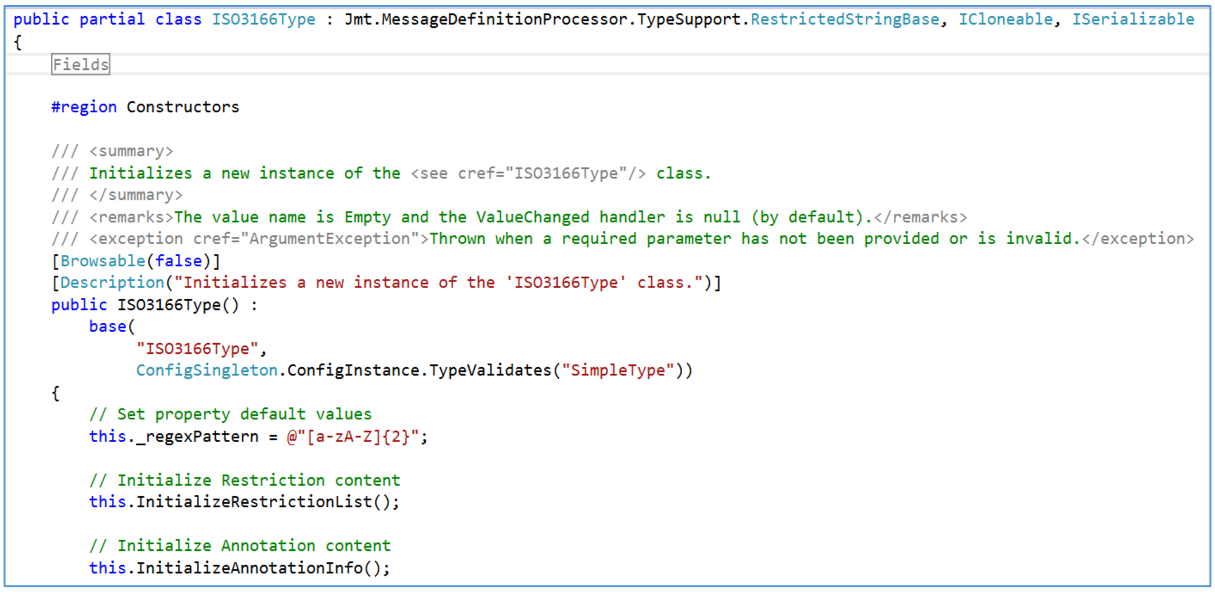
In Figure 2 we see a superficially similar type definition, that of `ISO3166`, where again it's a restriction of a 'string' base type with a different pattern to that of the `ISO4217`.



```
<xs:simpleType name="ISO3166">
    <xs:annotation>
        <xs:documentation xml:lang="en">Specifies a 2 character country code as defined in ISO3166.</xs:documentation>
    </xs:annotation>
    <xs:restriction base="xs:string">
        <xs:pattern value="[a-zA-Z]{2}"/>
    </xs:restriction>
</xs:simpleType>
```

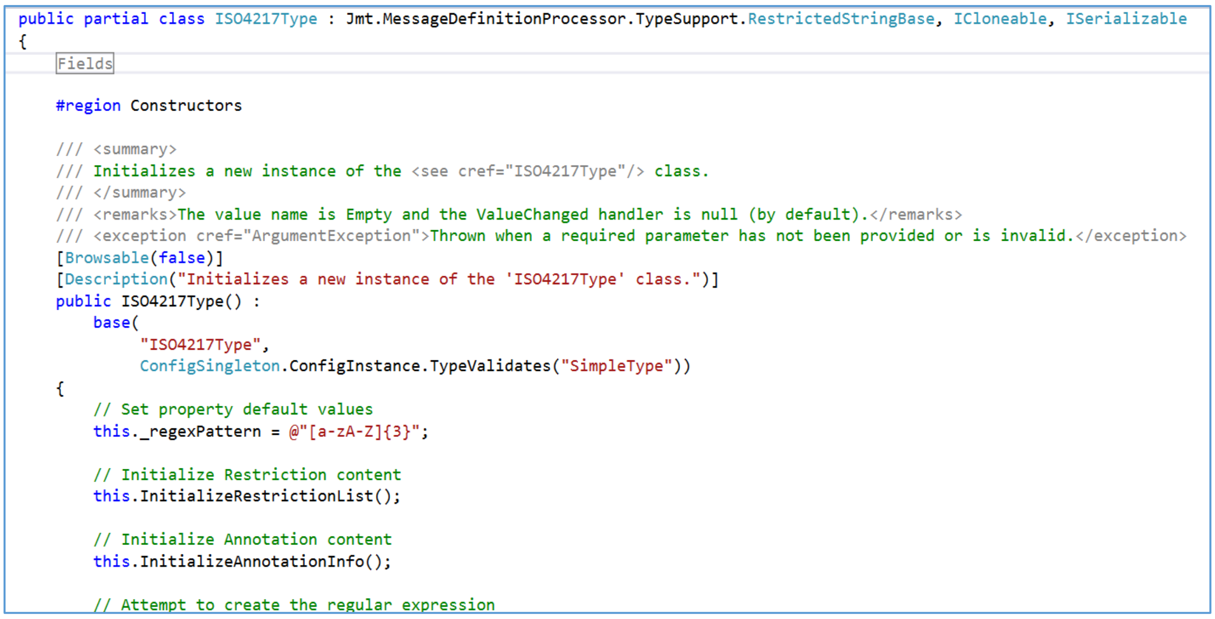
Figure 2 ISO3166 Type Definition

Typically these entities would be referenced as simple `string` types thus removing completely the important business-relevant information held in the `xs:pattern` data. Now let's look at their corresponding representations in the JMT library, fragments of the classes representing the two types are shown in the following figures.



```
public partial class ISO3166Type : Jmt.MessageDefinitionProcessor.TypeSupport.RestrictedStringBase, ICloneable, ISerializable
{
    Fields

    #region Constructors

    /// <summary>
    /// Initializes a new instance of the <see cref="ISO3166Type"/> class.
    /// </summary>
    /// <remarks>The value name is Empty and the ValueChanged handler is null (by default).</remarks>
    /// <exception cref="ArgumentException">Thrown when a required parameter has not been provided or is invalid.</exception>
    [Browsable(false)]
    [Description("Initializes a new instance of the 'ISO3166Type' class.")]
    public ISO3166Type() :
        base(
             "ISO3166Type",
             ConfigSingleton.ConfigInstance.TypeValidates("SimpleType"))
    {
        // Set property default values
        this._regexPattern = @"[a-zA-Z]{2}";

        // Initialize Restriction content
        this.InitializeRestrictionList();

        // Initialize Annotation content
        this.InitializeAnnotationInfo();
```

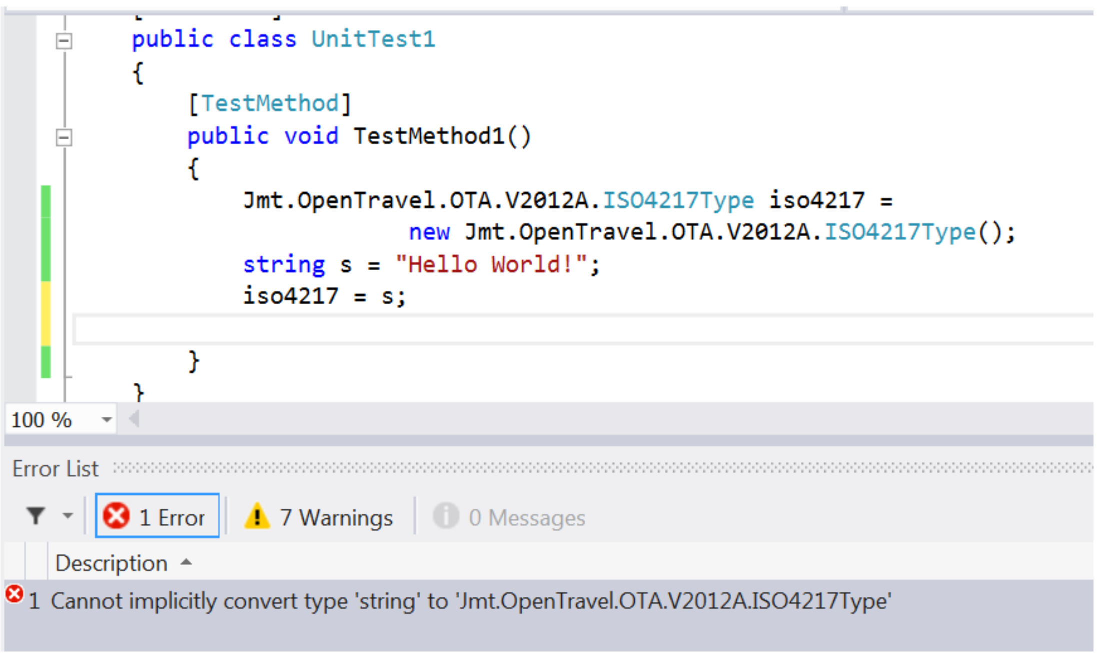
Figure 3 JMT Type Representation ISO3166



```
public partial class ISO4217Type : Jmt.MessageDefinitionProcessor.TypeSupport.RestrictedStringBase, ICloneable, ISerializable
{
    Fields

    #region Constructors

    /// <summary>
    /// Initializes a new instance of the <see cref="ISO4217Type"/> class.
    /// </summary>
    /// <remarks>The value name is Empty and the ValueChanged handler is null (by default).</remarks>
    /// <exception cref="ArgumentException">Thrown when a required parameter has not been provided or is invalid.</exception>
    [Browsable(false)]
    [Description("Initializes a new instance of the 'ISO4217Type' class.")]
    public ISO4217Type() :
        base(
             "ISO4217Type",
             ConfigSingleton.ConfigInstance.TypeValidates("SimpleType"))
    {
        // Set property default values
        this._regexPattern = @"[a-zA-Z]{3}";

        // Initialize Restriction content
        this.InitializeRestrictionList();

        // Initialize Annotation content
        this.InitializeAnnotationInfo();

        // Attempt to create the regular expression
```

Figure 4 JMT Type Representation ISO4217

These class definition, *of themselves*, represent strong type definitions that can be used in a general programming context. For example, taking ISO4217Type, we might write statements as shown in Figure 5 and try and set the value of an ISO4217Type object directly to the value of a simple string and attempt to compile:



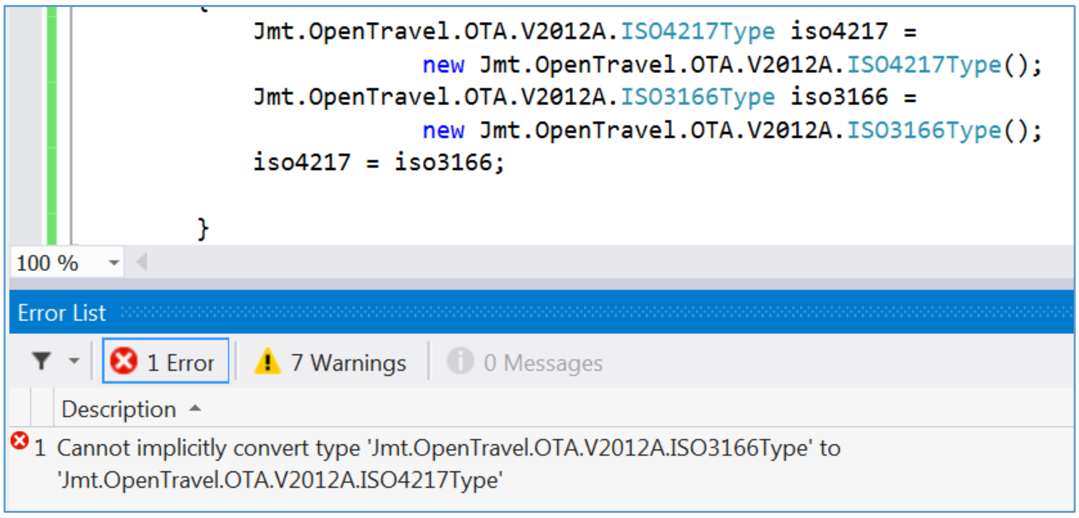
```
public class UnitTest1
{
    [TestMethod]
    public void TestMethod1()
    {
        Jmt.OpenTravel.OTA.V2012A.ISO4217Type iso4217 =
                  new Jmt.OpenTravel.OTA.V2012A.ISO4217Type();
        string s = "Hello World!";
        iso4217 = s;

    }
}
```

```
1 Cannot implicitly convert type 'string' to 'Jmt.OpenTravel.OTA.V2012A.ISO4217Type'
```

Figure 5 Compile-Time Type Violation (Variant 1)

The strong typing of the ISO4217Type is now clearly demonstrated by the error displayed in the Error List of Visual Studio.

In addition, we could also attempt to set the type value to each other, as show in Figure 6 below:



```
            Jmt.OpenTravel.OTA.V2012A.ISO4217Type iso4217 =
                        new Jmt.OpenTravel.OTA.V2012A.ISO4217Type();
            Jmt.OpenTravel.OTA.V2012A.ISO3166Type iso3166 =
                        new Jmt.OpenTravel.OTA.V2012A.ISO3166Type();
            iso4217 = iso3166;

        }
```

100 %

Error List

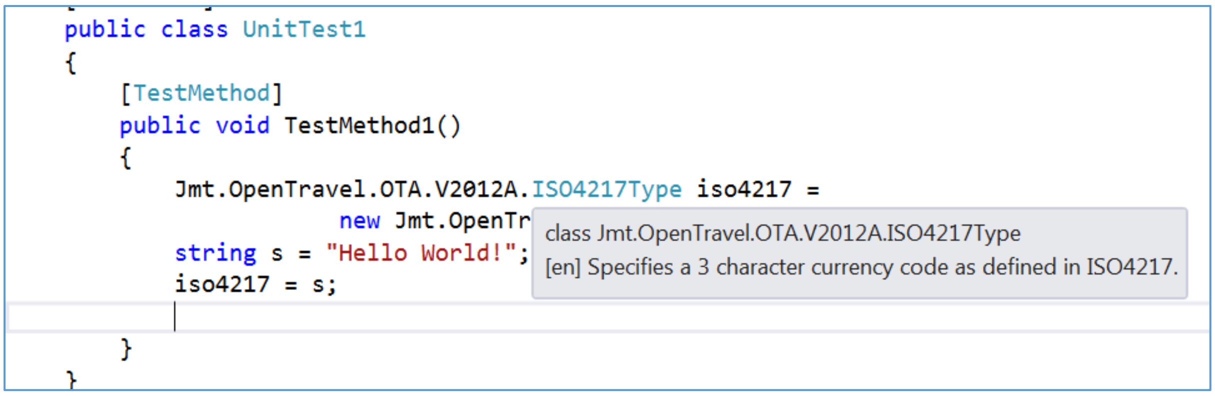
1 Error | 7 Warnings | 0 Messages

Description

1 Cannot implicitly convert type 'Jmt.OpenTravel.OTA.V2012A.ISO3166Type' to 'Jmt.OpenTravel.OTA.V2012A.ISO4217Type'

Figure 6 Compile-Time Type Violation (Variant 2)

It should be also noted that as the *original schema had specified documentation* then the generated type preserves this vital information for display to the developer when writing code and hovering over the type name, as we can see in Figure 7 below:



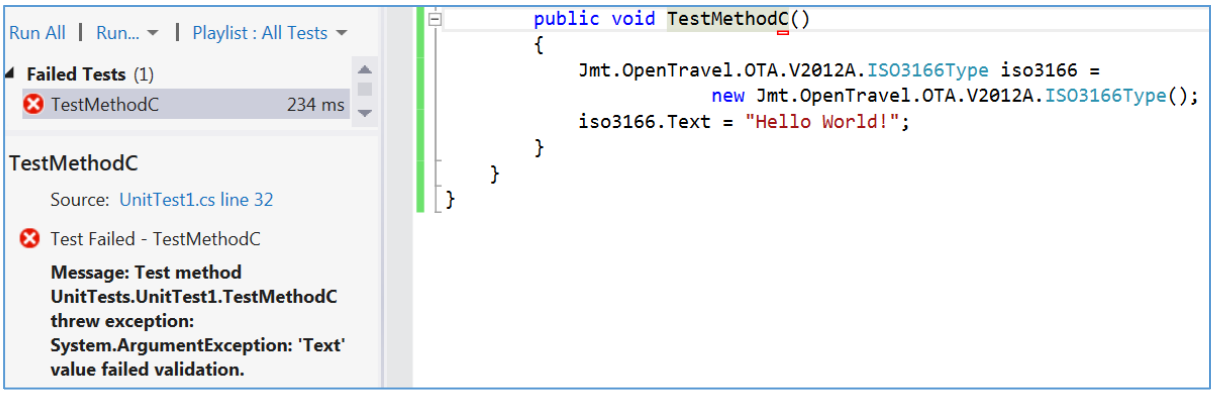
```
    public class UnitTest1
    {
        [TestMethod]
        public void TestMethod1()
        {
            Jmt.OpenTravel.OTA.V2012A.ISO4217Type iso4217 =
                        new Jmt.OpenTr
            string s = "Hello World!";
            iso4217 = s;

        }
    }
```

class Jmt.OpenTravel.OTA.V2012A.ISO4217Type
[en] Specifies a 3 character currency code as defined in ISO4217.

Figure 7 Schema Documentation for the Developer

It's one thing to have such a key 'helper' as the basic strong typing, to ensure correct programming constructs at compile-time, but there are a range of other operational characteristics that have to be thought about when crafting real types.

If we think about the definitions of the types in the schema, it is expressed as a restriction, specifying precisely the allowed form of values of an `ISO3166 or ISO4217`. As a question of type design, we need to provide a consistent means to enforce this restriction. Within the JMT library, types have an event defined that fires when an attempt is made to set an 'invalid' value, compared, in these specific cases, with the defining pattern.



Run All | Run... | Playlist : All Tests

Failed Tests (1)
TestMethodC 234 ms

TestMethodC
Source: UnitTest1.cs line 32

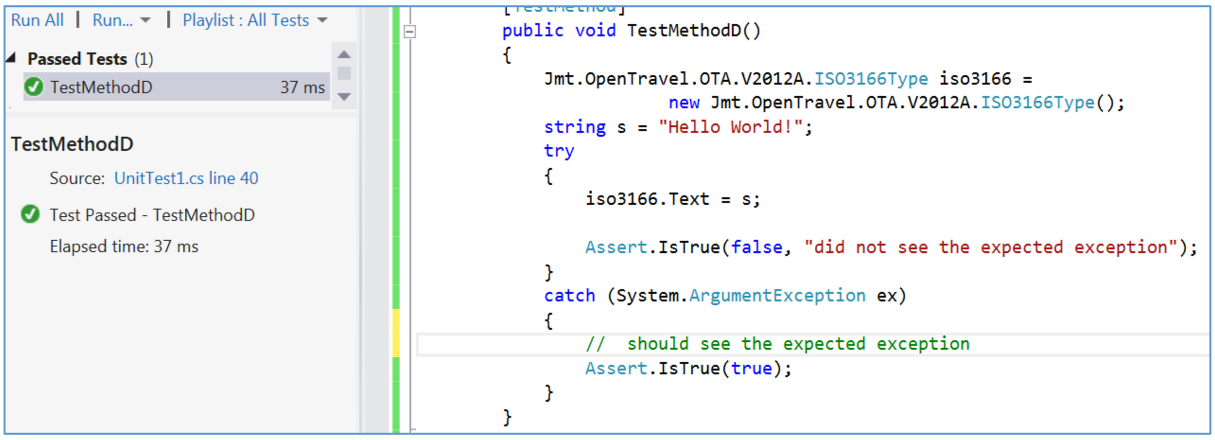
Test Failed - TestMethodC
Message: Test method UnitTests.UnitTest1.TestMethodC threw exception: System.ArgumentException: 'Text' value failed validation.

```
        public void TestMethodC()
        {
            Jmt.OpenTravel.OTA.V2012A.ISO3166Type iso3166 =
                      new Jmt.OpenTravel.OTA.V2012A.ISO3166Type();
            iso3166.Text = "Hello World!";
        }
    }
}
```

Figure 8 Invalid Value Exception

In Figure 8, an attempt is being made to set the ‘Text’ property, which holds the ‘value’ of the object, of the `ISO3166Type` object to (an obviously) incorrect value. Without taking any steps to guard against this sort of action, the left hand panel of Visual Studio informs us that an exception has been thrown of type `System.ArgumentException`, this being thrown by the strong type *itself* in this circumstance.

Of course, the business application developer would not write code that simply threw an exception when the value of a typed object is being set to an ‘invalid’ value. The normal and more useful approach would be to wrap the value setting statements in a try/catch block, as exemplified in the code fragments of Figure 9 and Figure 10, for ‘bad’ and ‘good’ values:



Run All | Run... | Playlist : All Tests

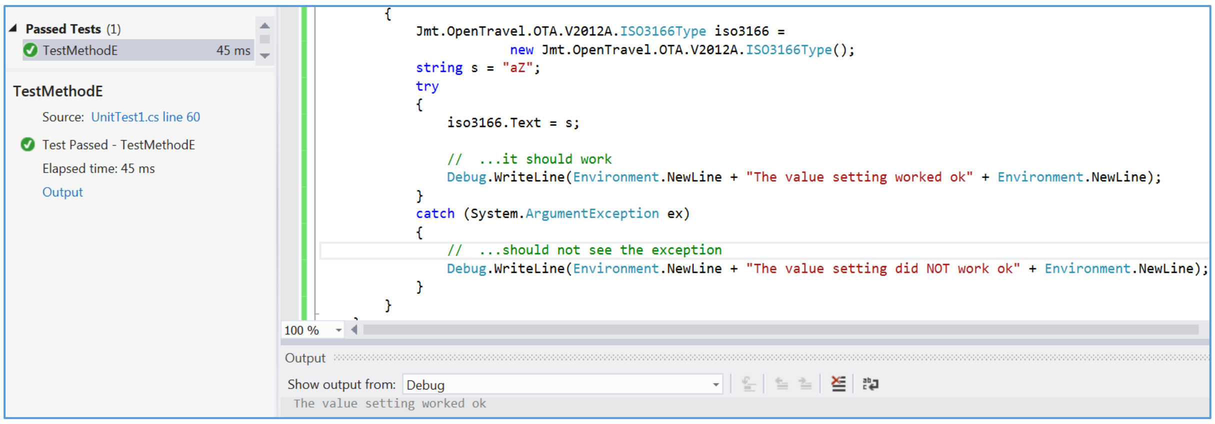
Passed Tests (1)
TestMethodD 37 ms

TestMethodD
Source: UnitTest1.cs line 40

Test Passed - TestMethodD
Elapsed time: 37 ms

```
        public void TestMethodD()
        {
            Jmt.OpenTravel.OTA.V2012A.ISO3166Type iso3166 =
                      new Jmt.OpenTravel.OTA.V2012A.ISO3166Type();
            string s = "Hello World!";
            try
            {
                iso3166.Text = s;

                Assert.IsTrue(false, "did not see the expected exception");
            }
            catch (System.ArgumentException ex)
            {
                //  should see the expected exception
                Assert.IsTrue(true);
            }
        }
```

Figure 9 Try/Catch Block – Basic (Invalid Value)



Passed Tests (1)
TestMethodE 45 ms

TestMethodE
Source: UnitTest1.cs line 60
Test Passed - TestMethodE
Elapsed time: 45 ms
Output

```
        {
            Jmt.OpenTravel.OTA.V2012A.ISO3166Type iso3166 =
                        new Jmt.OpenTravel.OTA.V2012A.ISO3166Type();
            string s = "aZ";
            try
            {
                iso3166.Text = s;

                //  ...it should work
                Debug.WriteLine(Environment.NewLine + "The value setting worked ok" + Environment.NewLine);
            }
            catch (System.ArgumentException ex)
            {
                //  ...should not see the exception
                Debug.WriteLine(Environment.NewLine + "The value setting did NOT work ok" + Environment.NewLine);
            }
        }
```

Output
Show output from: Debug

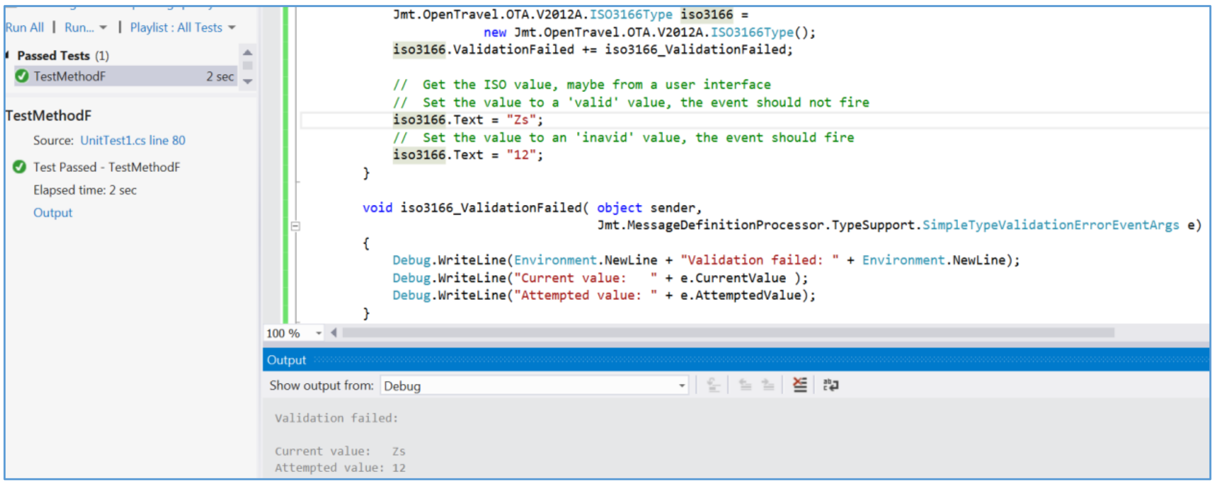
```
The value setting worked ok
```

Figure 10 Try/Catch Block – Improved (Valid Value)

When using JMT types it may not be sufficient to just catch the ‘validation failed’ exception as shown above. For such themes as “separation of concerns” and architectural patterns such as Model/View-Model/Model, it is highly desirable to connect components using events.

This event model is also catered for in the JMT types, so, for instance, as shown below, the developer can subscribe to a specific ‘validation failed’ event and take appropriate action in a defined handler. This handler could be one that signals to a user that an invalid IBAN has been selected or entered or perhaps composed a reply/response message that is sent to a caller of the component where such validation is performed.

The code fragment of Figure 11 shows the approach:



Run All | Run... | Playlist : All Tests

Passed Tests (1)
TestMethodF 2 sec

TestMethodF
Source: UnitTest1.cs line 80
Test Passed - TestMethodF
Elapsed time: 2 sec
Output

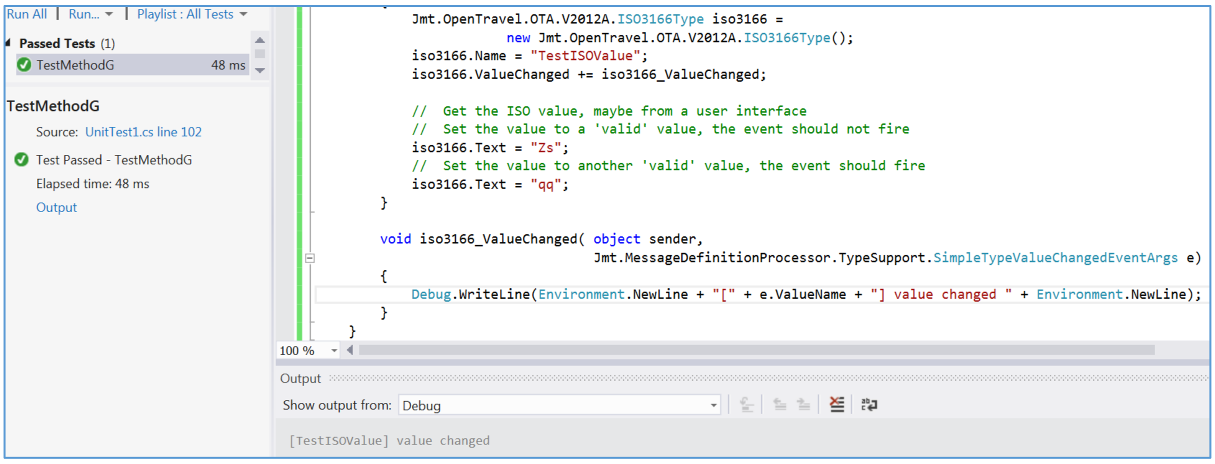
```
            Jmt.OpenTravel.OTA.V2012A.ISO3166Type iso3166 =
                        new Jmt.OpenTravel.OTA.V2012A.ISO3166Type();
            iso3166.ValidationFailed += iso3166_ValidationFailed;

            //  Get the ISO value, maybe from a user interface
            //  Set the value to a 'valid' value, the event should not fire
            iso3166.Text = "Zs";
            //  Set the value to an 'inavid' value, the event should fire
            iso3166.Text = "12";
        }

        void iso3166_ValidationFailed( object sender,
                                       Jmt.MessageDefinitionProcessor.TypeSupport.SimpleTypeValidationErrorEventArgs e)
        {
            Debug.WriteLine(Environment.NewLine + "Validation failed: " + Environment.NewLine);
            Debug.WriteLine("Current value:   " + e.CurrentValue );
            Debug.WriteLine("Attempted value: " + e.AttemptedValue);
        }
```

Output
Show output from: Debug

```
Validation failed:

Current value:   Zs
Attempted value: 12
```

Figure 11 Validation Failed Event Subscription

Along with the ‘validation failed’ event, the JMT types allow the developer to catch an event that signals when the value of a type changes. This would be particularly relevant for a User Interface (UI) component. This general theme is exemplified in the code fragment of Figure 12:



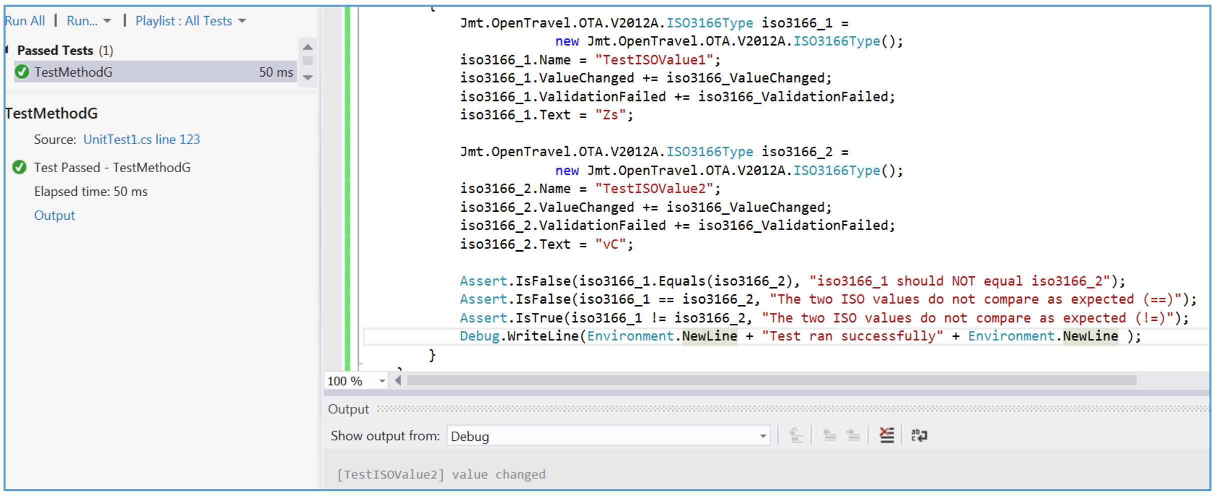
```
            Jmt.OpenTravel.OTA.V2012A.ISO3166Type iso3166 =
                        new Jmt.OpenTravel.OTA.V2012A.ISO3166Type();
            iso3166.Name = "TestISOValue";
            iso3166.ValueChanged += iso3166_ValueChanged;

            //  Get the ISO value, maybe from a user interface
            //  Set the value to a 'valid' value, the event should not fire
            iso3166.Text = "Zs";
            //  Set the value to another 'valid' value, the event should fire
            iso3166.Text = "qq";
        }

        void iso3166_ValueChanged( object sender,
                                   Jmt.MessageDefinitionProcessor.TypeSupport.SimpleTypeValueChangedEventArgs e)
        {
            Debug.WriteLine(Environment.NewLine + "[" + e.ValueName + "] value changed " + Environment.NewLine);
        }
    }
```

```
[TestISOValue] value changed
```

Figure 12 Value Changed Event Subscription

Note in this example how the object has been named so that the specific entity that is having its value changed can be identified from the information in the event handler argument (event handler could be shared).

Another key features of strong types is that they can be used in operator-type programming statements, they allow meaningful cloning and possess a type-specific `Equals()` method. The JMT types possess all these characteristics, as demonstrated in the code fragment of Figure 13:

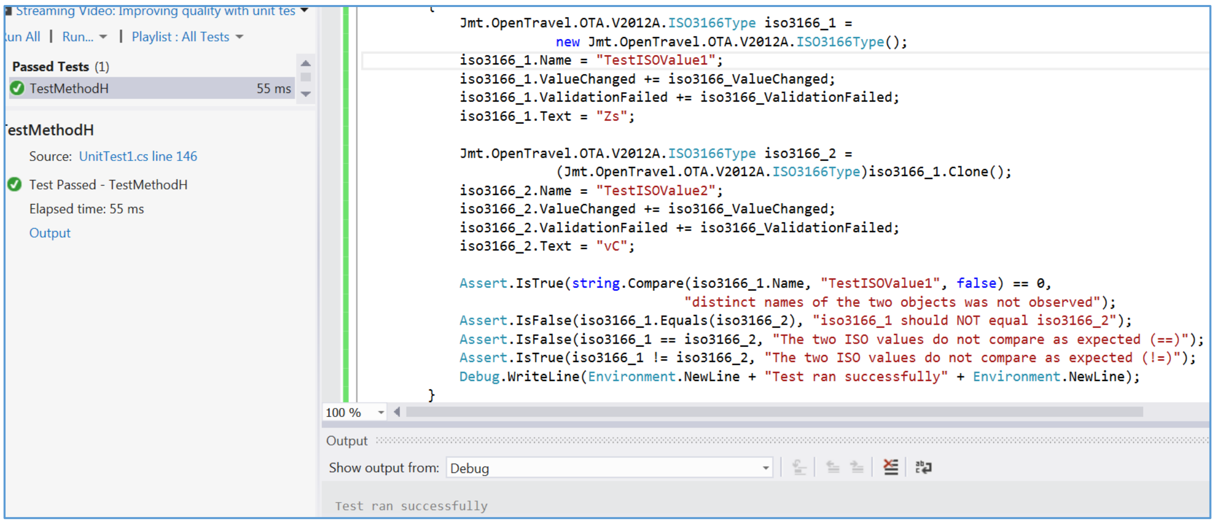


```
            Jmt.OpenTravel.OTA.V2012A.ISO3166Type iso3166_1 =
                        new Jmt.OpenTravel.OTA.V2012A.ISO3166Type();
            iso3166_1.Name = "TestISOValue1";
            iso3166_1.ValueChanged += iso3166_ValueChanged;
            iso3166_1.ValidationFailed += iso3166_ValidationFailed;
            iso3166_1.Text = "Zs";

            Jmt.OpenTravel.OTA.V2012A.ISO3166Type iso3166_2 =
                        new Jmt.OpenTravel.OTA.V2012A.ISO3166Type();
            iso3166_2.Name = "TestISOValue2";
            iso3166_2.ValueChanged += iso3166_ValueChanged;
            iso3166_2.ValidationFailed += iso3166_ValidationFailed;
            iso3166_2.Text = "vC";

            Assert.IsFalse(iso3166_1.Equals(iso3166_2), "iso3166_1 should NOT equal iso3166_2");
            Assert.IsFalse(iso3166_1 == iso3166_2, "The two ISO values do not compare as expected (==)");
            Assert.IsTrue(iso3166_1 != iso3166_2, "The two ISO values do not compare as expected (!=)");
            Debug.WriteLine(Environment.NewLine + "Test ran successfully" + Environment.NewLine );
        }
```

```
[TestISOValue2] value changed
```

Figure 13 Comparison Operators & Methods

All the assertions in this code fragment will pass for the two, distinct `ISO3166Type` objects. The developer can now choose whichever idiom they wish to assert (non-)equality between not only pairs of this specific type but any JMT type pairs. As noted above, in addition to these 'equality' idioms our strong types should allow for cloning. In Figure 14 below, this characteristic is seen in operation.



Streaming Video: Improving quality with unit tes

un All | Run... | Playlist : All Tests

Passed Tests (1)

TestMethodH 55 ms

estMethodH

Source: UnitTest1.cs line 146

Test Passed - TestMethodH

Elapsed time: 55 ms

Output

```
        {
            Jmt.OpenTravel.OTA.V2012A.ISO3166Type iso3166_1 =
                        new Jmt.OpenTravel.OTA.V2012A.ISO3166Type();
            iso3166_1.Name = "TestISOValue1";
            iso3166_1.ValueChanged += iso3166_ValueChanged;
            iso3166_1.ValidationFailed += iso3166_ValidationFailed;
            iso3166_1.Text = "Zs";

            Jmt.OpenTravel.OTA.V2012A.ISO3166Type iso3166_2 =
                        (Jmt.OpenTravel.OTA.V2012A.ISO3166Type)iso3166_1.Clone();
            iso3166_2.Name = "TestISOValue2";
            iso3166_2.ValueChanged += iso3166_ValueChanged;
            iso3166_2.ValidationFailed += iso3166_ValidationFailed;
            iso3166_2.Text = "vC";

            Assert.IsTrue(string.Compare(iso3166_1.Name, "TestISOValue1", false) == 0,
                                  "distinct names of the two objects was not observed");
            Assert.IsFalse(iso3166_1.Equals(iso3166_2), "iso3166_1 should NOT equal iso3166_2");
            Assert.IsFalse(iso3166_1 == iso3166_2, "The two ISO values do not compare as expected (==)");
            Assert.IsTrue(iso3166_1 != iso3166_2, "The two ISO values do not compare as expected (!=)");
            Debug.WriteLine(Environment.NewLine + "Test ran successfully" + Environment.NewLine);
        }
```

Output

Show output from: Debug

```
Test ran successfully
```

Figure 14 Cloning Method

Jet Messaging Technologies AG
Rotwandstrasse 35, 8004 Zurich, Switzerland
Phone +41 79 176 89 80

Email info@jet-messaging.com
www.jet-messaging.com