JMT and NDC/Airline Shopping
Accelerating development for NDC/Airline Shopping

This TechNote gives concrete information as to how JMT Types for New Distribution Capability (NDC) Shopping can be used. The type landscape for NDC is specified by Schemas published by IATA (http://www.iata.org/whatwedo/airline-distribution/ndc/Pages/default.aspx) and the first public release covers the Shopping dictionary. It is this that forms the types used in this TechNote.

Summary

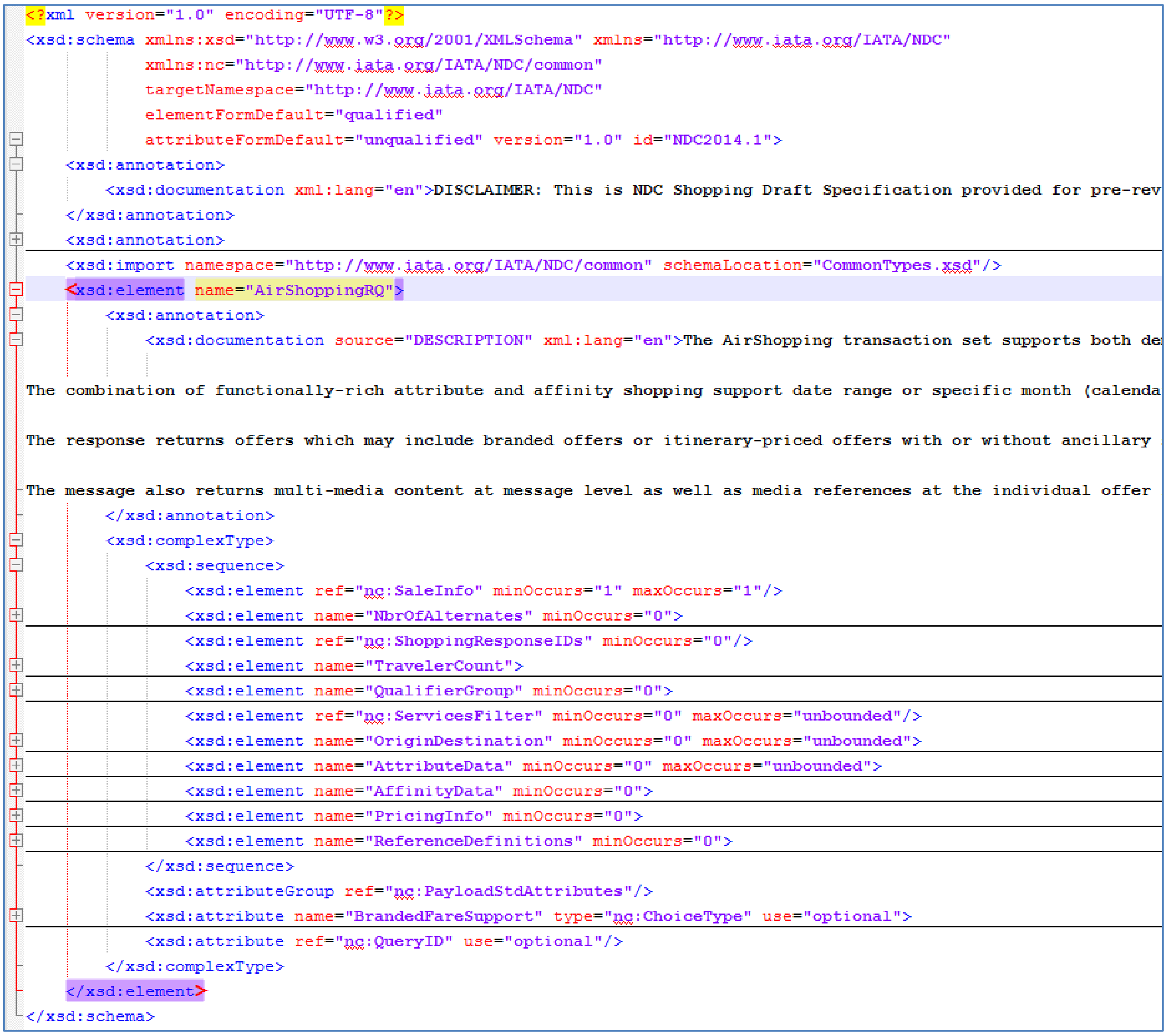
This TechNote demonstrates the following:

- the ease with which JMT types can be used by the developer
- how the developer, at coding time, gets to see the full range of Schema-based documentation
- how Intellisense and Auto-Completion are key accelerators of development
- how type validation (for every JMT type) is handled
- how the serialisation of JMT types can be achieved

The NDC Airline Shopping Schemas

The JMT processor generates a type landscape from the Schemas published by IATA. The basis of this TechNote is the Schemas at version 1.0 (id="NDC2014.1").

To illustrate how the NDC Airline Shopping type library can accelerate the development task, we will be looking specifically at the type named AirShoppingRQ contained in the Schema file AirShoppingRQ.xsd, an overview of this type is given in Figure 1.



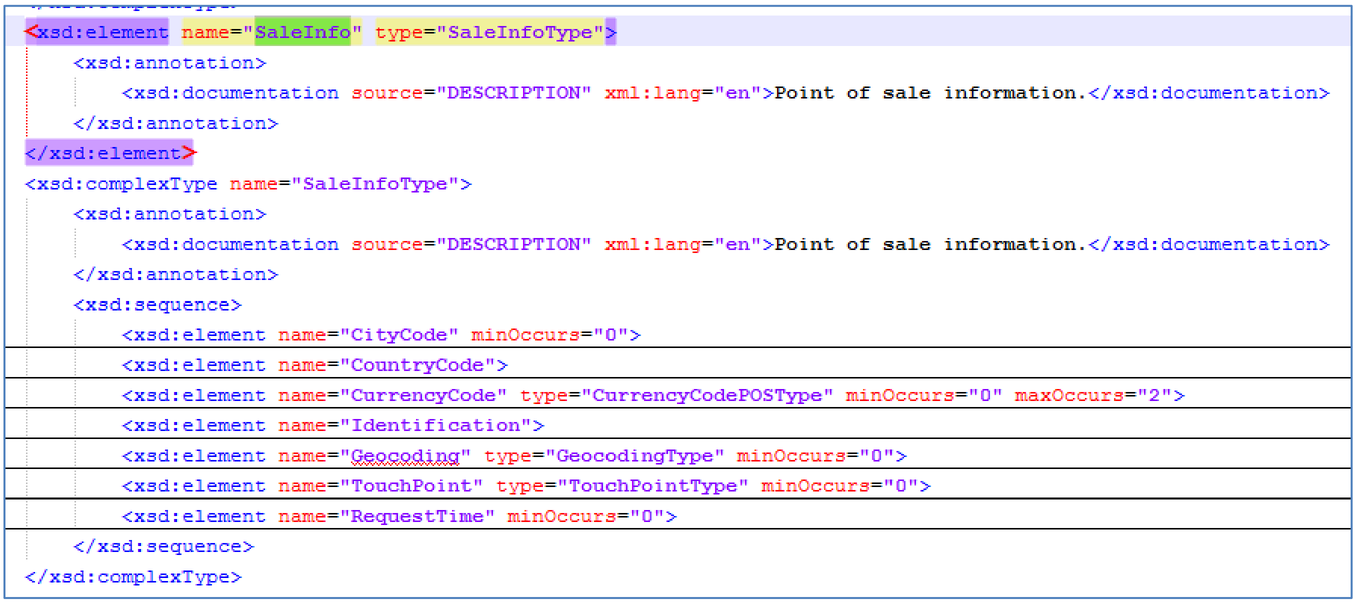
```
<?xml version="1.0" encoding="UTF-8"?>
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema" xmlns="http://www.iata.org/IATA/NDC"
            xmlns:nc="http://www.iata.org/IATA/NDC/common"
            targetNamespace="http://www.iata.org/IATA/NDC"
            elementFormDefault="qualified"
            attributeFormDefault="unqualified" version="1.0" id="NDC2014.1">
    <xsd:annotation>
        <xsd:documentation xml:lang="en">DISCLAIMER: This is NDC Shopping Draft Specification provided for pre-rev
    </xsd:annotation>
    <xsd:annotation>
    <xsd:import namespace="http://www.iata.org/IATA/NDC/common" schemaLocation="CommonTypes.xsd"/>
    <xsd:element name="AirShoppingRQ">
        <xsd:annotation>
            <xsd:documentation source="DESCRIPTION" xml:lang="en">The AirShopping transaction set supports both de

The combination of functionally-rich attribute and affinity shopping support date range or specific month (calenda

The response returns offers which may include branded offers or itinerary-priced offers with or without ancillary

The message also returns multi-media content at message level as well as media references at the individual offer
        </xsd:annotation>
        <xsd:complexType>
            <xsd:sequence>
                <xsd:element ref="nc:SaleInfo" minOccurs="1" maxOccurs="1"/>
                <xsd:element name="NbrOfAlternates" minOccurs="0">
                <xsd:element ref="nc:ShoppingResponseIDs" minOccurs="0"/>
                <xsd:element name="TravelerCount">
                <xsd:element name="QualifierGroup" minOccurs="0">
                <xsd:element ref="nc:ServicesFilter" minOccurs="0" maxOccurs="unbounded"/>
                <xsd:element name="OriginDestination" minOccurs="0" maxOccurs="unbounded">
                <xsd:element name="AttributeData" minOccurs="0" maxOccurs="unbounded">
                <xsd:element name="AffinityData" minOccurs="0">
                <xsd:element name="PricingInfo" minOccurs="0">
                <xsd:element name="ReferenceDefinitions" minOccurs="0">
            </xsd:sequence>
            <xsd:attributeGroup ref="nc:PayloadStdAttributes"/>
            <xsd:attribute name="BrandedFareSupport" type="nc:ChoiceType" use="optional">
            <xsd:attribute ref="nc:QueryID" use="optional"/>
        </xsd:complexType>
    </xsd:element>
</xsd:schema>
```

Figure 1 AirShoppingRQ Schema Definition

Here we see that essentially this type is composed of a sequence followed by attributes. Within the sequence some entities are mandatory (minOccurs=1 or use="required") whilst others are optional (minOccurs=0 or use="optional"). In particular we can identify the top-level mandatory entities in this message, see the following table:

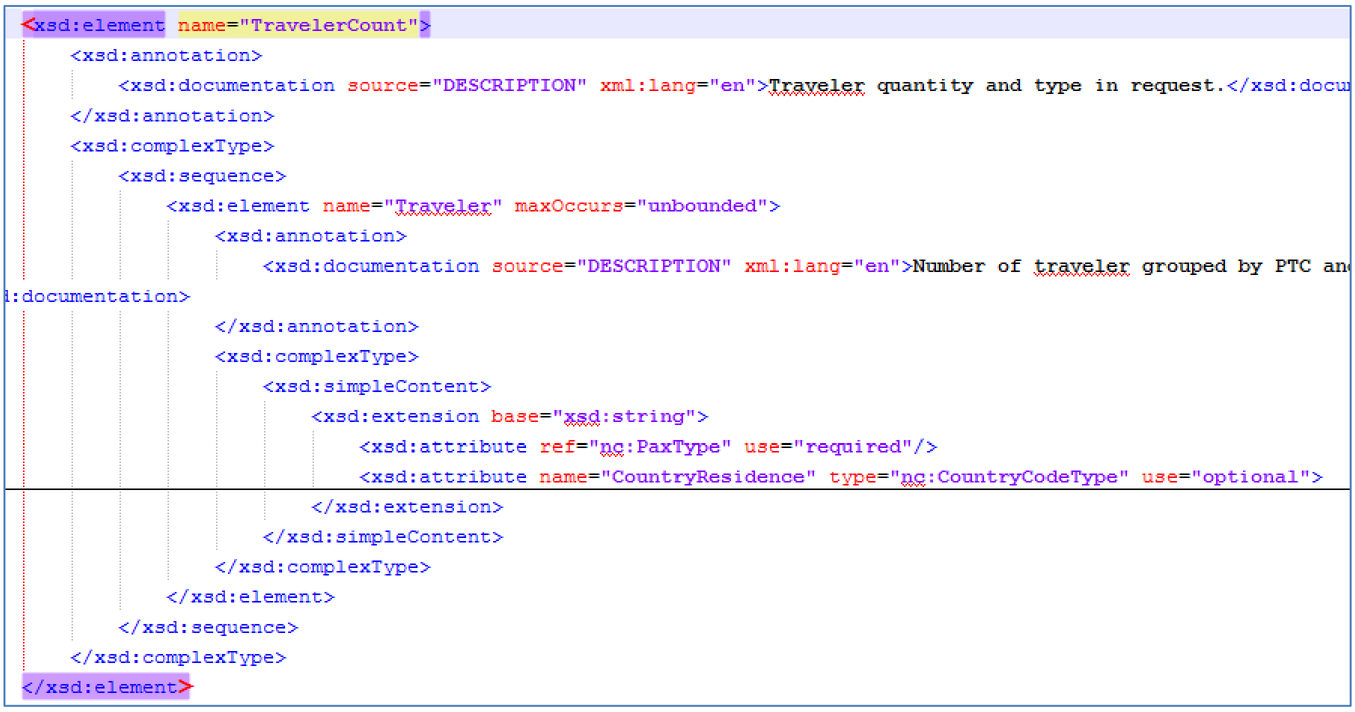
Entity	Comment
nc:SaleInfo	This is an element which must occur once only. The definition of this entity is specified in the Schema CommonTypes.xsd
TravelerCount	This is an element which by default must occur only once. The definition of this entity is specified locally, inline

So, in fact, the mandatory parts of this particular message are quite small. However the definitions of the two entities identified above are quite complex, see Figure 2 and Figure 3 below:



```
<xsd:element name="SaleInfo" type="SaleInfoType">
    <xsd:annotation>
        <xsd:documentation source="DESCRIPTION" xml:lang="en">Point of sale information.</xsd:documentation>
    </xsd:annotation>
</xsd:element>
<xsd:complexType name="SaleInfoType">
    <xsd:annotation>
        <xsd:documentation source="DESCRIPTION" xml:lang="en">Point of sale information.</xsd:documentation>
    </xsd:annotation>
    <xsd:sequence>
        <xsd:element name="CityCode" minOccurs="0">
        <xsd:element name="CountryCode">
        <xsd:element name="CurrencyCode" type="CurrencyCodePOSType" minOccurs="0" maxOccurs="2">
        <xsd:element name="Identification">
        <xsd:element name="Geocoding" type="GeocodingType" minOccurs="0">
        <xsd:element name="TouchPoint" type="TouchPointType" minOccurs="0">
        <xsd:element name="RequestTime" minOccurs="0">
    </xsd:sequence>
</xsd:complexType>
```

Figure 2 SaleInfo/SaleInfoType Type Definitions



```
<xsd:element name="TravelerCount">
    <xsd:annotation>
        <xsd:documentation source="DESCRIPTION" xml:lang="en">Traveler quantity and type in request.</xsd:docu
    </xsd:annotation>
    <xsd:complexType>
        <xsd:sequence>
            <xsd:element name="Traveler" maxOccurs="unbounded">
                <xsd:annotation>
                    <xsd:documentation source="DESCRIPTION" xml:lang="en">Number of traveler grouped by PTC an
:documentation>
                </xsd:annotation>
                <xsd:complexType>
                    <xsd:simpleContent>
                        <xsd:extension base="xsd:string">
                            <xsd:attribute ref="ng:PaxType" use="required"/>
                            <xsd:attribute name="CountryResidence" type="ng:CountryCodeType" use="optional">
                        </xsd:extension>
                    </xsd:simpleContent>
                </xsd:complexType>
            </xsd:element>
        </xsd:sequence>
    </xsd:complexType>
</xsd:element>
```

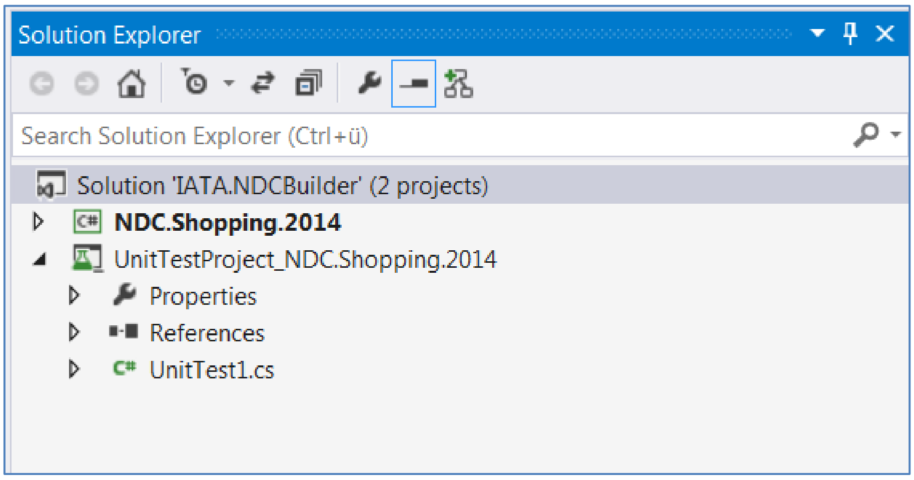
Figure 3 TravelCount Local Type Definition

The JMT Type - Testing

To explore this request message type as it appears in the JMT library, we have constructed a Visual Studio Solution containing the following:

Item	Comment
Project: NDC.Shopping.2014	This is a Class Library project containing the source code generated from the JMT Processor for NDC/Airline Shopping Schemas V2014.1
Project: UnitTestProject_NDC.Shopping.2014	This is a Unit Test project which we will use to exercise the generated code

This Solution/Project layout is as shown in Figure 4:


Solution Explorer

Search Solution Explorer (Ctrl+ü)

Solution 'IATA.NDCBuilder' (2 projects)
- NDC.Shopping.2014
- UnitTestProject_NDC.Shopping.2014
  - Properties
  - References
  - UnitTest1.cs

Figure 4 Solution/Project Layout

Within the project NDC.Shopping.2014, there are folders in which the types corresponding to Schema entities are held. For example, in the case of types arising from element definitions, the folder contents are as shown, below:

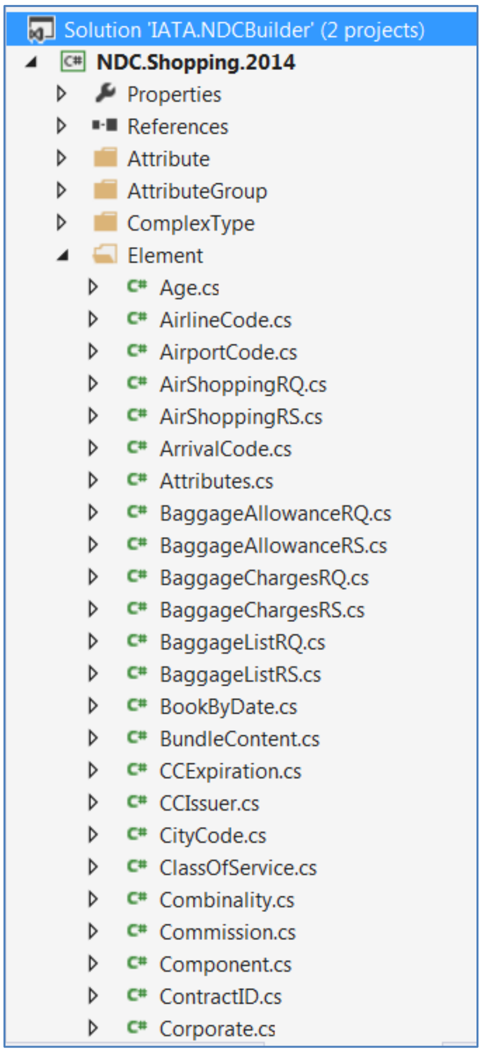

Solution 'IATA.NDCBuilder' (2 projects)
- NDC.Shopping.2014
  - Properties
  - References
  - Attribute
  - AttributeGroup
  - ComplexType
  - Element
    - Age.cs
    - AirlineCode.cs
    - AirportCode.cs
    - AirShoppingRQ.cs
    - AirShoppingRS.cs
    - ArrivalCode.cs
    - Attributes.cs
    - BaggageAllowanceRQ.cs
    - BaggageAllowanceRS.cs
    - BaggageChargesRQ.cs
    - BaggageChargesRS.cs
    - BaggageListRQ.cs
    - BaggageListRS.cs
    - BookByDate.cs
    - BundleContent.cs
    - CCExpiration.cs
    - CCIssuer.cs
    - CityCode.cs
    - ClassOfService.cs
    - Combinality.cs
    - Commission.cs
    - Component.cs
    - ContractID.cs
    - Corporate.cs

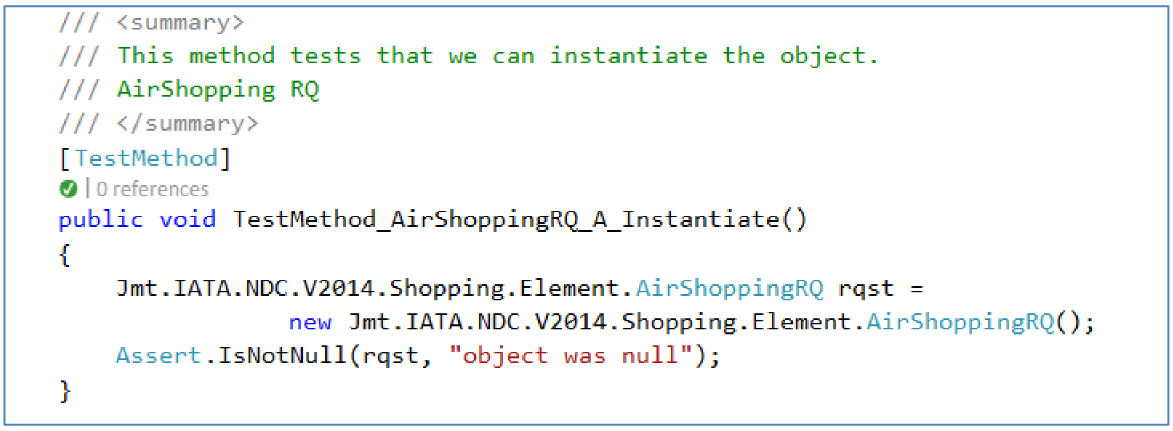
Figure 5 NDC/Airline Shopping Element-based Types

The Unit Test project (UnitTestProject_NFC.2014) is designed to exercise classes in the main library project (NDC.Shopping.2014). In particular we will look at the NDC/Airline Shopping, AirShoppingRQ, which represents a message. In practice, of course, a development project would have a Reference to the *installed* JMT library rather than one to the generated code project. However, for the purpose of this TechNote it is useful to deal with the generated code in this way. To illustrate the simplicity of using a JMT library, the Unit Test Methods are designed to gradually build to a point where the message is serialized. The following table lists the tests:

Test	Comment
TestMethod_AirShoppingRQ_A_Instantiate	This test asserts that the JMT class Jmt.IATA.NDC.V2014.Shopping.Element.AirShoppingRQ can be instantiated without exception.
TestMethod_AirShoppingRQ_B_InitialState	This test asserts that the initial state of the JMT class Jmt.IATA.NDC.V2014.Shopping.Element.AirShoppingRQ is as expected.
TestMethod_AirShoppingRQ_C_Populate	This test asserts that the JMT class Jmt.IATA.NDC.V2014.Shopping.Element.AirShoppingRQ can be populated with its mandatory data without error or exception.
TestMethod_AirShoppingRQ_D_Serialise	This test asserts that the JMT class Jmt.IATA.NDC.V2014.Shopping.Element.AirShoppingRQ can be serialised correctly, in this case to a file.

We will now look at the details of each of these tests.

AirShoppingRQ_A_Instantiate
This test, shown in the figure below (Figure 6), ensures that the type AirShoppingRQ can be instantiated.
Here, the full name of the JMT is shown "Jmt.IATA.NDC.V2014.Shopping.Element.AirShoppingRQ". This referencing of types follows a standard structure within libraries generated by the JMT Processor.



```
/// <summary>
/// This method tests that we can instantiate the object.
/// AirShopping RQ
/// </summary>
[TestMethod]
| 0 references
public void TestMethod_AirShoppingRQ_A_Instantiate()
{
    Jmt.IATA.NDC.V2014.Shopping.Element.AirShoppingRQ rqst =
                new Jmt.IATA.NDC.V2014.Shopping.Element.AirShoppingRQ();
    Assert.IsNotNull(rqst, "object was null");
}
```

Figure 6 AirShoppingRQ_A_Instantiate Test Method

In particular the elements of the type namespace have the following significance:

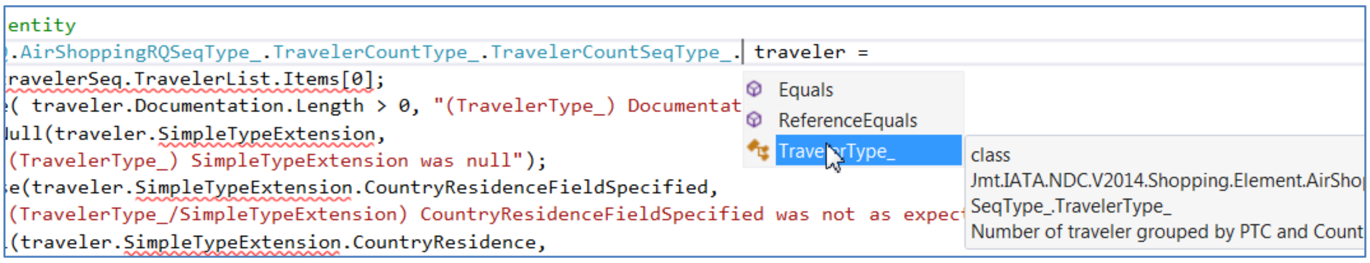
Item	Description
IATA	Within JMT libraries this component is referred to as the Provider Name
NDC	Within JMT libraries this component is referred to as the Provider Ident
V2014	Within JMT libraries this component is referred to as the Provider Version
Element	This component is added where the underlying Schema entity of the type is a element

Figure 6 shows how straightforward it is to use JMT types, they operate as any other type in the C#/.Net compendium of types. Once instantiated the type representing the message AirShoppingRQ, can be used just like any other object by the developer. In particular, for this type there are the following Properties and Methods:

Entity Name (type)	Type (* - Jmt.IATA.NDC.V2014.Shopping)	Comment
Documentation (Property)	String	Returns the documentation assigned to the type by the Schema writer

AirShoppingRQSeq (Property)	GuardedList<AirShoppingRQSeqType_>	Returns the list of sequence entities that exist with AirShoppingRQ. The returned list is a Generic one where the specific type is defined in an inner (nested) class to AirShoppingRQ
PayloadStdAttributes (Property)	*.AttributeGroup.PayloadStdAttributesType	Returns the AttributeGroup representing the PayloadStdAttributes as defined in CommonTypes.xsd
BrandedFareSupport (Property)	ChoiceType	Use this to set or get the object whose type is defined in CommonTypes.xsd
BrandedFareSupportField Specified (Property)	bool	Returns a flag indicating if the BrandedFareSupport field in AirShoppingRQ has been set (instantiated). (§) Initially, all optional fields which are reference types, have the 'value' of null. This means that the overall memory requirement of objects is kept to a minimum. When the field is set, then this "tracker" boolean gets set internally to true
QueryID (Property)	*.Attribute.QueryIDType	Use this to set or get the object whose type is defined in CommonTypes.xsd
QueryIDFieldSpecified (Property)	bool	Returns a flag indicating if the QueryID field in AirShoppingRQ has been set (instantiated). See (§) above
WriteXml (Method)	Void	Serialises the AirShoppingRQ to XML given a configured XmlWriter stream (parameter). Use this when the class **IS NOT** inherited
SerializeToXml (Method)	Void	Serialise the AirShoppingRQ to XML given a configured XmlWriter stream (parameter).Use this when the class **IS** inherited
ReadXml (Method)	Void	Deserialise data from a specified XmlReader stream (parameter) to an AirShoppingRQ object. Use this when the class **IS NOT** inherited
DerializeFromXml	Void	Deserialise data from a specified XmlReader stream (parameter) to an AirShoppingRQ object. Use this when the class **IS** inherited
== (Operator)	bool	This operator permits boolean comparison between two AirShoppingRQ objects
!= (Operator)	bool	This operator permits boolean comparison between two AirShoppingRQ objects
Equals (Method)	bool	This operator permits comparison between two AirShoppingRQ objects
Clone (Method)	object	This method performs a deep clone of an AirShoppingRQ object
ToString (Method)	string	This method provides a string represenation of an AirShoppingRQ object
AirShoppingRQSeqType_ (inner class)	-	This inner class represents the collection of objects comprising the sequence within the AirShoppingRQ entity
BrandedFareSupportType _	-	This inner class represents the locally defined Attribute based upon the type ChoiceType defined in CommonTypes.xsd

From the above table it should be clear how well provisioned the types from JMT are in comparison with those produced by current tooling approaches. This remarkable contrast can be further judged when we look at the types associated with the properties and collections above. Current tooling would attenuate these types into basic .Net types for example. In JMT types all definitional structures expressed in the Schema are transformed, local definitions, wherever they occur, become inner (local) classes, thus reflecting the Schema itself.
At first sight, the (full) names of JMT types, Properties and Methods can seem to be long and unwieldy. In practice, the developer will find that auto-completion removes the need for typing, an example is shown below:



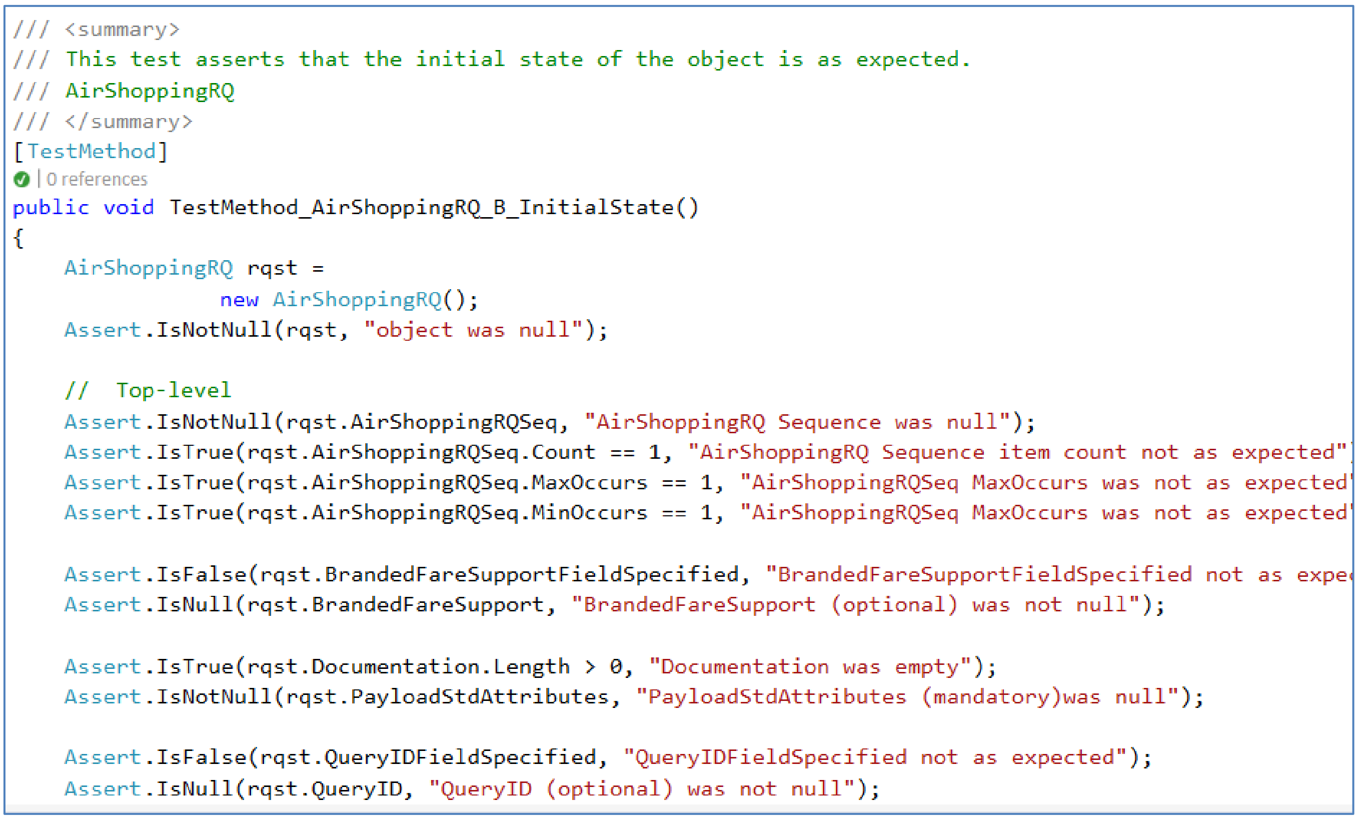
```
entity
).AirShoppingRQSeqType_.TravelerCountType_.TravelerCountSeqType_.| traveler =
ravelerSeq.TravelerList.Items[0];
:( traveler.Documentation.Length > 0, "(TravelerType_) Documentat
Null(traveler.SimpleTypeExtension,
(TravelerType_) SimpleTypeExtension was null");
e(traveler.SimpleTypeExtension.CountryResidenceFieldSpecified,
(TravelerType_/SimpleTypeExtension) CountryResidenceFieldSpecified was not as expec
(traveler.SimpleTypeExtension.CountryResidence,
```

class
Jmt.IATA.NDC.V2014.Shopping.Element.AirSho
SeqType_.TravelerType_
Number of traveler grouped by PTC and Count

Figure 7 Auto-Completion & Documentation

AirShoppingRQ_B_InitialState

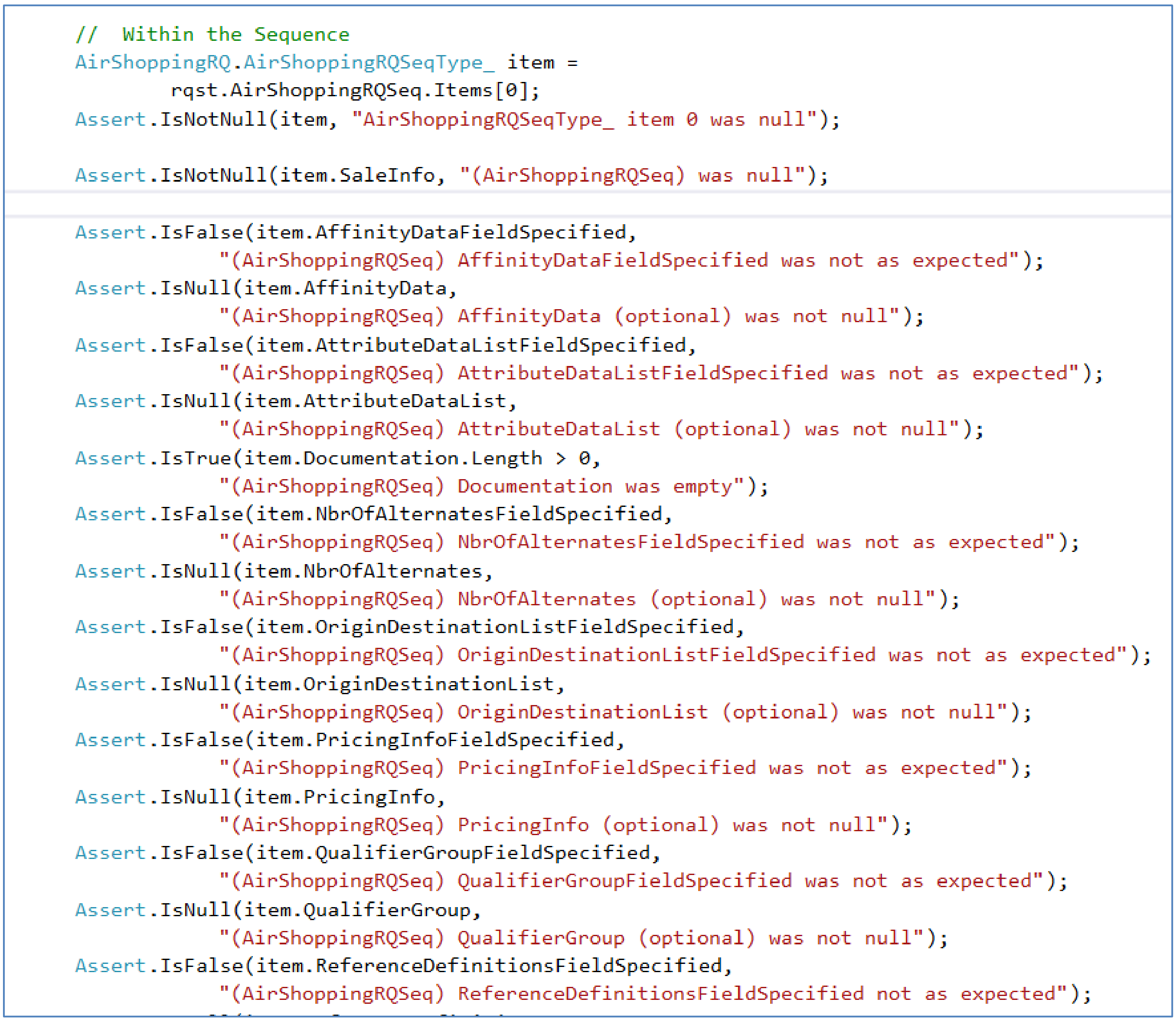
In this test the class, AirShoppingRQ, is instantiated, then the initial state of the class is asserted.



```
/// <summary>
/// This test asserts that the initial state of the object is as expected.
/// AirShoppingRQ
/// </summary>
[TestMethod]
| 0 references
public void TestMethod_AirShoppingRQ_B_InitialState()
{
    AirShoppingRQ rqst =
                new AirShoppingRQ();
    Assert.IsNotNull(rqst, "object was null");

    //  Top-level
    Assert.IsNotNull(rqst.AirShoppingRQSeq, "AirShoppingRQ Sequence was null");
    Assert.IsTrue(rqst.AirShoppingRQSeq.Count == 1, "AirShoppingRQ Sequence item count not as expected"
    Assert.IsTrue(rqst.AirShoppingRQSeq.MaxOccurs == 1, "AirShoppingRQSeq MaxOccurs was not as expected
    Assert.IsTrue(rqst.AirShoppingRQSeq.MinOccurs == 1, "AirShoppingRQSeq MaxOccurs was not as expected

    Assert.IsFalse(rqst.BrandedFareSupportFieldSpecified, "BrandedFareSupportFieldSpecified not as expe
    Assert.IsNull(rqst.BrandedFareSupport, "BrandedFareSupport (optional) was not null");

    Assert.IsTrue(rqst.Documentation.Length > 0, "Documentation was empty");
    Assert.IsNotNull(rqst.PayloadStdAttributes, "PayloadStdAttributes (mandatory)was null");

    Assert.IsFalse(rqst.QueryIDFieldSpecified, "QueryIDFieldSpecified not as expected");
    Assert.IsNull(rqst.QueryID, "QueryID (optional) was not null");
```

Figure 8 (a) AirShoppingRQ_B_InitialState Test Method

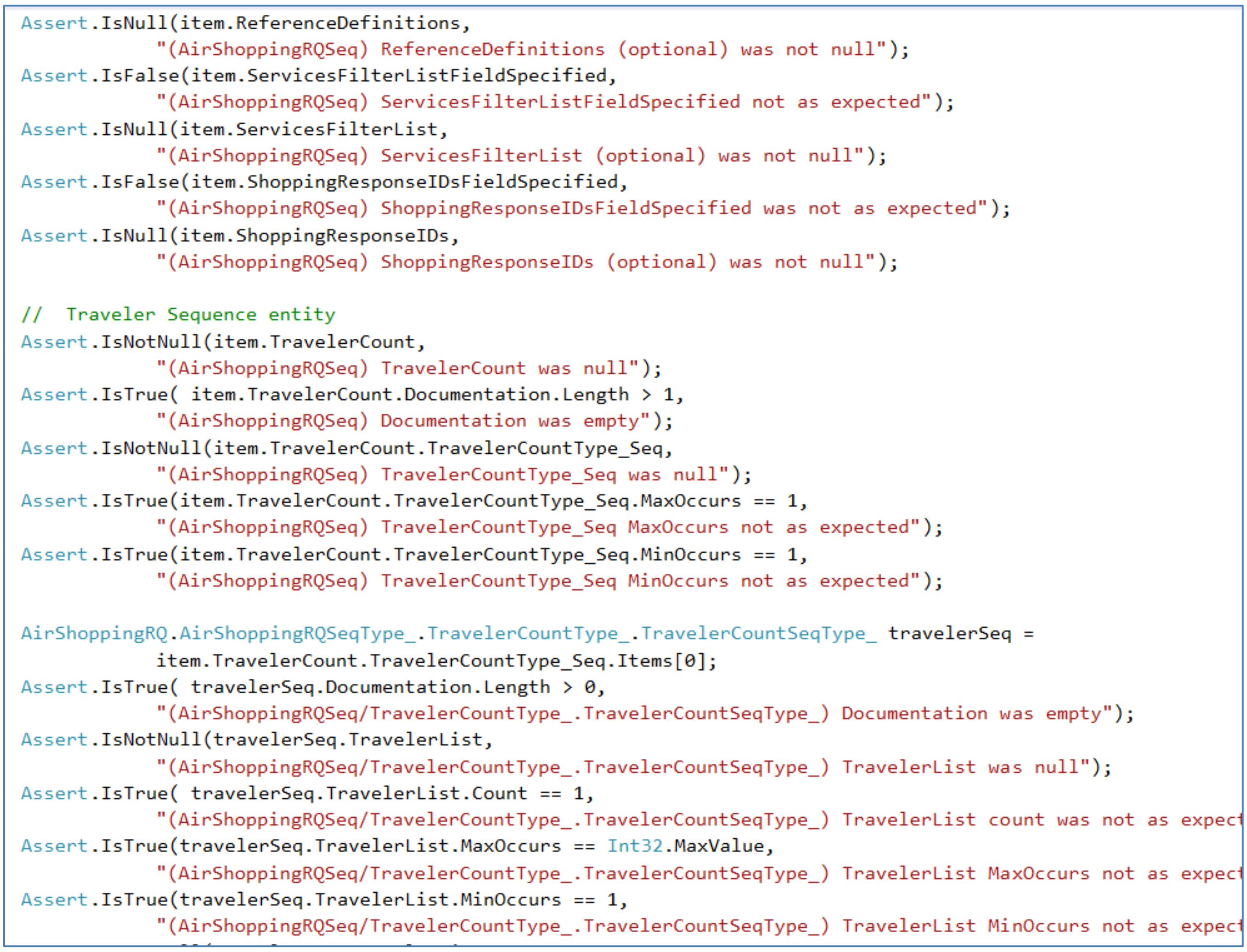
In above Figure 8 we see how, after instantiating the basic object, we assert that the Property AirShoppingRQSeq, which returns the inner collection representing the sequence expressed in the source Schema, is not null. In addition, we test its min- and maxOccurs and its (defaulted) collection item count. We also test for the basic state of the returned object from the property BrandedFareSupport as well as asserting, since it's optional, the state of its tracker field, BrandedFareSupportFieldSpecified. We test that the Documentation string is returned non-empty.
Since the PayloadStdAttributes entity in the Schema definition is mandatory, we assert that the corresponding property in the JMT type returns non-null. In contrast, for the QueryID property, we asset that the tracker field is false and that the corresponding property returns null.



```
    //  Within the Sequence
    AirShoppingRQ.AirShoppingRQSeqType_ item =
            rqst.AirShoppingRQSeq.Items[0];
    Assert.IsNotNull(item, "AirShoppingRQSeqType_ item 0 was null");

    Assert.IsNotNull(item.SaleInfo, "(AirShoppingRQSeq) was null");

    Assert.IsFalse(item.AffinityDataFieldSpecified,
                "(AirShoppingRQSeq) AffinityDataFieldSpecified was not as expected");
    Assert.IsNull(item.AffinityData,
                "(AirShoppingRQSeq) AffinityData (optional) was not null");
    Assert.IsFalse(item.AttributeDataListFieldSpecified,
                "(AirShoppingRQSeq) AttributeDataListFieldSpecified was not as expected");
    Assert.IsNull(item.AttributeDataList,
                "(AirShoppingRQSeq) AttributeDataList (optional) was not null");
    Assert.IsTrue(item.Documentation.Length > 0,
                "(AirShoppingRQSeq) Documentation was empty");
    Assert.IsFalse(item.NbrOfAlternatesFieldSpecified,
                "(AirShoppingRQSeq) NbrOfAlternatesFieldSpecified was not as expected");
    Assert.IsNull(item.NbrOfAlternates,
                "(AirShoppingRQSeq) NbrOfAlternates (optional) was not null");
    Assert.IsFalse(item.OriginDestinationListFieldSpecified,
                "(AirShoppingRQSeq) OriginDestinationListFieldSpecified was not as expected");
    Assert.IsNull(item.OriginDestinationList,
                "(AirShoppingRQSeq) OriginDestinationList (optional) was not null");
    Assert.IsFalse(item.PricingInfoFieldSpecified,
                "(AirShoppingRQSeq) PricingInfoFieldSpecified was not as expected");
    Assert.IsNull(item.PricingInfo,
                "(AirShoppingRQSeq) PricingInfo (optional) was not null");
    Assert.IsFalse(item.QualifierGroupFieldSpecified,
                "(AirShoppingRQSeq) QualifierGroupFieldSpecified was not as expected");
    Assert.IsNull(item.QualifierGroup,
                "(AirShoppingRQSeq) QualifierGroup (optional) was not null");
    Assert.IsFalse(item.ReferenceDefinitionsFieldSpecified,
                "(AirShoppingRQSeq) ReferenceDefinitionsFieldSpecified not as expected");
```

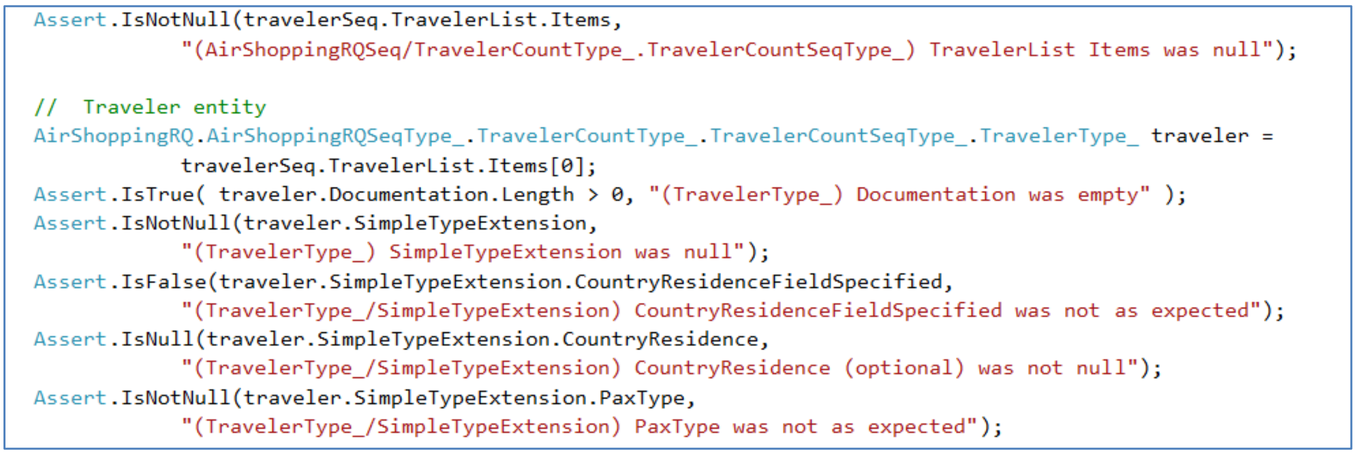
Figure 9 (b) AirShoppingRQ_B_InitialState Test Method

In this section of the "initial state" test, we look at the individual entities within the sequence as defined in the Schema and reflected in the AirShoppingRQ inner type AirShoppingRQSeqType_ (note: all inner types of classes are signaled by having names that end in the "_" character).
As a first step, since the minOccurs of the sequence is 1 and a single item will have been inserted in the collection as part of the overall instantiation process, we retrieve the first item from the collection and validate the state of its properties.
We then ensure that the SaleInfo Property returns non-null as the SaleInfo entity in the Schema is defined as mandatory.
We then proceed to validate the state of the remaining objects returned from the Properties of the sequence NbrOfAlternates, ShoppingResponseIDs, TravelerCount, QualifierGroup, ServiceFilter, OriginDestination, AttributeData, AffinityData, PricingInfo and ReferenceDefinitions. These assertions are shown in Figure 10 and Figure 11



```
Assert.IsNull(item.ReferenceDefinitions,
            "(AirShoppingRQSeq) ReferenceDefinitions (optional) was not null");
Assert.IsFalse(item.ServicesFilterListFieldSpecified,
            "(AirShoppingRQSeq) ServicesFilterListFieldSpecified not as expected");
Assert.IsNull(item.ServicesFilterList,
            "(AirShoppingRQSeq) ServicesFilterList (optional) was not null");
Assert.IsFalse(item.ShoppingResponseIDsFieldSpecified,
            "(AirShoppingRQSeq) ShoppingResponseIDsFieldSpecified was not as expected");
Assert.IsNull(item.ShoppingResponseIDs,
            "(AirShoppingRQSeq) ShoppingResponseIDs (optional) was not null");

//  Traveler Sequence entity
Assert.IsNotNull(item.TravelerCount,
            "(AirShoppingRQSeq) TravelerCount was null");
Assert.IsTrue( item.TravelerCount.Documentation.Length > 1,
            "(AirShoppingRQSeq) Documentation was empty");
Assert.IsNotNull(item.TravelerCount.TravelerCountType_Seq,
            "(AirShoppingRQSeq) TravelerCountType_Seq was null");
Assert.IsTrue(item.TravelerCount.TravelerCountType_Seq.MaxOccurs == 1,
            "(AirShoppingRQSeq) TravelerCountType_Seq MaxOccurs not as expected");
Assert.IsTrue(item.TravelerCount.TravelerCountType_Seq.MinOccurs == 1,
            "(AirShoppingRQSeq) TravelerCountType_Seq MinOccurs not as expected");

AirShoppingRQ.AirShoppingRQSeqType_.TravelerCountType_.TravelerCountSeqType_ travelerSeq =
            item.TravelerCount.TravelerCountType_Seq.Items[0];
Assert.IsTrue( travelerSeq.Documentation.Length > 0,
            "(AirShoppingRQSeq/TravelerCountType_.TravelerCountSeqType_) Documentation was empty");
Assert.IsNotNull(travelerSeq.TravelerList,
            "(AirShoppingRQSeq/TravelerCountType_.TravelerCountSeqType_) TravelerList was null");
Assert.IsTrue( travelerSeq.TravelerList.Count == 1,
            "(AirShoppingRQSeq/TravelerCountType_.TravelerCountSeqType_) TravelerList count was not as expect
Assert.IsTrue(travelerSeq.TravelerList.MaxOccurs == Int32.MaxValue,
            "(AirShoppingRQSeq/TravelerCountType_.TravelerCountSeqType_) TravelerList MaxOccurs not as expect
Assert.IsTrue(travelerSeq.TravelerList.MinOccurs == 1,
            "(AirShoppingRQSeq/TravelerCountType_.TravelerCountSeqType_) TravelerList MinOccurs not as expect
```

Figure 10 (c) AirShoppingRQ_B_InitialState Test Method



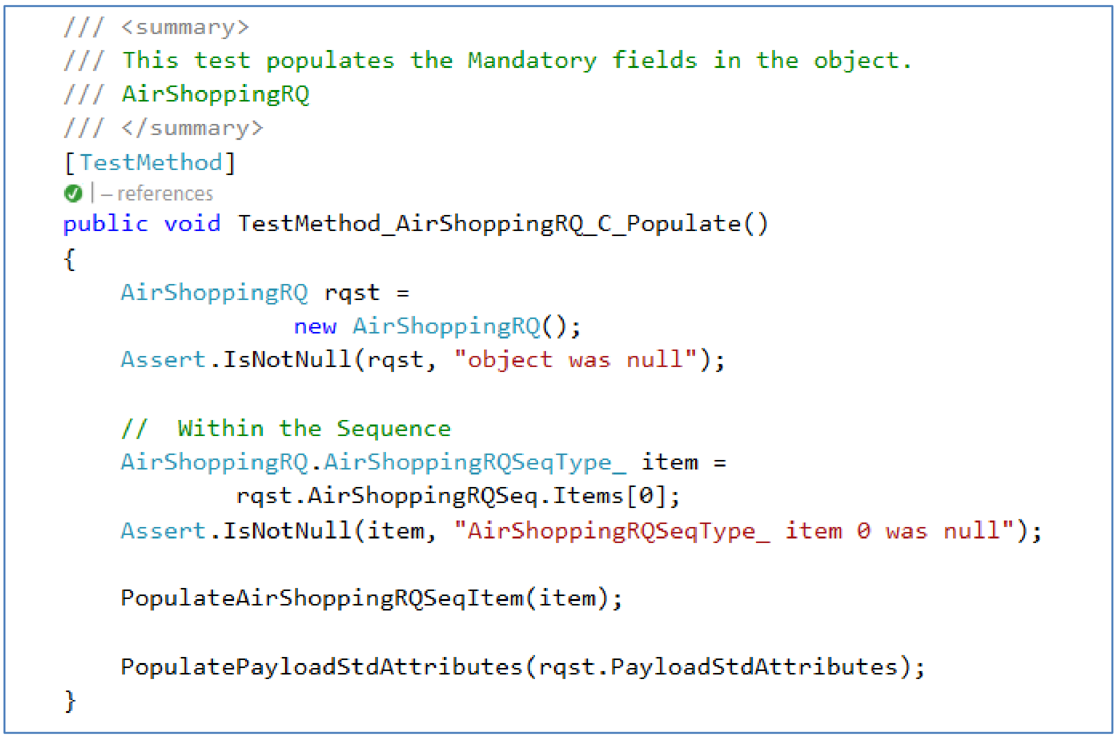
```
Assert.IsNotNull(travelerSeq.TravelerList.Items,
            "(AirShoppingRQSeq/TravelerCountType_.TravelerCountSeqType_) TravelerList Items was null");

//  Traveler entity
AirShoppingRQ.AirShoppingRQSeqType_.TravelerCountType_.TravelerCountSeqType_.TravelerType_ traveler =
            travelerSeq.TravelerList.Items[0];
Assert.IsTrue( traveler.Documentation.Length > 0, "(TravelerType_) Documentation was empty" );
Assert.IsNotNull(traveler.SimpleTypeExtension,
            "(TravelerType_) SimpleTypeExtension was null");
Assert.IsFalse(traveler.SimpleTypeExtension.CountryResidenceFieldSpecified,
            "(TravelerType_/SimpleTypeExtension) CountryResidenceFieldSpecified was not as expected");
Assert.IsNull(traveler.SimpleTypeExtension.CountryResidence,
            "(TravelerType_/SimpleTypeExtension) CountryResidence (optional) was not null");
Assert.IsNotNull(traveler.SimpleTypeExtension.PaxType,
            "(TravelerType_/SimpleTypeExtension) PaxType was not as expected");
```

Figure 11 (d) AirShoppingRQ_B_InitialState Test Method

From the foregoing, it should be clear that the JMT types have full fidelity with the definitions expressed in the source Schema. To be sure, we see particularly collection types that are specific to the programming world of C#/.Net but this is to be expected. That we have access to these collections in a natural way from the developer standpoint is key.

AirShoppingRQ_C_Populate

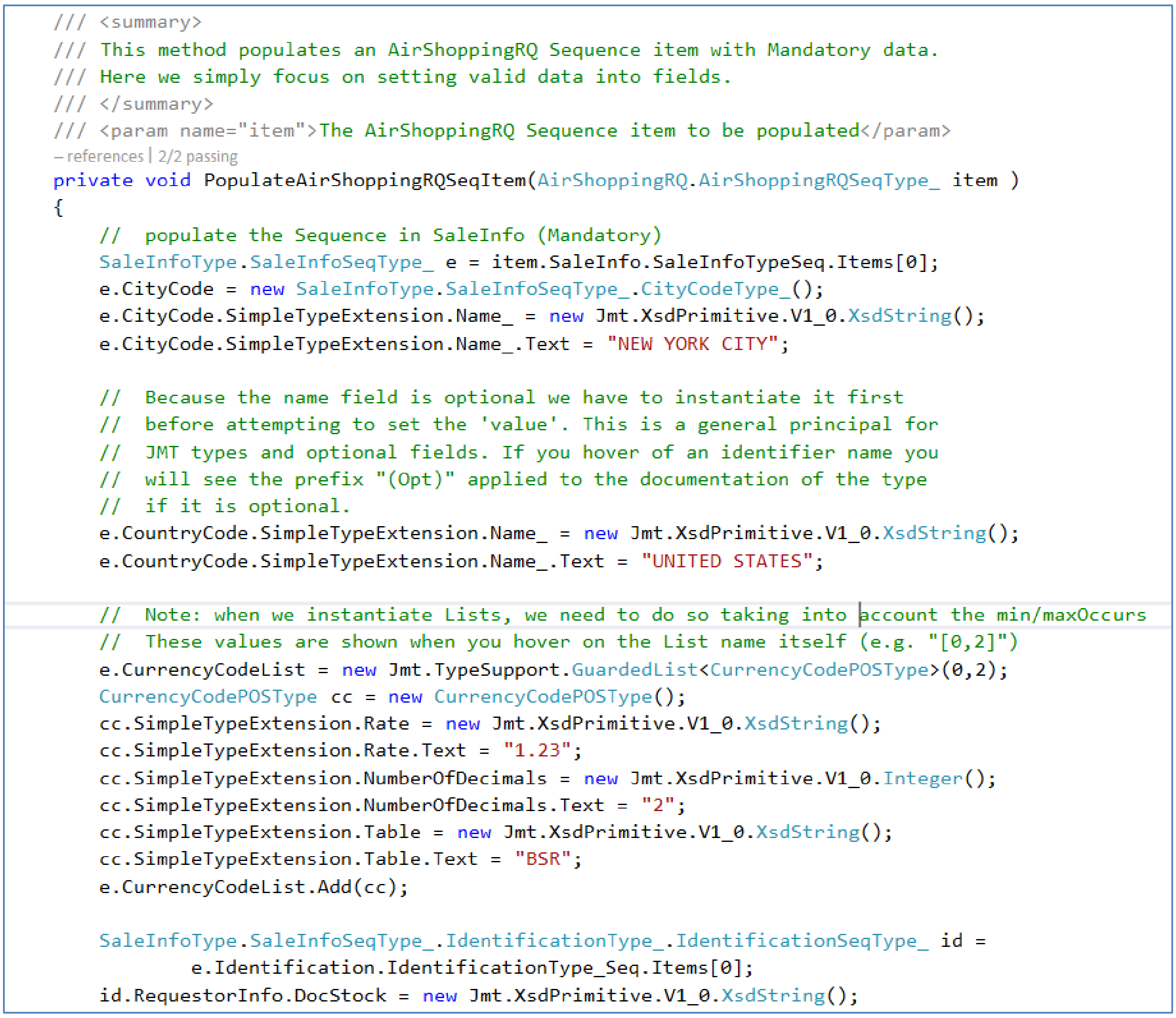
Once we have asserted the initial state of our AirShoppingRQ type, then it’s natural to then assert that the populating of the type proceeds as expected. In such a ‘test’ we perhaps are not so much interested in the ‘value’ of things but the usability of types and the overall type structure. With this in mind, let’s take a look at the ‘populate’ test.
As a pattern when ‘populate’ testing of JMT types, we break out the individual entities and handle them in private methods, these can be clearly seen in Figure 12, below:



```
/// <summary>
/// This test populates the Mandatory fields in the object.
/// AirShoppingRQ
/// </summary>
[TestMethod]
✓ | – references
public void TestMethod_AirShoppingRQ_C_Populate()
{
    AirShoppingRQ rqst =
                new AirShoppingRQ();
    Assert.IsNotNull(rqst, "object was null");

    //  Within the Sequence
    AirShoppingRQ.AirShoppingRQSeqType_ item =
            rqst.AirShoppingRQSeq.Items[0];
    Assert.IsNotNull(item, "AirShoppingRQSeqType_ item 0 was null");

    PopulateAirShoppingRQSeqItem(item);

    PopulatePayloadStdAttributes(rqst.PayloadStdAttributes);
}
```

Figure 12 (a) AirShoppingRQ Populate Test Method

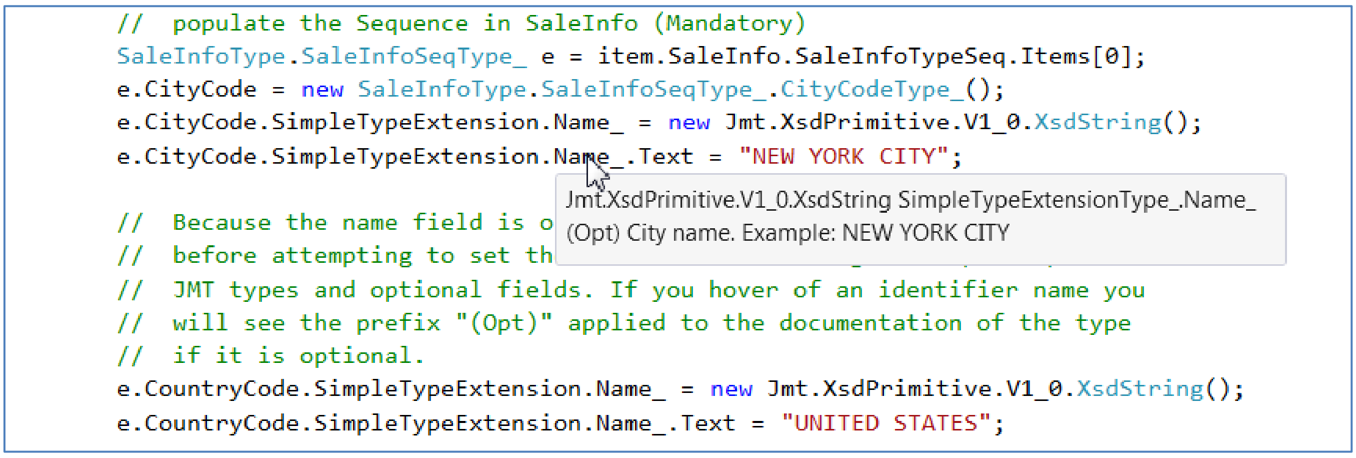
We start by retrieving the (single) existing item in the sequence collection within AirShoppingRQ (this sequence collection, by default, will have an existing typed item added). After asserting that this retrieved item is non-null we proceed to call the methods that populate it, PopulateAirShoppingRQSeqItem, and then populate the PayloadStdAttributes entity, PopulatePayloadStdAttributes.



```
    /// <summary>
    /// This method populates an AirShoppingRQ Sequence item with Mandatory data.
    /// Here we simply focus on setting valid data into fields.
    /// </summary>
    /// <param name="item">The AirShoppingRQ Sequence item to be populated</param>
    – references | 2/2 passing
    private void PopulateAirShoppingRQSeqItem(AirShoppingRQ.AirShoppingRQSeqType_ item )
    {
        //  populate the Sequence in SaleInfo (Mandatory)
        SaleInfoType.SaleInfoSeqType_ e = item.SaleInfo.SaleInfoTypeSeq.Items[0];
        e.CityCode = new SaleInfoType.SaleInfoSeqType_.CityCodeType_();
        e.CityCode.SimpleTypeExtension.Name_ = new Jmt.XsdPrimitive.V1_0.XsdString();
        e.CityCode.SimpleTypeExtension.Name_.Text = "NEW YORK CITY";

        //  Because the name field is optional we have to instantiate it first
        //  before attempting to set the 'value'. This is a general principal for
        //  JMT types and optional fields. If you hover of an identifier name you
        //  will see the prefix "(Opt)" applied to the documentation of the type
        //  if it is optional.
        e.CountryCode.SimpleTypeExtension.Name_ = new Jmt.XsdPrimitive.V1_0.XsdString();
        e.CountryCode.SimpleTypeExtension.Name_.Text = "UNITED STATES";

        //  Note: when we instantiate Lists, we need to do so taking into account the min/maxOccurs
        //  These values are shown when you hover on the List name itself (e.g. "[0,2]")
        e.CurrencyCodeList = new Jmt.TypeSupport.GuardedList<CurrencyCodePOSType>(0,2);
        CurrencyCodePOSType cc = new CurrencyCodePOSType();
        cc.SimpleTypeExtension.Rate = new Jmt.XsdPrimitive.V1_0.XsdString();
        cc.SimpleTypeExtension.Rate.Text = "1.23";
        cc.SimpleTypeExtension.NumberOfDecimals = new Jmt.XsdPrimitive.V1_0.Integer();
        cc.SimpleTypeExtension.NumberOfDecimals.Text = "2";
        cc.SimpleTypeExtension.Table = new Jmt.XsdPrimitive.V1_0.XsdString();
        cc.SimpleTypeExtension.Table.Text = "BSR";
        e.CurrencyCodeList.Add(cc);

        SaleInfoType.SaleInfoSeqType_.IdentificationType_.IdentificationSeqType_ id =
                e.Identification.IdentificationType_Seq.Items[0];
        id.RequestorInfo.DocStock = new Jmt.XsdPrimitive.V1_0.XsdString();
```

Figure 13 PopulateAirShoppingRQSeqItem Method

In the code fragment we can see that the individual fields, like CountryCode, are getting assigned 'valid' values. In this specific case CountryCode is formed from a SimpleType extension in the Schema and this is reflected in the Property name that is used to access the final value, 'Text', SimpleTypeExtension. It is important to note how the 'valid' values where discovered at this point in the testing workflow. In fact the information is provided, in this case, by the helpful Schema author; see Figure 14 for how CountryCode was assigned simple a *valid* value.

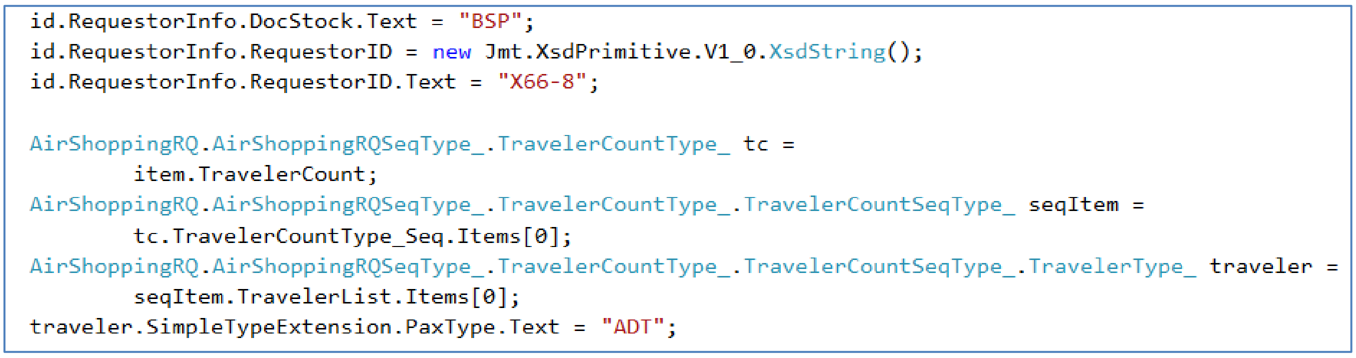


```
        //  populate the Sequence in SaleInfo (Mandatory)
        SaleInfoType.SaleInfoSeqType_ e = item.SaleInfo.SaleInfoTypeSeq.Items[0];
        e.CityCode = new SaleInfoType.SaleInfoSeqType_.CityCodeType_();
        e.CityCode.SimpleTypeExtension.Name_ = new Jmt.XsdPrimitive.V1_0.XsdString();
        e.CityCode.SimpleTypeExtension.Name_.Text = "NEW YORK CITY";

        //  Because the name field is o
        //  before attempting to set th
        //  JMT types and optional fields. If you hover of an identifier name you
        //  will see the prefix "(Opt)" applied to the documentation of the type
        //  if it is optional.
        e.CountryCode.SimpleTypeExtension.Name_ = new Jmt.XsdPrimitive.V1_0.XsdString();
        e.CountryCode.SimpleTypeExtension.Name_.Text = "UNITED STATES";
```

Jmt.XsdPrimitive.V1_0.XsdString SimpleTypeExtensionType_.Name_
(Opt) City name. Example: NEW YORK CITY

Figure 14 Assigning Valid Values – Documentation

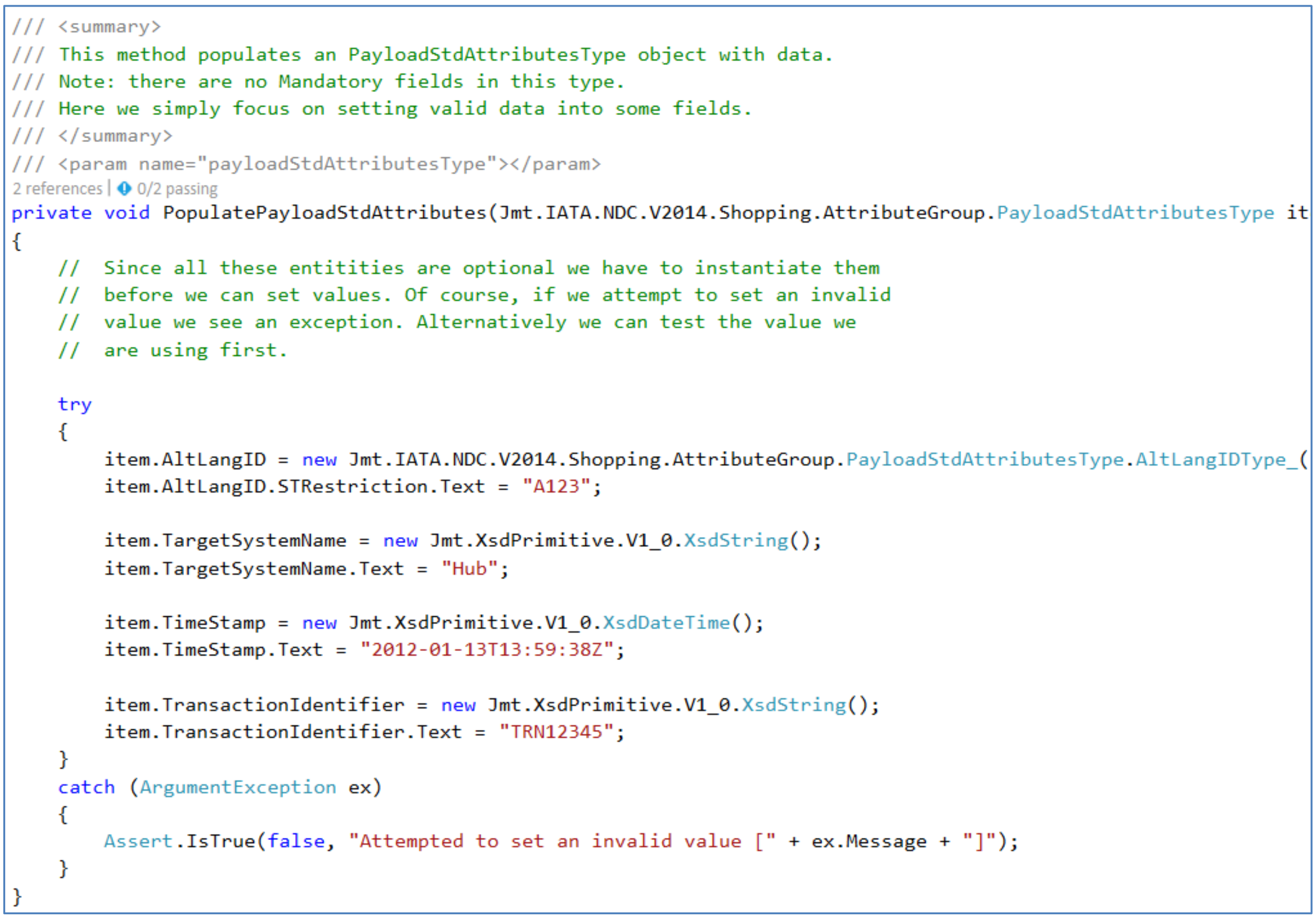
Hovering over the Property name to which the value will be assigned, shows us just what the Schema designer wanted to tell us about the data definition and what would constitute a valid value. In this case we simply take what we are shown as the value, “NEW YORK CITY”.



```
id.RequestorInfo.DocStock.Text = "BSP";
id.RequestorInfo.RequestorID = new Jmt.XsdPrimitive.V1_0.XsdString();
id.RequestorInfo.RequestorID.Text = "X66-8";

AirShoppingRQ.AirShoppingRQSeqType_.TravelerCountType_ tc =
        item.TravelerCount;
AirShoppingRQ.AirShoppingRQSeqType_.TravelerCountType_.TravelerCountSeqType_ seqItem =
        tc.TravelerCountType_Seq.Items[0];
AirShoppingRQ.AirShoppingRQSeqType_.TravelerCountType_.TravelerCountSeqType_.TravelerType_ traveler =
        seqItem.TravelerList.Items[0];
traveler.SimpleTypeExtension.PaxType.Text = "ADT";
```

Figure 15 PopulateAirShoppingRQSeqItem Method

The remaining part of the method is concerned with the RequestorInfo and TravelerCount data setting and, again, we make use of the Schema documentation shown to us via Intellisense to decide on valid values.
In the case where we populate the PayloadStdAttributes object, the method is as shown below:



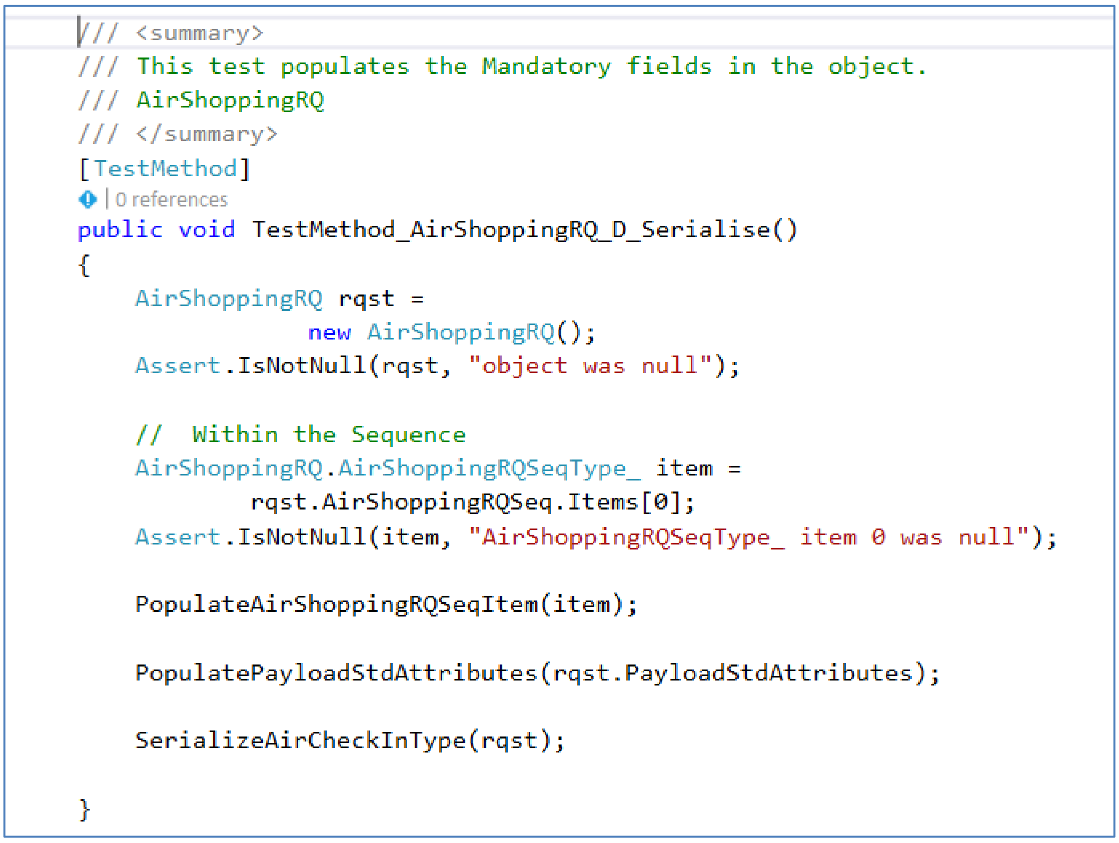
```
/// <summary>
/// This method populates an PayloadStdAttributesType object with data.
/// Note: there are no Mandatory fields in this type.
/// Here we simply focus on setting valid data into some fields.
/// </summary>
/// <param name="payloadStdAttributesType"></param>
2 references | 0/2 passing
private void PopulatePayloadStdAttributes(Jmt.IATA.NDC.V2014.Shopping.AttributeGroup.PayloadStdAttributesType it
{
    //  Since all these entitities are optional we have to instantiate them
    //  before we can set values. Of course, if we attempt to set an invalid
    //  value we see an exception. Alternatively we can test the value we
    //  are using first.

    try
    {
        item.AltLangID = new Jmt.IATA.NDC.V2014.Shopping.AttributeGroup.PayloadStdAttributesType.AltLangIDType_(
        item.AltLangID.STRestriction.Text = "A123";

        item.TargetSystemName = new Jmt.XsdPrimitive.V1_0.XsdString();
        item.TargetSystemName.Text = "Hub";

        item.TimeStamp = new Jmt.XsdPrimitive.V1_0.XsdDateTime();
        item.TimeStamp.Text = "2012-01-13T13:59:38Z";

        item.TransactionIdentifier = new Jmt.XsdPrimitive.V1_0.XsdString();
        item.TransactionIdentifier.Text = "TRN12345";
    }
    catch (ArgumentException ex)
    {
        Assert.IsTrue(false, "Attempted to set an invalid value [" + ex.Message + "]");
    }
}
```

Figure 16 PopulatePayloadStdAttributes Method

The main thing to note in this method is that the try/catch approach to handling the setting of invalid values is demonstrated. In fact, none of the values we show here are invalid, so the exception that is built into JMT SimpleTypes for invalid value alerting, is actually thrown. Another noteworthy feature of this test, is the timestamp value we use, matching exactly the W3C format for an XSD DateTime type.

AirShoppingRQ_D_Serialise

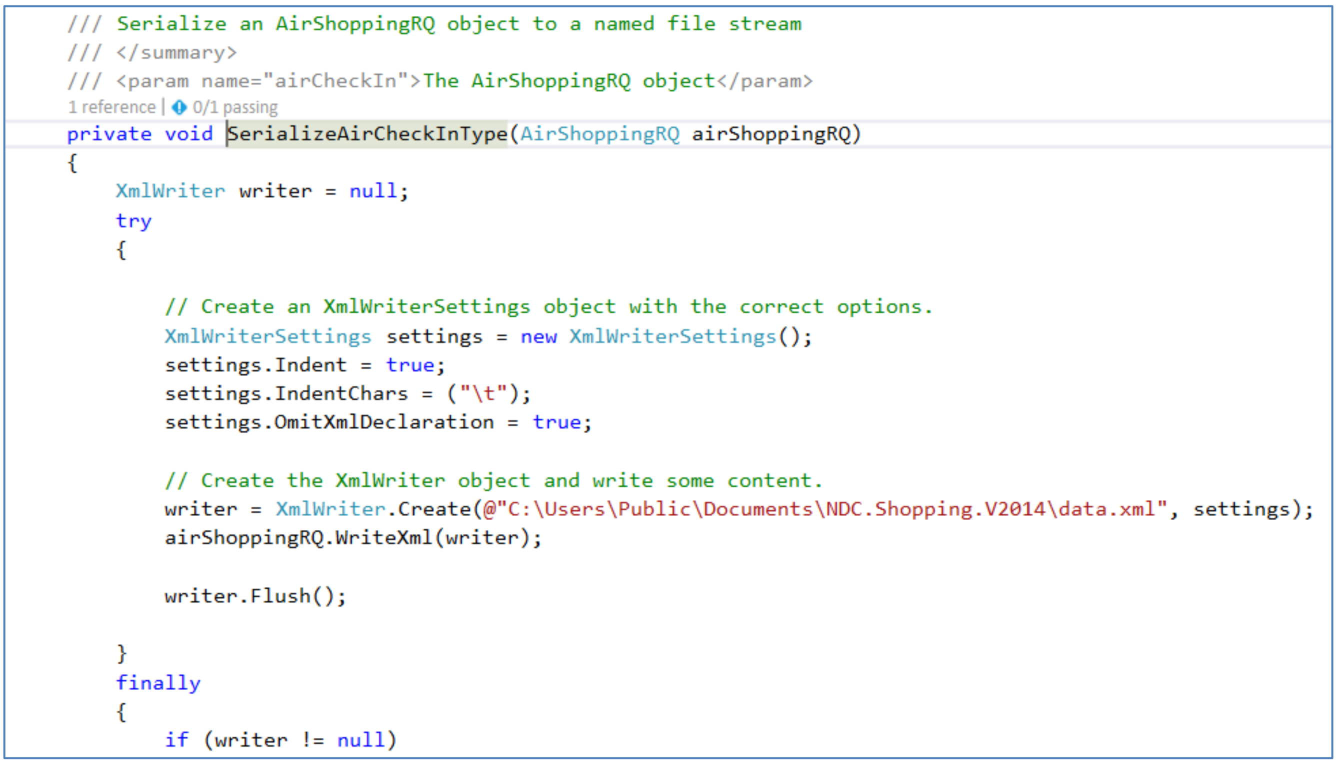
Once we have an object instantiated and populated, we can test how it is serialized to XML, this we do as shown below, Figure 17 :



```
/// <summary>
/// This test populates the Mandatory fields in the object.
/// AirShoppingRQ
/// </summary>
[TestMethod]
| 0 references
public void TestMethod_AirShoppingRQ_D_Serialise()
{
    AirShoppingRQ rqst =
                new AirShoppingRQ();
    Assert.IsNotNull(rqst, "object was null");

    //  Within the Sequence
    AirShoppingRQ.AirShoppingRQSeqType_ item =
            rqst.AirShoppingRQSeq.Items[0];
    Assert.IsNotNull(item, "AirShoppingRQSeqType_ item 0 was null");

    PopulateAirShoppingRQSeqItem(item);

    PopulatePayloadStdAttributes(rqst.PayloadStdAttributes);

    SerializeAirCheckInType(rqst);

}
```

Figure 17 AirShoppingRQ Serialise Test Method

The main part is as we saw for the populate test, it's just the last method call that does the work of getting the object to serialize. This method is as shown below:



```
/// Serialize an AirShoppingRQ object to a named file stream
/// </summary>
/// <param name="airCheckIn">The AirShoppingRQ object</param>
1 reference | 0/1 passing
private void SerializeAirCheckInType(AirShoppingRQ airShoppingRQ)
{
    XmlWriter writer = null;
    try
    {

        // Create an XmlWriterSettings object with the correct options.
        XmlWriterSettings settings = new XmlWriterSettings();
        settings.Indent = true;
        settings.IndentChars = ("\t");
        settings.OmitXmlDeclaration = true;

        // Create the XmlWriter object and write some content.
        writer = XmlWriter.Create(@"C:\Users\Public\Documents\NDC.Shopping.V2014\data.xml", settings);
        airShoppingRQ.WriteXml(writer);

        writer.Flush();

    }
    finally
    {
        if (writer != null)
```

Figure 18 SerialiseAirCheckInType Method

As in the populate test, it can be seen that we don't really 'test' here as we simply use .Net calls to set up an XmlWriter stream and call the WriteXml method of the AirShoppingRQ object. This call results in the following file being produced:

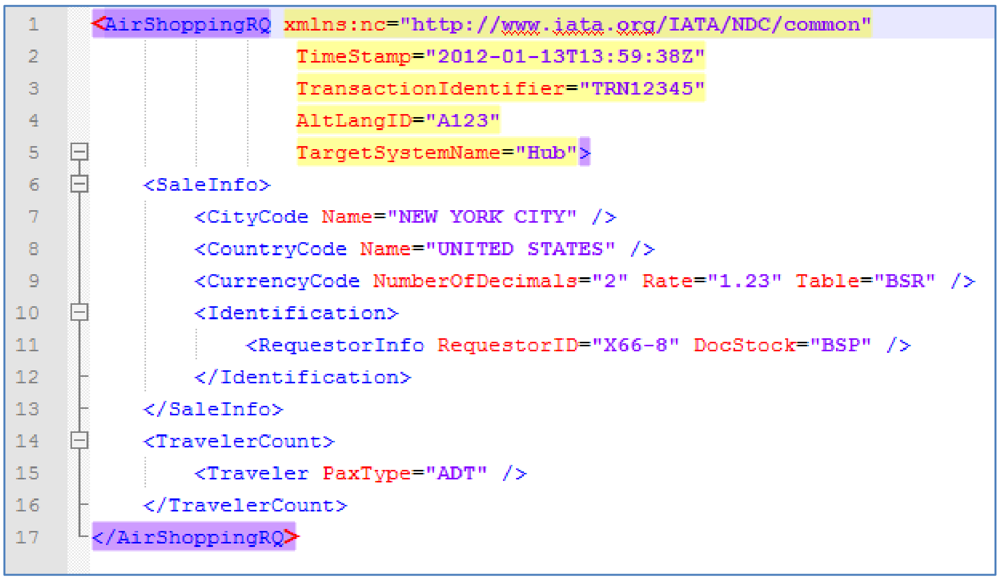

```
<AirShoppingRQ xmlns:nc="http://www.iata.org/IATA/NDC/common"
               TimeStamp="2012-01-13T13:59:38Z"
               TransactionIdentifier="TRN12345"
               AltLangID="A123"
               TargetSystemName="Hub">
    <SaleInfo>
        <CityCode Name="NEW YORK CITY" />
        <CountryCode Name="UNITED STATES" />
        <CurrencyCode NumberOfDecimals="2" Rate="1.23" Table="BSR" />
        <Identification>
            <RequestorInfo RequestorID="X66-8" DocStock="BSP" />
        </Identification>
    </SaleInfo>
    <TravelerCount>
        <Traveler PaxType="ADT" />
    </TravelerCount>
</AirShoppingRQ>
```

Figure 19 Serialisation output (data.xml)

Summary

What have we achieved at this point?
We have demonstrated in a real-world NDC/Airline Shopping type definition, how JMT types are used by a developer. In particular, the following should be noted:

- how easy it is to link the development environment to the JMT type library
- the fluent structure of referencing type structure in JMT types and how they reflect fully the Schema structure
- the way in which the developer gets actual Schema documentation at the time of writing code
- how code completion plays a key role in writing code involving JMT types
- how serialization is reduced to a simple method call

In practice the workflow of a travel application involves an orchestration of message sending as well as taking appropriate action based on received information. Using JMT types in such a scenario would dramatically speed development and since the full type landscape is provided, the full range of possibilities can be handled. More focus can be placed on this workflow/business aspect that has hitherto been possible. This must, in itself, improve the project outcome as well as the fulfillment of the project developers working with Schema- based type definitions.

Jet Messaging Technologies AG
Rotwandstrasse 35, 8004 Zurich,
Switzerland
Phone +41 79 303 05 63

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