Introduction

It's one to have a library of *strong-types* which have high-fidelity with the definitions and means in a Schema, but such a type landscape needs to be useful to a developer in crafting a business application. Of course, one central benefit to the developer of a message-related business application is that those types which represent 'messages' can be *serialised* to a form consistent with the messaging channel to be used, for example, an XML structure packaged into a SOAP container. However, prior to the serialisation of a type in order to send it somewhere, it must be appropriately populated with data. Typically, types which represent messages are quite complex structures and, as defined by the Schema definition, involve mandatory and optional parts which have business meaning. Often the construction of such complex business-related messages involves a user entering data via a User Interface of some sort and the code-behind (the View-Model) gathering such input into the final composition, the message.

How the JMT Library types assist in this compositional task is the central theme of this technical note. This note should be read in conjunction with those providing insights into the basic strong type aspects of the JMT types [1,2].

Intellisense & Code-Completion

Anecdotal evidence strongly suggests that developers faced with using a new API[3] get up to speed *in the first instance* by using a combination of three elements:

1. Intellisense
2. Code Completion
3. Example code

The first two of the above bind very clearly to the properties of the types involved in the new landscape, whilst the latter is a separate deliverable from the API vendor.

In the case of the JMT Library, Intellisense and Code Completion have been very central topics when designing the types. However, it should also be noted that a subtle, but key part of (1) & (2) above is that of showing the XML documentation of types as the developer writes code. This form of hinting also helps the developer understand the types that are in his focus at any point. For JMT types this documentation comes *directly from the Schema definition*, if the schema writer has described clearly the types in the definition then this knowledge is shown to the developer.

Let us take a concrete example; Taking a Schema as shown in Figure 1, drawn from the HealthCare domain, and processing it, we would see the corresponding JMT class as shown in Figure 2.

[1] Strong Typing Travel.doc – discusses strong typing in the context of the Travel domain
[2] Strong Typing Finance.doc - discusses strong typing in the context of the Finance domain
[3] API – Application Programming Interface


```
<xs:element name = "Record">
    <xs:annotation>
        <xs:documentation>This represents a Record for a drug item and contains Administration
        and Public information as well as detailed information on published abstracts
        related to the drug.</xs:documentation>
    </xs:annotation>
    <xs:complexType>
        <xs:sequence>
            <xs:element name = "AdminSection">
                <xs:annotation>
                    <xs:documentation>
                        This represents the Administration information related to a named drug. It contains the
                        details of Copyright and Originator as well as the Category and Archive number for the
                        main product information.
                    </xs:documentation>
                </xs:annotation>
                <xs:complexType>
                    <xs:sequence>
                        <xs:element name = "Record">
                            <xs:complexType>
                                <xs:sequence>
                                    <xs:element name = "Copyright" type = "xs:string"></xs:element>
                                    <xs:element name = "Originator" type = "xs:string">
                                        </xs:element>
                                    <xs:element name = "ArchiveNumber" type = "xs:integer">
```

Figure 1 Example Schema (fragment)

```
/// <summary>
/// This represents a Record for a drug item and contains Administration and
/// Public information as well as detailed information on published abstracts
/// related to the drug.
/// </summary>
[System.CodeDom.Compiler.GeneratedCodeAttribute("CodeConstruction", "2.0.0")]
[System.SerializableAttribute()]
[System.Diagnostics.DebuggerStepThroughAttribute()]
[System.ComponentModel.DesignerCategoryAttribute("code")]
[System.Xml.Serialization.XmlRootAttribute(Namespace = "Jetstream.Technologies", ElementName = "recordType", IsNullable = true)]
public partial class Record : ElementBase, ICloneable, ISerializable
{

    //  This represents a Record for a drug item and contains Administration and Public information as well as detailed information on published abstracts related to the drug.
    private string _documentationField;

    private string xmlTag = "recordType";

    private GuardedList<RecordSeqType_> _recordSeqField;

    private Integer _uIDField;

    public Record() :
            base()
    {
        this._documentationField = "This represents a Record for a drug item and contains Administration" +
                                   "and Public information as well as detailed information on published" +
                                   "abstracts related to the drug.";
```

Figure 2 C# Code for Class (fragment)

To be noted in these figures is that the documentation in the Schema is retained in the Class, both as a `<summary>` to the class as well as a member field (`this._documentationField`) which is accessible via a Property. This feature is typical of all library classes.

If we were to now write code using the above class (defined by Schema), following the namespace appropriate to the schema provider, we would see as shown in Figure 3, below:

```
[TestMethod]
public void TestMethod_BasicIntellisenseAndDocumentation()
{
    Jmt.HealthCare.CRec.v1_0.Element.
}
```

DescriptorList
Record

class Jmt.HealthCare.CRec.v1_0.Element.Record
This represents a Record for a drug item and contains Administration and Public information as well as detailed information on published abstracts related to the drug.

Figure 3 Intellisense at Class-level

Here it can be clearly seen how the types existing in the library namespace are displayed and the one selected actually shows to the developer the schema writers source annotation, which we may presume as meaningful in the business domain.

If we have the situation as shown in Figure 3 and the developer presses TAB, the outcome would be an auto-completion step, as shown in Figure 4:

```
[TestMethod]
public void TestMethod_BasicIntellisenseAndDocumentation()
{
    Jmt.HealthCare.CRec.v1_0.Element.Record
}
```

Figure 4 Auto-Completion at Class Level

If the developer now enters a period ('.'), we would see the members of the class `Record` as shown in the figure below:

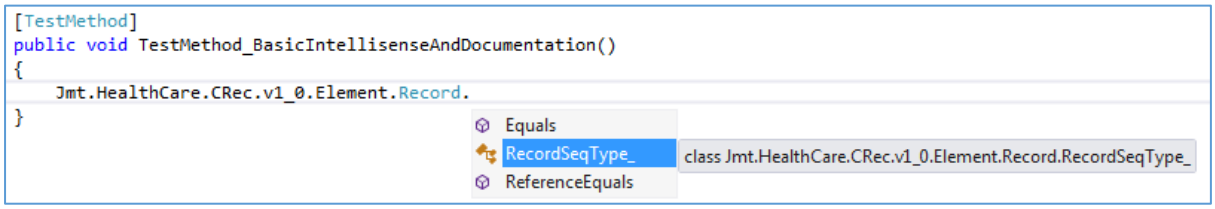

Figure 5 Intellisense at Member Level

In this case the Intellisense list shows us that the content of Record is a synthetic type (not defined explicitly in the source Schema) which reflects a `<sequence>` of objects (which *is* as defined in the source Schema). If the developer continues with this 'browsing' strategy, then the outcome as shown below:

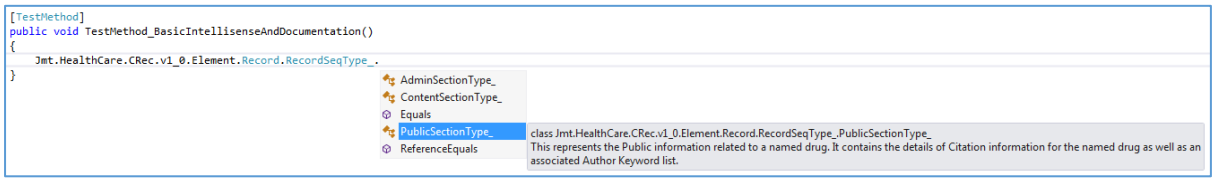

Figure 6 Intellisense for Defined Type

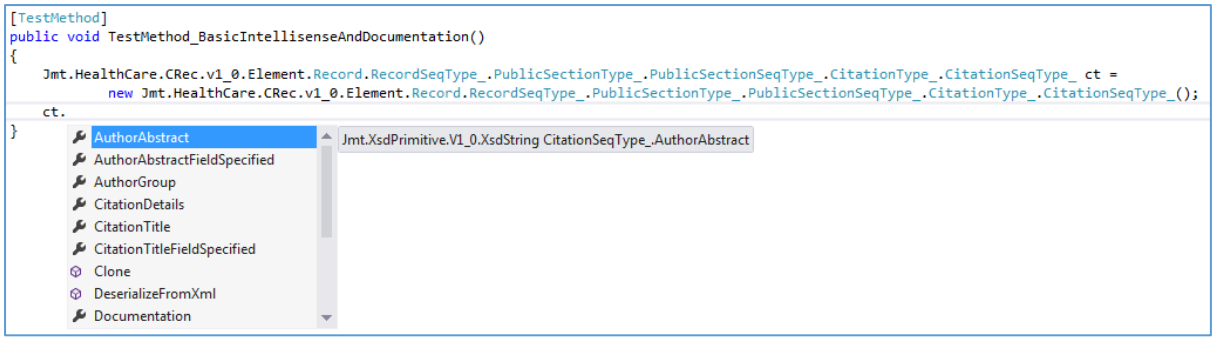

Figure 7 Intellisense for Class Members

It should be noted that in Figure 7 the lack of documentation is because the source Schema lacks this information.

Building an Application

Of course, as noted in the Introduction, Intellisense and auto-completion play a big part in the easy adoption by developers of a new API or object framework. However, once the focus shifts from basic programming tasks to one where a real business application is to be developed, then some other aspects of the JMT Library emerge to help that process also.

Consider the proof-of-concept application, developed in simple Windows Forms technology which is to provide the business functionality that allows a user to "search for a low fare" in the domain of Air Travel, send a message and get an appropriate response from a provider.

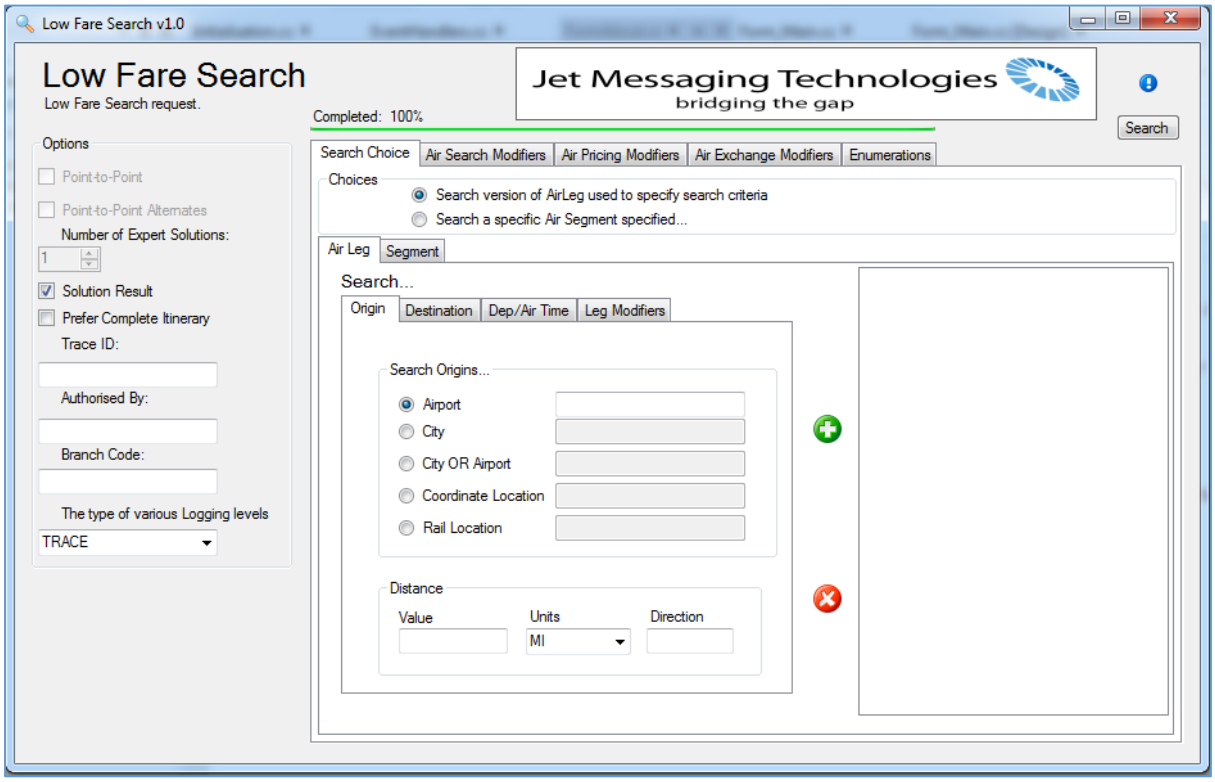

Figure 8 Application UI for LowFareSearch

It is important to remark here that the application is not intended to exemplify best practice as far as UI design is concerned, a commercial application might well decide to position itself on the web and be developed using an appropriate technology framework there, such as Silverlight or WPF[4]. The only purpose of this application is to explore the beneficial patterns that emerge for the developer when using JMT strong types.

In this sample application, the structure of the various tab Panels and Radio Button groups reflect the structure of the Schema that define the '`LowFareSearch`' message, fragments of which are shown in the figures following:

[4] WPF – Windows Presentation Foundation


```
<xs:element name="LowFareSearchReq">
    <xs:annotation>
        <xs:documentation>
            Low Fare Search request.
        </xs:documentation>
    </xs:annotation>
    <xs:complexType>
        <xs:complexContent>
            <xs:extension base="BaseLowFareSearchReq">
            </xs:extension>
        </xs:complexContent>
    </xs:complexType>
</xs:element>
```

Figure 9 Schema Request for LowFareSearch

```
<xs:complexType name="BaseLowFareSearchReq">
    <xs:annotation>
        <xs:documentation>
            Base Low Fare Search Request
        </xs:documentation>
    </xs:annotation>
    <xs:complexContent>
        <xs:extension base="AirSearchReq">
            <xs:sequence>
                <xs:element ref="common:SearchPassenger" maxOccurs="18" >
                    <xs:annotation>
                        <xs:documentation> Maxinumber of passenger increased in to 18 to support 9 INF passenger along with 9 ADT,CHD,INS                    passenger</xs:documentation>
                    </xs:annotation>
                </xs:element>
                <xs:element ref="AirPricingModifiers" minOccurs="0" />
                <xs:element ref="Enumeration" minOccurs="0" />
                <xs:element ref="common:PointOfSale" minOccurs="0" maxOccurs="5" />
                <xs:element ref="AirExchangeModifiers" minOccurs="0" />
            </xs:sequence>
            <xs:attribute name="EnablePointToPointSearch" type="xs:boolean" use="optional" default="false">
                <xs:annotation>
                    <xs:documentation>Indicates that low cost providers should be queried for top connection options and the results returned with the search.</xs:documentation>
                </xs:annotation>
            </xs:attribute>
            <xs:attribute name="EnablePointToPointAlternates" type="xs:boolean" use="optional" default="false">
                <xs:annotation>
                    <xs:documentation>Indicates that suggestions for alternate connection cities for low cost providers should be returned with the search.</xs:documentation>
                </xs:annotation>
            </xs:attribute>
            <xs:attribute name="MaxNumberOfExpertSolutions" type="xs:integer" use="optional" default="0">
                <xs:annotation>
                    <xs:documentation>Indicates the Maximum Number of Expert Solutions to be returned from the Knowledge Base for the provided search criteria</xs:documentation>
                </xs:annotation>
            </xs:attribute>
            <xs:attribute name="SolutionResult" type="xs:boolean" use="optional" default="true">
                <xs:annotation>
                    <xs:documentation>Indicates whether the response will contain Solution result (AirPricingSolution) or Non Solution Result (AirPricingPoints).
                    The default value is true.
                    This attribute cannot be combined with EnablePointToPointSearch, EnablePointToPointAlternates and MaxNumberOfExpertSolutions.</xs:documentation>
                </xs:annotation>
            </xs:attribute>
            <xs:attribute name="PreferCompleteItinerary" type="xs:boolean" use="optional" default="true">
                <xs:annotation>
                    <xs:documentation>This attribute is only supported for ACH .It works in conjunction with the @SolutionResult flag </xs:documentation>
                </xs:annotation>
            </xs:attribute>
        </xs:extension>
    </xs:complexContent>
</xs:complexType>
```

Figure 10 Schema Base Definition for LowFareSearch

Considering Figure 10 above, it can be seen how the top-level `<attributes>` form the left hand set of information elements in the User Interface (UI), whilst the `<element>` sequence is reflected in the right hand part. The individual elements in this `<sequence>` are complex and involve choices for the user when entering data.

The following topics are exemplified in this example, and will be discussed individually in following sections:

- How complex data compositions can be formed leveraging the JMT type characteristics
- How the JMT types can be used in normal C# programming idioms such as type comparison
- How a UI can be embellished with Schema-provided documentation
- How JMT types can be used in Generic class designs which facilitate application architecture
- How JMT types can be reflected over which facilitate application architecture
- How JMT types can have the normal class extension approach applied to them
- How JMT can help in tracking data entry completeness for a complex data structure

Complex Data Compositions

As can be sensed from the above schema and application UI views, the data composition for LowFareSearch is considerable. It is in just such a complex situation that the developer needs to have a reliable type landscape for the entities in play. In addition, however, there needs to be a strategy for building, composing the eventual object that in this case represents a message to be sent to a communicating service from where, it can be presumed, the LowFareSearch result (response) will emerge.

In this application the various groupings of data naturally relate to the individual `TabPage` controls. So, as an example of development tactic, these individual groupings, which are in fact strong types, are retained in the (so-called) `Tag` object of the `TabPage` objects themselves. The figure below, illustrates this approach:

```
/// <summary>
/// Initialise the Tab Page entitled: Origin
/// </summary>
/// <param name="form">The main form object</param>
/// <param name="tp">The TabPage object</param>
private static void InitialiseTabPageAirLegOrigin(Form form, TabPage originTabPage)
{
    Jmt.[redacted].v4_1_0_69.TypeSearchLocationType tslt =
            new Jmt.[redacted].v4_1_0_69.TypeSearchLocationType();
    // put the overall TypeSearchLocation in the Tag of the Origin TabPage
    originTabPage.Tag = (object)tslt;
```

Figure 11 JMT Object Persistence

Here we see the object of type `TypeSearchLocationType` (a JMT type, in a provider namespace) being instantiated and then saved in the `Tag` of the `TabPage` which contains all the data entry controls appropriate to the type, `originTabPage`. With this strategy the full composition, assembling as it does to the message type, can be assembled in structured manner.

Consistent Validation

This approach to instantiating and persisting a real strong type in a control Tag also extends to the other controls such as `TextBox` and `ComboBox`. This has the key benefit that we can apply the Schema-defined validation rules to the values entered at the UI. For example, within the right-hand side of the UI pictured in Figure 8, we see the fields, TextBox controls in the group entitled “Search Origin” (see Figure 12, below).

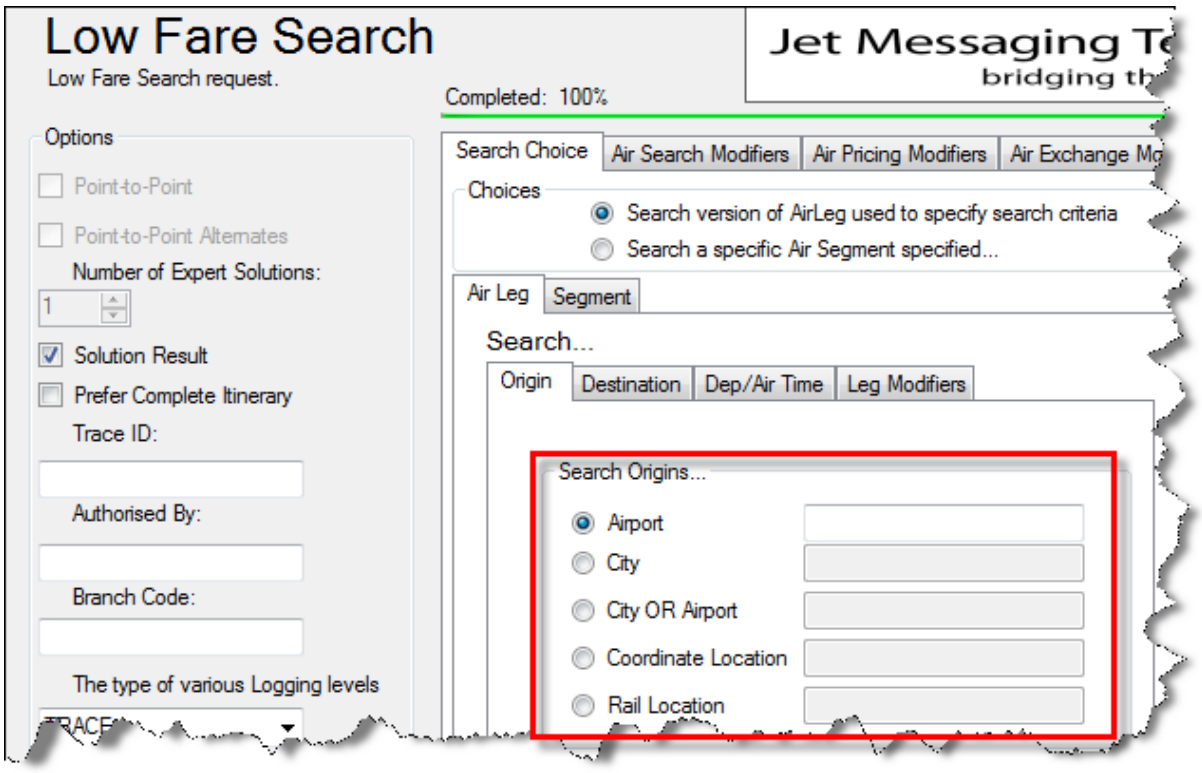

Figure 12 Search Origins Group

In the initialisation phase for our application, the specific types representing the data in each of these `TextBox` controls (comprising a `<choice>`) were saved in the Tag of the individual controls, for example:


```
// ..... City
{
    Jmt.[redacted].v4_1_0_69.Element.City city =
        Tools.Choice.GetChoiceElementObjectByName<Jmt.[redacted].v4_1_0_69.Element.City>("City", choiceList);
    System.Windows.Forms.RadioButton rb =
        (System.Windows.Forms.RadioButton)originGroupBox.Controls.Find("radioButton_SL_City", true)[0];
    Tools.ToolTips.ConfigureToolTip(new ToolTip(), rb,
        Tools.DocumentationRetriever<Jmt.TravelPort.UAPI.v4_1_0_69.Element.City>.GetDocumentation(city,
                                                    "Origin is a City"));
    System.Windows.Forms.TextBox cityTB =
        (System.Windows.Forms.TextBox)originGroupBox.Controls.Find("textBox_CityValue", true)[0];
    cityTB.Tag = (object)city;
}

// ..... City OR Airport
{
    Jmt.[redacted].v4_1_0_69.Element.CityOrAirport cityOrAirport =
        Tools.Choice.GetChoiceElementObjectByName<Jmt.[redacted].v4_1_0_69.Element.CityOrAirport>("CityOrAirport", choiceList);
    System.Windows.Forms.RadioButton rb =
        (System.Windows.Forms.RadioButton)originGroupBox.Controls.Find("radioButton_SL_CityOrAirport", true)[0];
    Tools.ToolTips.ConfigureToolTip(new ToolTip(), rb,
        Tools.DocumentationRetriever<Jmt.[redacted].v4_1_0_69.Element.CityOrAirport>.GetDocumentation(cityOrAirport,
                                                    "Origin is a City OR Airport"));
    System.Windows.Forms.TextBox cityOrAirportTB =
        (System.Windows.Forms.TextBox)originGroupBox.Controls.Find("textBox_CityOrAirportValue", true)[0];
    cityOrAirportTB.Tag = (object)cityOrAirport;
}

// ..... Coordinate Location
{
    Jmt.[redacted].v4_1_0_69.Element.CoordinateLocation coordinateLocation =
        Tools.Choice.GetChoiceElementObjectByName<Jmt.[redacted].v4_1_0_69.Element.CoordinateLocation>("CoordinateLocation", choiceList);
    System.Windows.Forms.RadioButton rb =
        (System.Windows.Forms.RadioButton)originGroupBox.Controls.Find("radioButton_SL_CoordinateLocation", true)[0];
    Tools.ToolTips.ConfigureToolTip(new ToolTip(), rb,
        Tools.DocumentationRetriever<Jmt.[redacted].v4_1_0_69.Element.CoordinateLocation>.GetDocumentation(coordinateLocation,
                                                    "Origin is specified by its coordinates"));
    System.Windows.Forms.TextBox coordinateLocationTB =
        (System.Windows.Forms.TextBox)originGroupBox.Controls.Find("textBox_CoordLocationValue", true)[0];
    coordinateLocationTB.Tag = (object)coordinateLocation;
}

// ..... Rail Location
{
    Jmt.[redacted].v4_1_0_69.Element.RailLocation railLocation =
        Tools.Choice.GetChoiceElementObjectByName<Jmt.[redacted].v4_1_0_69.Element.RailLocation>("RailLocation", choiceList);
    System.Windows.Forms.RadioButton rb =
        (System.Windows.Forms.RadioButton)originGroupBox.Controls.Find("radioButton_SL_RailLocation", true)[0];
    Tools.ToolTips.ConfigureToolTip(new ToolTip(), rb,
        Tools.DocumentationRetriever<Jmt.[redacted].v4_1_0_69.Element.RailLocation>.GetDocumentation(railLocation,
```

Figure 13 Initialising Group Controls

Highlighted here, we are instantiating the JMT types `City`, `CityOrAirport` and `CoordinateLocation` and assigning them to the respective UI controls.

Now, when the `TextChanged` event fires for whichever specific control is entering data, i.e. the content entered in the TextBox changes, we make direct use of the underlying type which represents a definition from the Schema to validate the entered data. Specifically, because in this example we think of a UI, when the entered data is invalid we show an appropriately decorated `ErrorProvider`. To improve the velocity of development, this example uses a single event handler for these controls, this event handler we see in Figure 14.

It is also important to note that the pattern applied to validation is very regular and this assists the developer in achieving consistent, quality outcome for the overall application.


```
/// <summary>
/// Handle all the TextBox Text Changed events here.
/// (We might think of a better organisation)
/// </summary>
/// <param name="sender"></param>
/// <param name="e"></param>
private void textBox_TextChanged(object sender, EventArgs e)
{
    //  we know its a text box who is sending
    TextBox tb = (TextBox)sender;
    if (string.Compare(tb.Name, "textBox_AirportValue", false) == 0)
    {
        //  associate this TB with an appropriate type, the strongly
        //  typed JMT object is in the Tag of the TextBox
        Jmt.[redacted].v4_1_0_69.Element.Airport airport =
                (Jmt.[redacted].v4_1_0_69.Element.Airport)tb.Tag;
        try
        {
            //  TODO text here is within AirportCode
            airport.Code.Text = tb.Text;
            errorProvider_Main.SetError(tb, String.Empty);
        }
        catch (ArgumentException)
        {
            errorProvider_Main.SetError(tb,
                                        "The Value entered is not valid");
        }
    }
    else if (string.Compare(tb.Name, "textBox_CityValue", false) == 0)
    {
        //  associate this TB with an appropriate type, the strongly
        //  typed JMT object is in the Tag of the TextBox
        Jmt.[redacted].v4_1_0_69.Element.City city =
                (Jmt.[redacted].v4_1_0_69.Element.City)tb.Tag;
        try
        {
            city.Code.Text = tb.Text;
            errorProvider_Main.SetError(tb, String.Empty);
        }
        catch (ArgumentException)
        {
            errorProvider_Main.SetError(tb,
                                        "The Value entered is not valid");
        }
    }
    else if (string.Compare(tb.Name, "textBox_CityOrAirportValue", false) == 0)
    {
        //  associate this TB with an appropriate type, the strongly
        //  typed JMT object is in the Tag of the TextBox
        Jmt.[redacted].v4_1_0_69.Element.CityOrAirport cityOrAirport =
                (Jmt.[redacted].v4_1_0_69.Element.CityOrAirport)tb.Tag;
        try
        {
```

Figure 14 Validating Control Data

C# Programming Idioms

As the JMT types are strong types, we can use them as with any other C#/.Net type in programming idioms that can be very helpful to the application developer in achieving a maintainable quality design with high velocity.

In this example, the Schema demands that the final message can comprise a <sequence> of entities such as `Origin`, `Destination`, `Dep/AirTime` and `LegModifiers`, as portrayed in the UI of Figure 15 below:

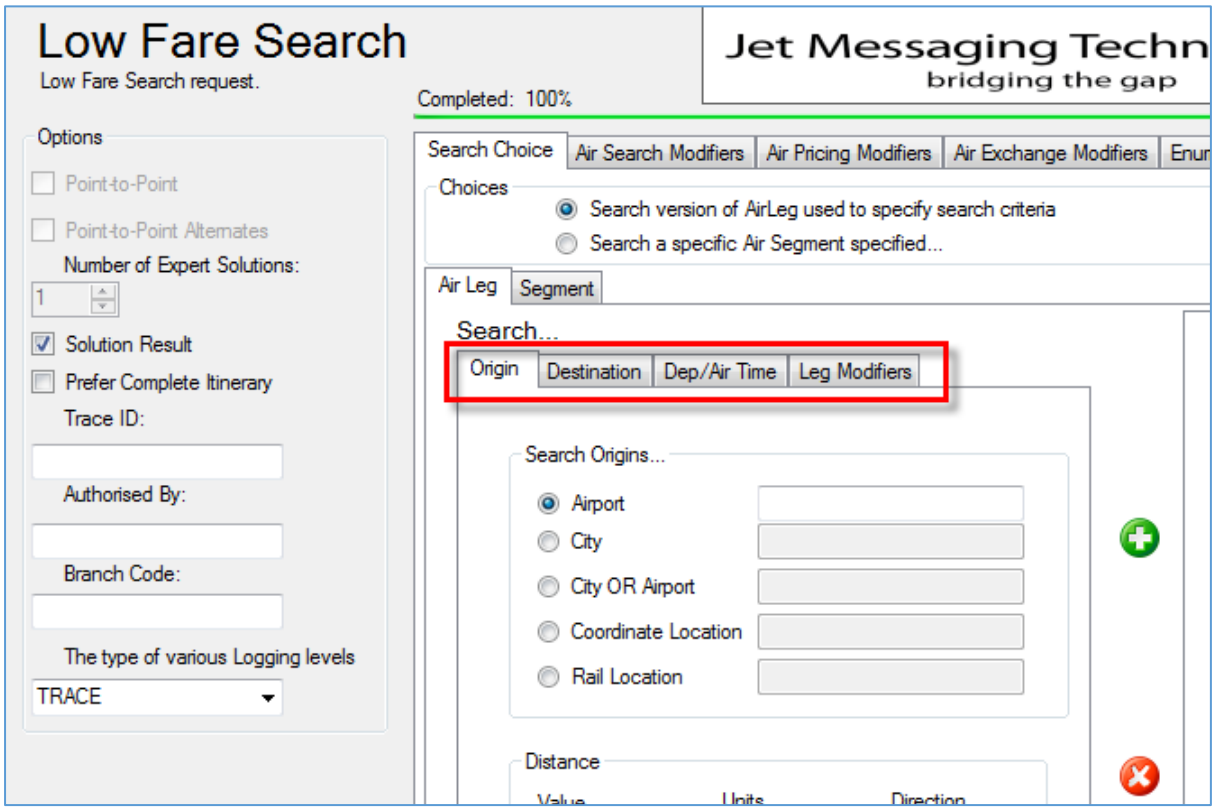

Figure 15 Air Leg Components

In this example, to compose the list of individual entities, there are the "+"/"x" controls to the right of the Air Leg/Search Origins Group. The (+) adds the current data (type) to the overall sequence, whilst the (x) removes a previously added data item. In the application we need to be efficient when adding such data items to the collection that represents the `<sequence>`. This efficiency is achieved by using the C# idiom as exemplified in Figure 16 , below:


```
/// <summary>
/// Handler for the click event of the '+' sign - move data on left to the
/// list on the right.
/// </summary>
/// <param name="sender">The sender object</param>
/// <param name="e">The event args object</param>
private void pictureBoxPlus_Click(object sender, EventArgs e)
{
    //  The '+' sign pictue has been clicked
    //  Assemble the data for the currently selected Air Leg
    //  inner TabPage.
    //  Would rather like not to use the TabPage Name to match on here, as
    //  we would need to re-code if we translated the app to a different
    //  language; German, French etc...
    System.Windows.Forms.TabPage tp = this.tabControl_AirLeg.SelectedTab;
    if ( string.Compare(tp.Name, "tabPage_AirLeg_SearchOrigin", false) == 0  &&
         tp.Tag is Jmt.[redacted].v4_1_0_69.TypeSearchLocationType)
    {
        //  Air Leg > Search Origin
        Jmt.[redacted].v4_1_0_69.TypeSearchLocationType tslt =
                (Jmt.[redacted].v4_1_0_69.TypeSearchLocationType)tp.Tag;
        //  here we would like to compose a line that identifies the element by a small
        //  fragment of its data - "left to the interested reader"
        //  TODO compose a better element entry in list
        this.listBox_AirLeg.Items.Add("Search Origin element [" + string.Format("{0,2}", airLegListBoxCounter + "]"));
        airLegListBoxCounter++;
    }
```

Figure 16 C# Type Idiom

Here, similarly to how the `TextBox` event handler was coded, we use a single event handler to capture the event that fires when (in this case) the "+" sign is clicked by the user. The key idiom here is that of being able to silently evaluate the type of the (otherwise neutral, object) `Tag` of the `TabPage`. Specifically, following the test of which `TabPage` raised the event (by name, with caveat as shown in the code comment) we silently test if the `Tag` of the `TabPage` is of the correct type (a JMT type). Once this is established we proceed to add (in this example, a rather simple) element to the list representing the Schema `<sequence>`.

Being able to interact with types in this way, just like any other type is extremely helpful to the developer who wants to maximise their focus on the development of the business application.

UI Embellishment

Labels and tooltips (see Fig 17 and 18)

In the design of the UI, a number of textual elements, `Labels`, have been initialised at design-time with generic text. If we look at the design surface for the main form of the application, Figure 17 below, the highlighted elements indicate these specific `Labels`.

The intention, for this example, was to demonstrate how source Schema information can be used to good effect in decorating the UI and helping the user understand the controls and groupings to be found there.

At initialisation-time, when the application starts, for example, the <annotation> for the overall LowFarSearch class is retrieved and set as the text shown in the figure blow as initially having the content "`label_ST`". The control itself in named `label_Subtitle` and the code to perform this initialisation is shown in Figure 18

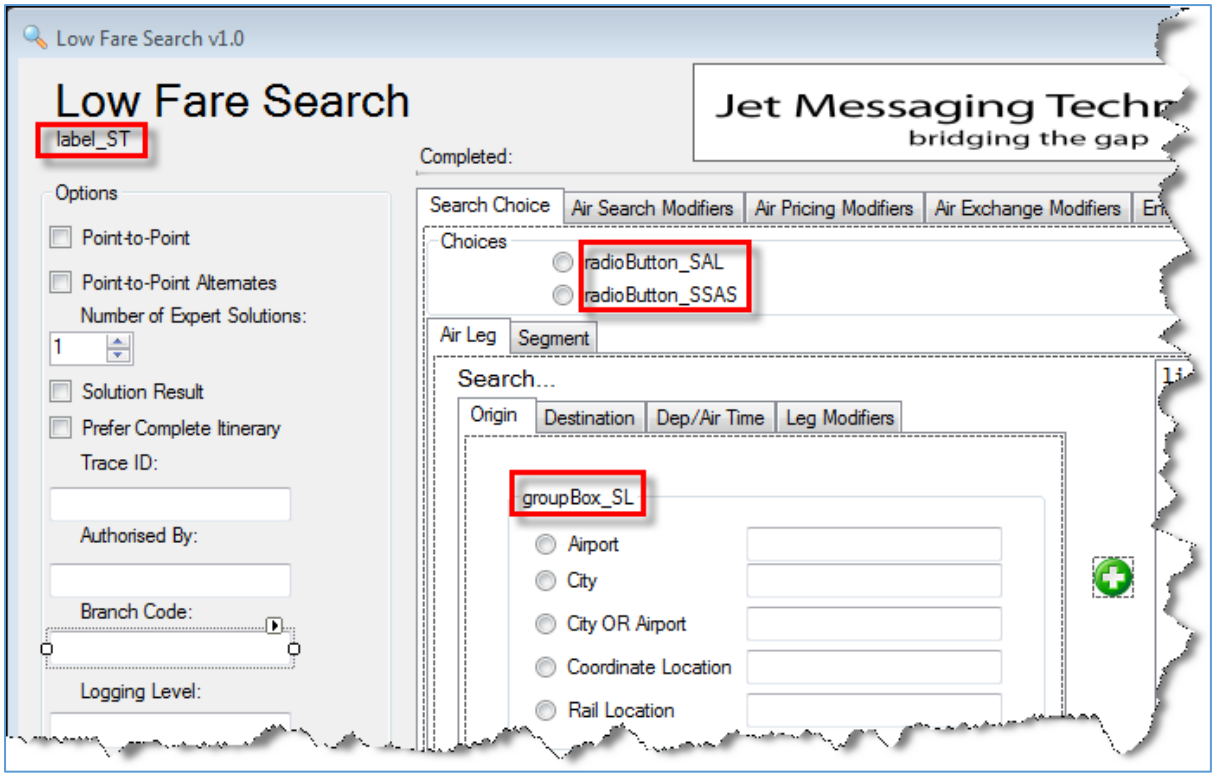

Figure 17 Design-Time Labels

```
public Form_Main()
{
    InitializeComponent();

    //  provide the schema documentation as a subtitle
    this.label_Subtitle.Text =
        Tools.DocumentationRetriever<Jmt.[redacted].v4_1_0_69.Element.LowFareSearchReq>.GetDocumentation(lfsr,
                                                "Search today for the best Fares going...");
```

Figure 18 Setting Label Text from Schema

The method `GetDocumentation()` is a helper which retrieves the documentation from a specific JMT type (specified in the invocation as the Generic type reference), this method will be introduced in the next section. When the application starts the UI eventually looks like as shown in below Figure 19.

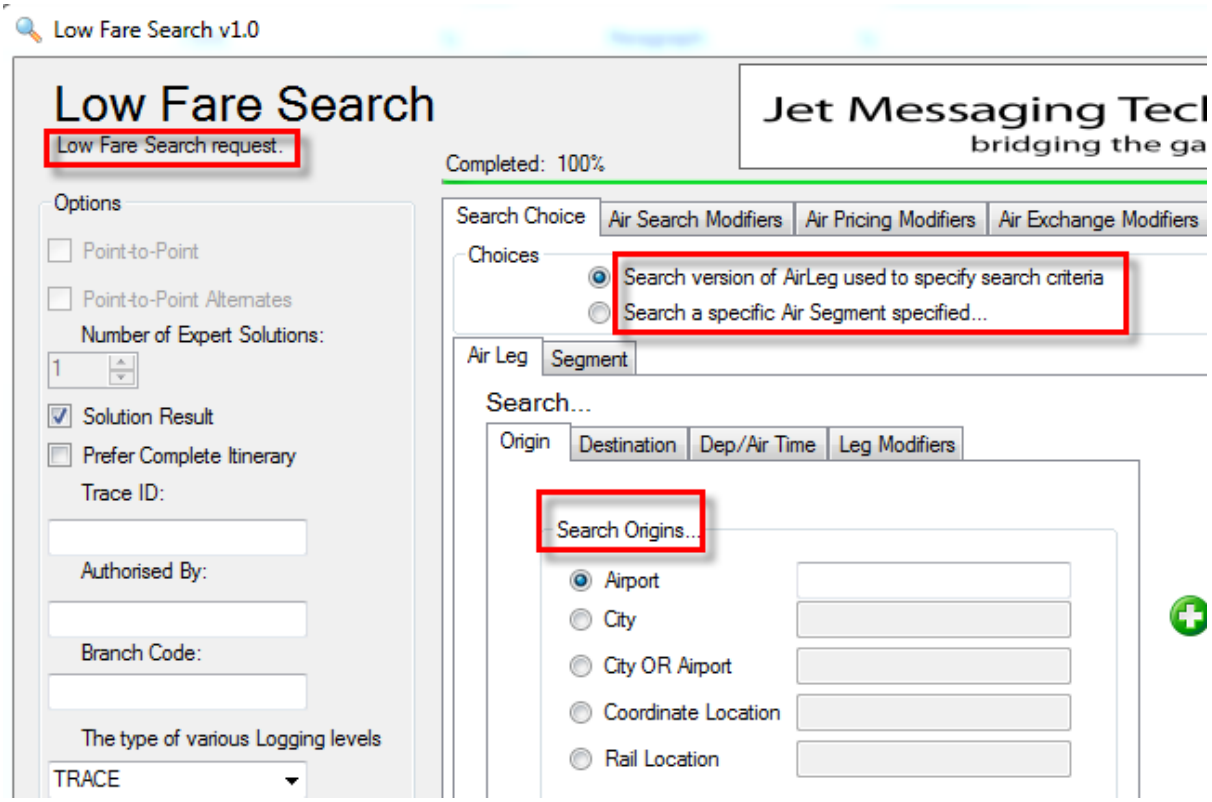

Figure 19 Labels at Runtime

It should be noted that the helper methods used here are hardened to cater for the situation that the type specified has no documentation, i.e. the source Schema does not define any.

The intention of this example is to illustrate how this schema information can be retrieved and used to decorate a UI, the specific labels used here are by no means the limit of such a strategy. It is possible that this approach could be extended to retrieve documentation for a specific language, e.g. Italian, English, German and so on.

Generic Class Design

As noted above, in the case of type documentation, use is made of Generic Classes and Methods. In the case of the documentation retrieval pattern discussed in the previous section, the key component is a Generic class that accepts JMT types from which the documentation is extracted. The key point here, as well as the use of Generic types, is that the JMT types retain the documentation defined by the schema designer.

If we look at a fragment of the code of this documentation retrieval method, Figure 20, we can get a sense of how it works. The generic type T, is a JMT type reference, and an object of such a type is passed as a parameter, along with a default text string to be adopted if the source Schema has no documentation defined. Provided the generic type is not a Value Type, then an object of the specific (JMT) type is created and its member collection reflected over. Since *all* JMT types have either a "`Documentation`" or "`DocumentationList`" Property, this is specifically located within the collection. Invoking the appropriate Property returns the required documentation string. In the case where no documentation is defined the default string is adopted.

Here we see the JMT types being used in a strong reflection situation which allows for a well-structured and accessible way to get the all-important information about types which was defined in the Schema and provides business information relevant to application design.


```
/// <summary>
/// Retrieve the documentation for a specific strong type, T.
/// We provide a default if no Schema documentation is provided.
/// </summary>
/// <param name="obj">The strong-typed object containing a documentation
/// Property/Method holding the Schema-defined documentation</param>
/// <param name="deflt">A default documentation if no schema-defined
/// form has been defined</param>
/// <returns>An appropriate documentation string</returns>
public static string GetDocumentation(T obj, string deflt)
{
    string ret = deflt;
    if (obj == null)
    {
        return deflt;
    }

    if (typeof(T) == typeof(String)) return deflt;
    if (typeof(T).IsValueType || typeof(T).FullName == "System.String")
    {
        return deflt;
    }
    else
    {
        T inst = Activator.CreateInstance<T>();
        //MethodInfo callInfo = typeof(ClassA).GetMethod(args[0], param_types);
        PropertyInfo pInfo = typeof(T).GetProperty("Documentation");

        if (pInfo == null)
        {
            MethodInfo mInfo1 = typeof(T).GetMethod("DocumentationList");
            if (mInfo1 != null)
            {
                ReadOnlyCollection<string> collection =
                        (ReadOnlyCollection<string>)mInfo1.Invoke(inst, new object[] { });
                ret = MassageSchemaDocumentationString(collection[0]);
            }
            else
            {
                ret = deflt;
            }
        }
        else
        {
            MethodInfo mInfo2 = pInfo.GetGetMethod();
            string s = (string)mInfo2.Invoke(inst, new object[] { });
            ret = MassageSchemaDocumentationString(s);
        }

        if (ret.Length == 0)
        {
            ret = deflt;
        }
```

Figure 20 Documentation Retrieval

Reflection

As we saw in the previous section, the JMT types can be used in situations where general object creation and reflection is used. In the case of type documentation we also relied on the JMT types having *known Properties and Methods*.

In LowFareSearch, as well as reflecting to find the all-important documentation data, there are a number of other helper methods, for example, to enable the application developer to retrieve the optional or mandatory fields in a type. The pattern here also relies on the consistent way in which JMT types are constructed and how optional/mandatory fields are signalled.

In Figure 21, a fragment of the Type Inspector code is shown, in this case to discover the Optional fields in a JMT type, as specified in the Schema:

```
/// <summary>
/// This method allows a developer to get a list of all the Optional
/// fields by their name.
/// </summary>
/// <remarks>
/// We look for all public Properties whose name ends in "FieldSpecified>.
/// The prefix to this part is the actual name of the field in the type.
/// </remarks>
/// <typeparam name="T">The type of the strong JMT type to
/// be examined</typeparam>
/// <param name="obj">A candidate object to be examined</param>
/// <returns>A list of type names, being the optional set</returns>
public static List<string> OptionalFieldNames<T>(T obj)
{
    List<string> list = new List<string>();

    if (typeof(T) == typeof(String)) return list;
    if (typeof(T).IsValueType || typeof(T).FullName == "System.String")
    {
        return list;
    }
    else
    {
        T inst = Activator.CreateInstance<T>();
        PropertyInfo[] pInfo = typeof(T).GetProperties();

        if (pInfo == null  ||  pInfo.Length == 0)
        {
            return list;
        }
        else
        {
            string name;
            foreach (PropertyInfo info in pInfo)
            {
                if (info.Name.EndsWith("FieldSpecified"))
                {
                    name = info.Name.Substring(0, info.Name.Length - 14);
                    list.Add(name);
                }
            }
        }
    }

    return list;
}
```

Figure 21 Optional Field Discovery

Data Entry Completeness

As discussed above, one of the patterns explored in the LowFareSearch demonstration application is that of assembling the overall 'message' using persisted objects, the inner elements of composition, that are bound to `TabPages`. Although, at the time of writing, incomplete, the target here is to provide an application developer faced with a complex composition of data such as existing in this application, a means to signal to the user, who is in fact the one entering the data, that all the mandatory items have been entered (and as we have noted above, validated as correct to the Schema definition). At present a placeholder signal is displayed above the right-hand side `TabPage` set, the green `ProgressBar` as shown below in Figure 22

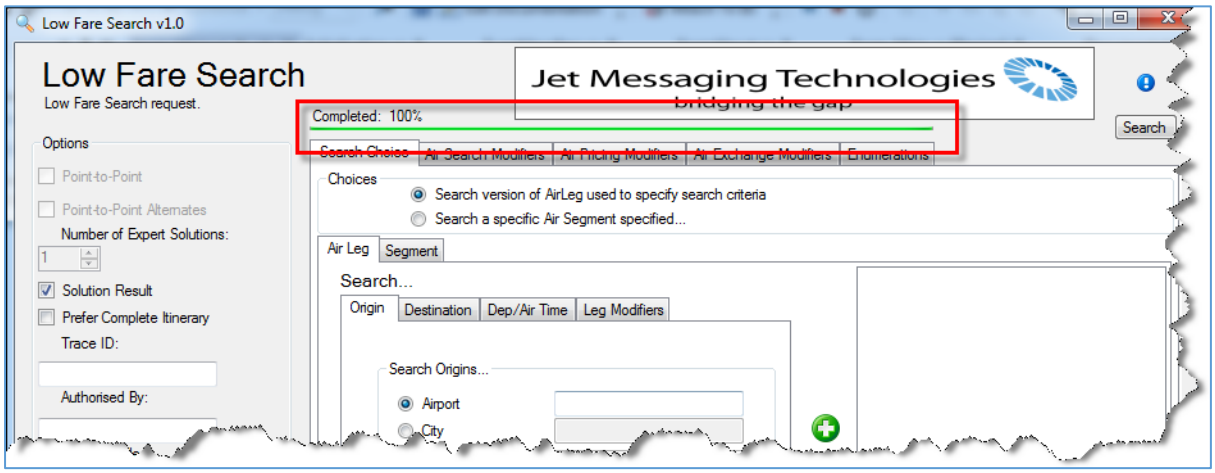

Figure 22 Placeholder Completion Status Signal

It is anticipated that events would be signalled by the Completion Manager component to subscribing listeners so that, for example the `ProgressBar` could be finalised as well as the "Search" button enabled.

JMT Documentation

The JMT Library deliverables include extensive documentation, covering both the classes themselves and their related Schema definitions. This documentation is shown in the following figures:

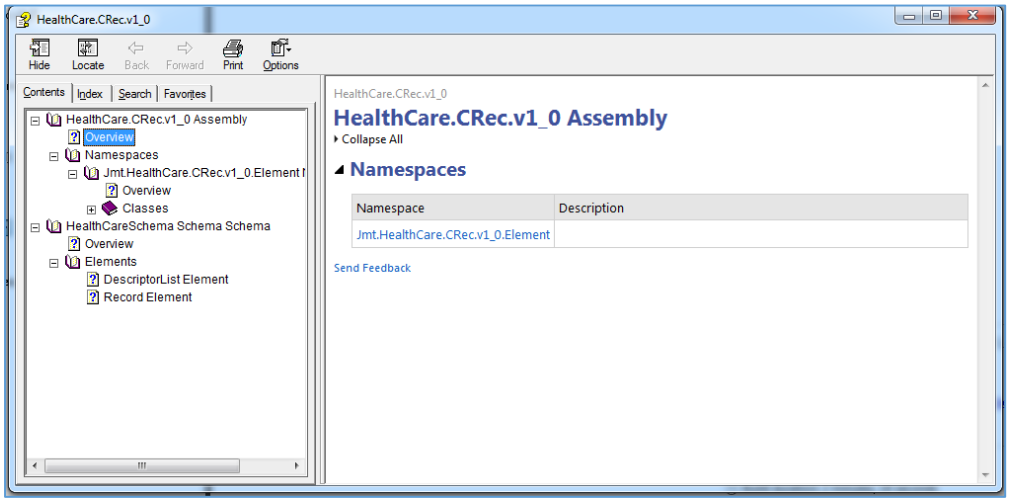

Figure 23 Top-level Documentation Sample

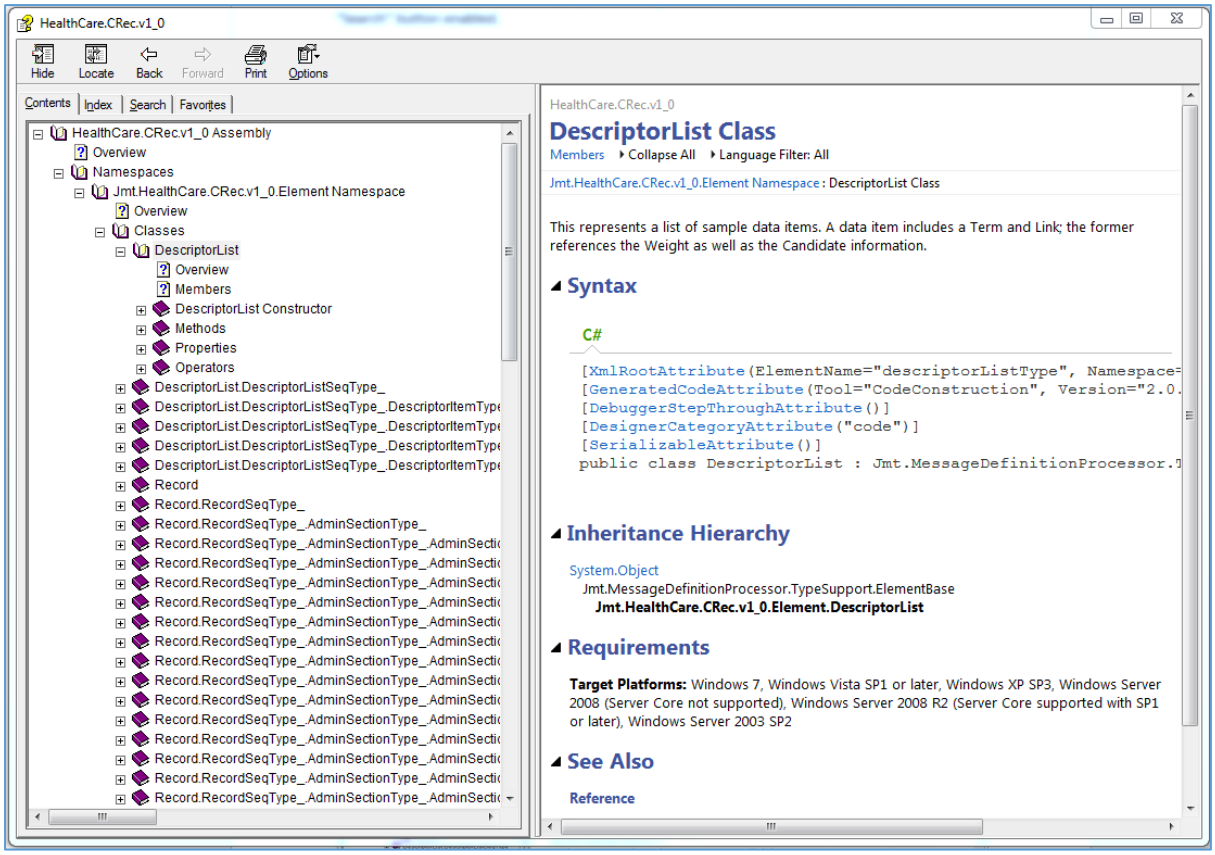

Figure 24 Class-level Documentation

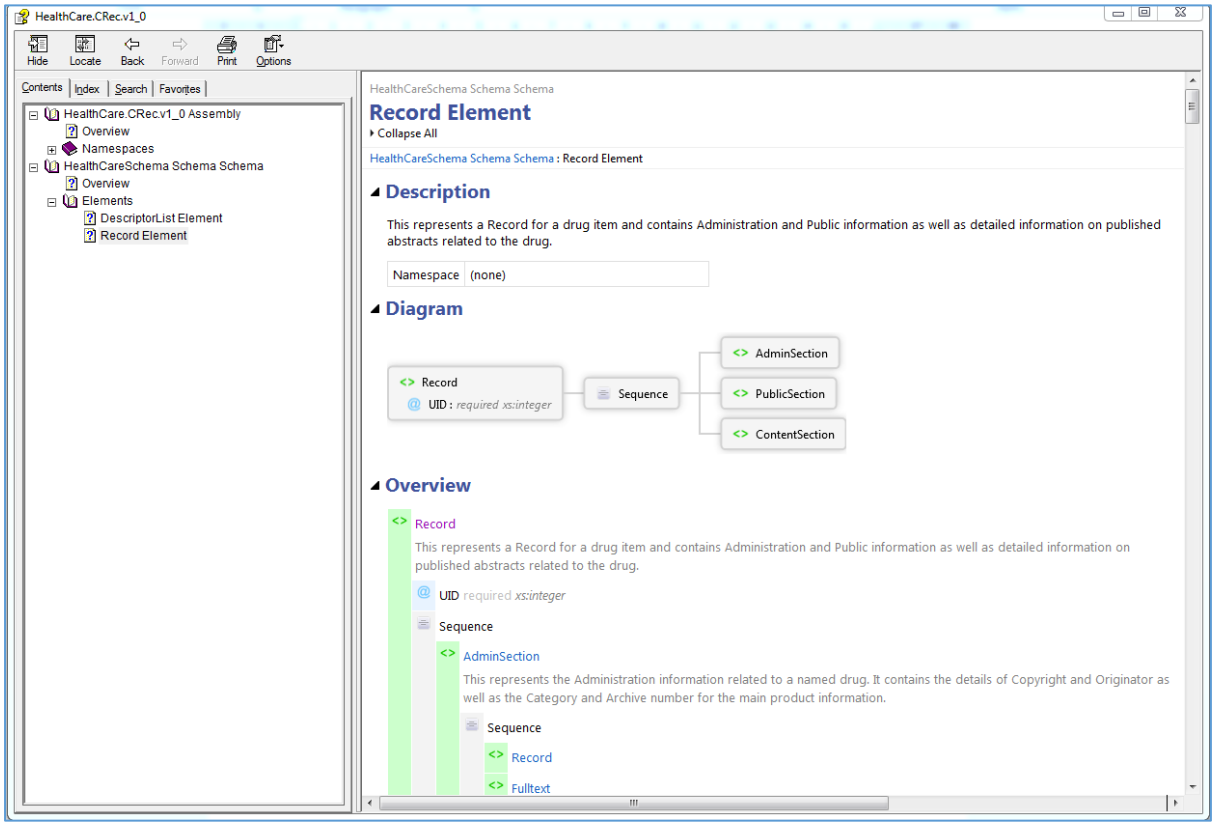

Figure 25 Schema-level Documentation

JMT Examples

As part of the JMT Library deliverables a range of examples will be provided. These will exemplify key concepts in using the JMT types, from ‘getting started’ to ‘advanced’ level.

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

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