COMPASS 3.X.X CUSTOM CODE PORTING GUIDE

This guide provides an overview of what needs to be done to port custom code from Compass 2.x.x/DCF 3.3.x to Compass 3.x.x/DCF 3.4.x. It is not intended to be an exhaustive list or step by step instructions on porting code.

This guide is intended for internal LBS usage and for use by LBS customers who have written their own custom extensions.

There are two classes of interfaces on Compass:

- DCF based filters that are executed by DCF, and
- Compass extensions such as Job Actions, Post Processing Actions, Rule Match Actions, Batch File Ingestors, Disk Folder De-encapsulators, DIMSE message relayers, etc.

1 USING STATEMENTS

You will need to adjust your using statements.

You must remove references to `LaurelBridge.DCS`, `LaurelBridge.DDS`, `Laurelridge.DIS`, and others
For DCF 3.4.x you will mostly need to use `LaurelBridge.DCF`.
Occasionally you may need to use `LaurelBridge.AppFramework` but those are special cases

Below is an example of the typical `using` statements needed by a custom DCF filter being used in Compass 3.x.x. Your specific `using` statements may be different.

Note: Make sure to include '`using LaurelBridge.DCF.Filters;`' if you are working on a filter.

```
//REF System.Core.dll
using System;
using System.Collections.Generic;
using System.Linq;
using LaurelBridge.Compass.Core;
using LaurelBridge.DCF;
using LaurelBridge.DCF.Configuration;
using LaurelBridge.DCF.Dicom;
using LaurelBridge.DCF.Dicom.Dimse;
using LaurelBridge.DCF.Dicom.Elements;
using LaurelBridge.DCF.Dicom.Query;
using LaurelBridge.DCF.Dicom.Worklist;
using LaurelBridge.DCF.Filters;
using LaurelBridge.DCF.IO;
using LaurelBridge.DCF.Logging;
using LaurelBridge.DCF.Networking;
using System.ComponentModel;
```

2 NAMESPACE

Internally we typically use the namespace `LaurelBridge.Custom`. You should be able to use any namespace you wish for your custom code.

If you are presently using `LaurelBridge.Compass.X`, please stop using it and switch to something that does not have `Compass` as a part of the name.

3 LOGGING

Logging needs to be done a little differently depending on whether you are using a DCF filter or a Compass extension.

3.1 DCF FILTER

An important thing to make sure you do when logging from custom filters is to pass the `DicomSessionSettings (dss)` object to all your logging calls. This will ensure that log message will be written to the correct session log. If you pass in a null `DicomSessionSettings` object (for example, into a static method or static initializer), then any log messages using the null `DicomSessionSettings` will get written to the generic `CompassService` log.

You will need the following `using` statements:

```
using LaurelBridge.DCF.Logging;
using Flags = LaurelBridge.DCF.Logging.LogDebugFlags;
```

You will need an instance of the `LogManager`, and we recommend creating a readonly string to hold the name of the filter.

```
public class DemographicsUpdateNavigatorInvokerJobFilter : IFilterAction
{
        // Static Logger instance
        private static readonly ILogger _logger = LogManager.GetCurrentClassLogger();
        private static readonly string FilterActionName = "DemographicsUpdateNavigatorInvokerJobFilter: ";
```

The following code fragment shows the first few lines of `ApplyAction`. Notice that we are using a `Debug` log statement and we are also setting up several `tagMarkers`.

Make sure you add `tagMarkers` for any tags this filter may change that you want DCF filter logging to automatically log.

```
public void ApplyAction(CFGGroup config, DicomSessionSettings dss, RelevantTagMarker tagMarker, ref DicomDataSet
dds)
{
        //Write a debug message if the current flags passed are for FilterSummary.
        _logger.Debug(Flags.FilterSummary, dss, FilterActionName + "In ApplyAction...");

        tagMarker(Tags.PatientName.ToString());
        tagMarker(Tags.PatientID.ToString());
```

You can log error messages like this:

```
_logger.Error(dss, e, FilterActionName + "Exception encountered updating demographics in ApplyAction");
```

There are also other log methods available such as `Info`, `DebugFormat`, `ErrorFomat`. It is recommended you use the `_logger.Debug` and `_logger.Error` methods most of the time. `DebugFormat` and `ErrorFormat` can be used if you need to log objects whose stringify methods could have an adverse performance impact.

3.2 COMPASS EXTENSIONS

Compass extensions can only write messages into the `CompassService` log.

You need the following `using` statements:

```
using LaurelBridge.Logging;
using Flags = LaurelBridge.DCF.Logging.LogDebugFlags;
```

The following code fragment shows the first few lines of a `JobAction` definition.

```
public class MWLWorklistQuerierJobAction : IJobActionEx2, IQueryListener
{
        private static readonly ILog _logger = LogManager.GetCurrentClassLogger(ComponentId.Application);
        private static readonly string JobActionName = "MWLWorklistQuerierJobAction: ";
```

To log a message, you will want to use the `CompassLogDebugFlags.ExecuteJobAction` debug flags so that your debugging messages will get logged if they are turned on by the Compass operator person.

```
_logger.DebugFormat((long)CompassLogDebugFlags.ExecuteJobAction, JobActionName + "MWL query failed", e);
```

You may want to 'OR' together other debugging flags depending on the situation. For example, here are logging flags that could be useful with Data Integration Post Processing Actions so that debug messages are logged if any of the flags are set by the user in the Compass GUI.

```
private const long DEBUG_FLAGS = (long)(CompassLogDebugFlags.ExecuteJobAction |
CompassLogDebugFlags.FluencyJobAction | CompassLogDebugFlags.PowerScribe360CustomFieldsJobAction);
```

Then when you need log a debug message you would do this:

```
_logger.Debug(DEBUG_FLAGS, COMPONENT_NAME + "atStrValue falling back to srDoubleValue..");
```

3.2.1 Turn on DIMSE logging from a Job action

Sometimes you need to turn on logging on a MWL or Q/R connection created in code from a job action such as the case when you are writing a job action that does a Q/R to PACS to get demographic updates. You'll want to do something like the following, which is basically saying that if logging for `ExecuteJobAction` has been enabled by the Compass user, then we should turn on the specific list of DCF Debug flags and OR them together. There could be other flags that could be useful, but the sample below turns on the most common ones: ACSE PDUs, DIMSE Messages, and filter logging.

The '`config`' in the below example is the CompassConfiguration object that is available from the `IJobActionEx2` interface.

```
DicomSessionSettings querySessionSettings = new DicomSessionSettings();

//Compass config from IJobActionEx2 interface
if (config.Settings.LogDebugFlags.HasFlag(CompassLogDebugFlags.ExecuteJobAction))
{
        querySessionSettings.DebugFlags |= (Flags.DimseRead | Flags.DimseWrite | Flags.FilterVerbose |
Flags.AcsePdu);
}
        // create our query client
        QRSCU queryScu = new QRSCU(ainfo, querySessionSettings);
```

3.3 DICOM handling classes and methods

This is not an exhaustive list but examples of some of the most common things you may encounter. A lot of method names have gotten capitalized. Most classes and methods you use to manipulate DICOM Data sets and DICOM Elements have this change.

For example:

`DicomFileInput.readDataSet(…)` is now: `DicomFileInput.ReadDataSet`

`DicomDataSet.removeElement(…)` is now: `DicomDataSet.RemoveElement`.

One change that may catch you off guard is this change to `DicomElementFactory`:

```
DicomElement  retval = DicomElementFactory.create(DCM.E_ACCESSION_NUMBER, accessionNumber);
```

became

```
// generate the new element
DicomElement retval = ElementFactory.Create(Tags.AccessionNumber, accessionNumber);
```

Another commonly used feature that has changed is that the accessor methods for setting up an `AssociationInfo` have become properties. For example:

```
// setup the query parameters
AssociationInfo ainfo = new AssociationInfo();
ainfo.calledPresentationAddress(QUERY_HOST + ":" + QUERY_PORT);
ainfo.callingTitle(QUERY_CALLING_AE_TITLE);
ainfo.calledTitle(QUERY_CALLED_AE_TITLE);
```

becomes

```
// setup the query parameters
AssociationInfo ainfo = new AssociationInfo();
ainfo.CalledPresentationAddress = QUERY_HOST + ":" + QUERY_PORT;
ainfo.CallingTitle = QUERY_CALLING_AE_TITLE;
ainfo.CalledTitle = QUERY_CALLED_AE_TITLE;
```

Some interfaces from DCF 3.3.x now have a capital letter in front of their name in DCF 3.4.x, for example:

`DicomInput` is now `IDicomInput`

`DicomDataDictionary.makeUID()` is now `DataDictionary.UidFactory.CreateUid();`

`DicomElement.getValueAsString()` is now deprecated.

You now have some added flexibility in whether you want the first value, all values as a collection, or all values as a single string delimited by the multi-valued delimeter, and whether you want internal strings trimmed.

It seems like the closest conversion to the old `getValueAsString` functionality is to use:
`DicomElement.GetValuesAsString(false)` where the Boolean tells the method whether to trim internal strings or not.

Another minor capitalization change is on `DicomFileInput`:

`DicomFileInput.ActualTSUID` is now `DicomFileInput.ActualTsUid;`

3.4 Attribute Tags

A few things have changed about handling of `AttributeTags`.

You can no longer construct them from a string, e.g., you cannot do this anymore:

```
AttributeTag tag = new AttributeTag("0010,0010");
```

Instead, you must use:

```
AttributeTag tag = AttributeTag.Parse("0010,0010");
```

However, for a performance boost, you can construct them directly from integers. The following examples produce the same tag:

```
AttributeTag tag = new AttributeTag(0x00100010);
```

And

```
AttributeTag tag = AttributeTag.Parse("0010,0010");
```

3.5 Constants for Attribute Tags and UIDs

The DCM and UID classes have been replaced. The object called `Tags` has an accessor for every tag in chapter 6 of the DICOM standard using their recommended names. Therefore, converting from the `DCM.E_*` format to the `Tags` should be straight forward.

For example:

`DCM.E_PATIENTS_NAME` becomes `Tags.PatientName`

`DCM.E_PATIENT_ID` becomes `Tags.PatientId`

UIDs have similar changes:

`UID.STUDY_ROOT_QUERYRETRIEVE_INFO_MDL_FIND` becomes `Uids.StudyRootQueryRetrieveInformationModelFIND`

And transfer syntaxes have their own additional intermediate class:

`UID.TRANSFERLITTLEENDIAN` becomes `Uids.TransferSyntax.ImplicitVRLittleEndian`

3.6 A special word about private tags

The name of the private data dictionary has changed. In Compass 3.x.x/DCF 3.4.x it is now called `ext_data_dictionary.txt`.

You may no longer need custom code that operated on private tags, especially if you had a special `removeElement.cs` filter that dropped private tags.

As of Compass 3.0.2 private tags defined in the `ext_data_dictionary.txt` file may show up in the list of available tags in filtering GUIs. Here are the known caveats:

1) The tag must not be in the private creator range.
2) It must not come after the pixel data
3) There cannot be any duplicate description text in your `ext_data_dictionary.txt` file
4) The tags cannot be inside a sequence

Most private tags should now appear in the Compass GUI's filter GUI's DICOM tag drop down controls. This is especially handy for removing specific private tags without the need for writing a custom script.

For Additional Information Contact:

Laurel Bridge Software, Inc.

support@LaurelBridge.com