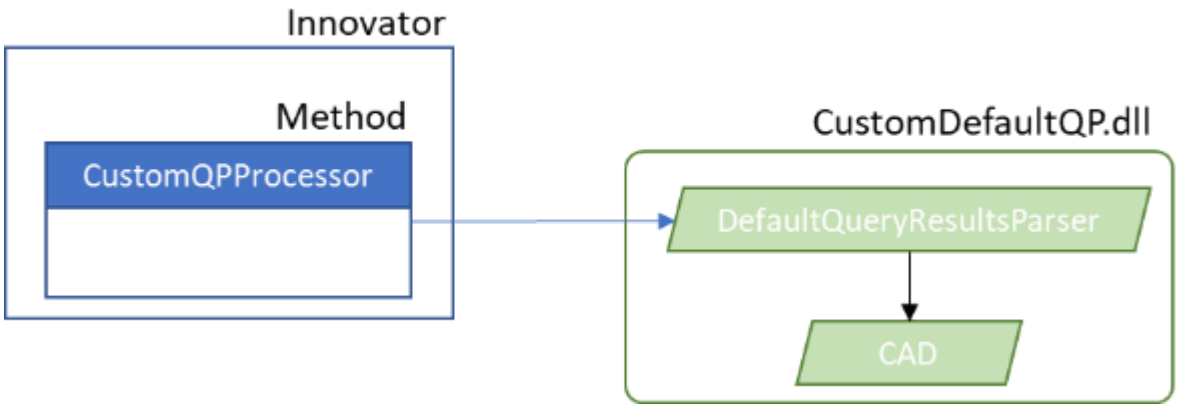
Create the Data Processor Method

The example described in this section relegates the bulk of the Query Processing logic to a separate DLL which is linked with the Aras Innovator Server. This separation of code is not necessary, but because of the amount of software that may apply to custom query processing this method of implementation is easier to manage. The actual Data Processor Method therefore is much smaller and is used mostly to validate input and make the necessary calls into the custom DLL in this case.

Important

Note: It is recommended to make methods OS agnostic for use with Linux and Windows. The main incompatibility issues in operating systems that need to be considered when implementing the method are path case-sensitivity, path separators, OS-specific line-endings, OS-specific code. For more information about cross-platform development, see section 2.3 Cross-platform development in the Aras Innovator 31 - Programmer's Guide.



The following is an example of the CustomQPProcessor Method, which is used to execute the Default Query Definition, pass the results to a custom DLL for processing, and generate the view data used for the 3D Viewer – Product Occurrences. The example is described here:

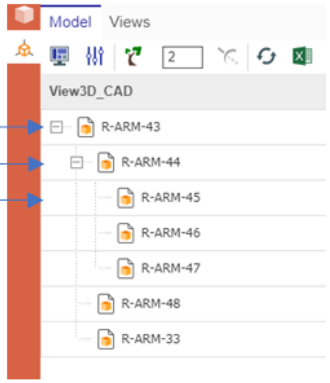
```
1 //MethodTemplateName=Csharpe:Aras.Public.Events.EventHandler<Aras.DynamicModelViewer.DataModel.DataProcessorArgs>;

2 If(eventArgs==null || !eventArgs.IsValid())
  return;

3 //Get Innovator Connection
  var connection = Innovator.getConnection();

4 //Get Query Definition Helper
  Aras.DynamicModelViewer.DataModel.Helpers.QueryDefinitionHelper qdHelper = eventArgs.QueryDefinitionHelper;

5 // Get Query Definition Item
  Item qdItem = qdHelper.GetQueryDefinitionItem(eventArgs.QueryDefinitionId);

6 // Apply Query Definition and get result
  Item qdResult = qdHelper.GetQueryDefinitionResultItem(qdItem, eventArgs.ItemId, eventArgs.QueryDefinitionParams);

7 //Custom Query Processing Class
  CustomDefaultQP.DefaultQueryResultsParser defQryResParser = new CustomDefaultQP.DefaultQueryResultsParser();

8 //Process the query response
  Var result = new Aras.DynamicModelViewer.DataModel.QueryProcessingResult();
  Result.SetProductOccurrenceList(defQryResParser.processQueryResults(qdResult));

9 //Return the results
  eventArgs.SetQueryProcessingResult(result);
```

1. This initial comment must be the first line in the Method. It is used to identify the Method template to be used.
2. The EventArgs object is passed into, and available, to this Method. This object contains The Query Definition ID, ID of the Item being viewed, the Query Definition Parameters map, and will contain the results created by the Query Processor.
3. Retrieves the Aras Innovator connection.
4. Retrieves the Query Definition Helper used for this Query Processor.
5. Retrieves the Query Definition Item.
6. Execute the Query Definition using the given Item ID and the query parameters used.
7. This example includes a custom DLL used for the query processing logic, with the main class – CustomDefaultQP.DefaultQueryResultsParser. This software, explained below, is used to parse the query results, and construct the response used to populate the 3D Dynamic/Streaming Viewer.
8. The results of the processing – Product Occurrence List – is constructed by the custom DLL and stored in the QueryProcessingResults Object which is then returned to the EventArgs Object passed into the Method (see Help files for API documentation).

Processing Results from Query Execution

The custom Query Processor example processes the results returned from the execution of the Query Definition. To illustrate this process this section will use the example query results shown in below figure and discuss each relevant XML Element when processing.

```
<Result>
  <Item alias="CAD">
    <id keyed_name="R-ARM-43">…</id>
    <Additional Item Properties…>
    <Relationships>
      <Item alias="CAD Structure">
        <Relationships>
          <Item alias="CAD Instance">
            <Additional Item Properties…>
            <transformation_matrix>…</transformation_matrix>
          </Item>
          <Item alias="CAD">
            <id keyed_name="R-ARM-44">…</id>
            <Additional Item Properties…>
            <Relationships>
              <Item alias="CAD Structure">
                <Relationships>
                  <Item alias="CAD Instance"/></Item>
                  <Item alias="CAD">
                    <id keyed_name="R-ARM-45">…</id>
                    <Relationships>
                      <Item alias="File">
                        <Relationships>
                          <Item alias="fr_Representation">
                            <Relationships>
                              <Item alias="fr_RepresentationFile">
                                <Relationships>
                                  <Item alias="File_1">
                                    <id keyed_name="J3 Spindle.scs">…</id>
                                    <filename>J3 Spindle.scs</filename>
                                  </Item>
                                </Relationships>
                              </Item>
                            </Relationships>
                          </Item>
                        </Relationships>
                      </Item>
                    </Relationships>
                  </Item>
                </Relationships>
              </Item>
            </Relationships>
          </Item>
        …
```



The above figure shows the partial XML result data when executing the Default (Base) Query Definition against a sample Assembly shown at the right. In this example, the root Assembly – R-ARM-43 – has one sub-Assembly – R-ARM-44 – and two child Components – R-ARM-48, R-ARM-33. Assembly R-ARM-44 has three Components – R-ARM-45, R-ARM-46, and R-ARM-47. The diagram highlights key XML Elements and Attributes in the XML results that are necessary when processing this data. The `<Additional Item Properties>` XML Element is notional: it

represents additional Properties included for the associated XML Element that were left out of the example for simplicity.

When implementing a Query Processor, it is necessary to process the result set such as displayed above and extract the required information. The following sections identify some points to consider for this purpose.

Aliases

Each Item in the results is represented by an `alias` attribute matching the alias value provided in the Query Definition. This is important, because the alias identifies the specific Query Item that matches the associated Item (see section Base Query Definition). There can be multiple Query Items in a Query Definition that refer to the same ItemType. The alias is the only way to distinguish the resulting query information.

Data Model Object

It is useful to collect the information extracted from the Query Results in a separate object, which can then be used to construct the Product Occurrence list. In the example shown in section, the Class `CAD` was included to store key information extracted from parsing the Query Results.