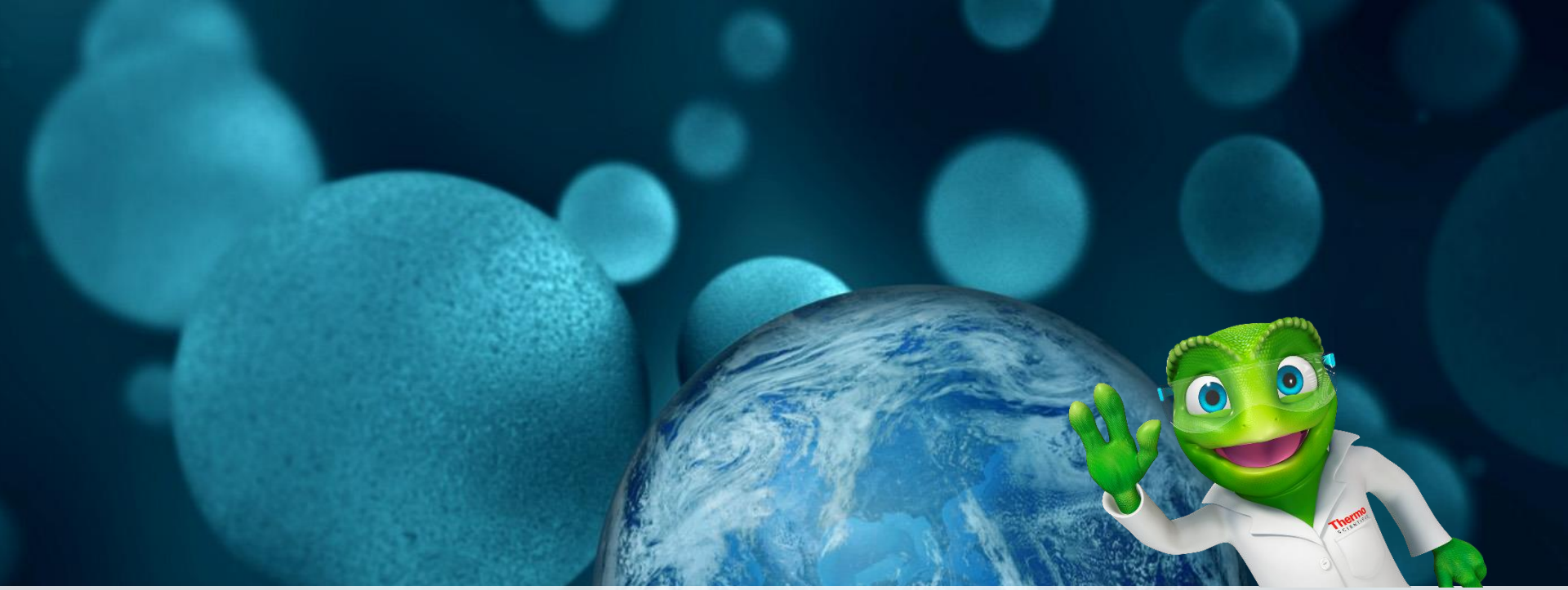


ThermoFisher SCIENTIFIC

Chromeleon DDK Development Training – Day4

Anton Kyosev, CMD Senior Staff Software Engineer
Yubo Dong, CMD Software Engineering Manager

Day 4 Morning Agenda

- Instrument Method
  - Instrument Method Editor and Wizard
  - Instrument Method structure
  - Instrument Method Plug-In structure
  - Creating a plug-in
  - Components: An easy way to create plug-ins.
  - Components: Enabling / Disabling, Constraint Checking
  - Editing an instrument method without components
  - System Components
  - Complex Device Components

Instrument Method Editor

Purpose of the instrument method editor:

- Allow users to create a injection method
- Guide to user through the creation process
- Allow modifying an existing method
- Support different modes (Wizard mode and Editor)
- Validate the input data

Important: Instrument method editor plug-ins work only on the symbol table

Instrument Method Editor – Wizard Mode

Chromeleon Instrument method editor supports two modes

Wizard mode

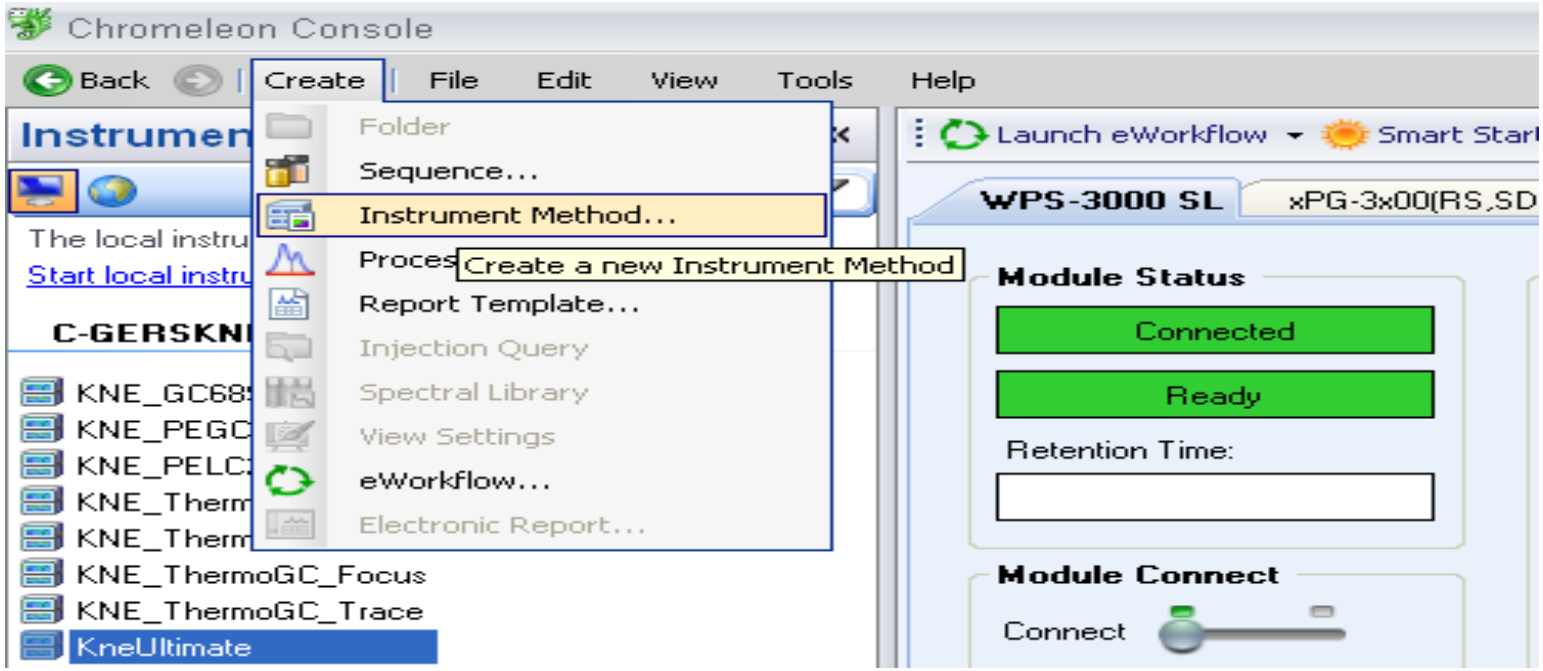
- Dialog based approach
- **Requires instrument connection**
- Does not allow empty input fields
- Does not display advanced options
- Is used to create new methods
- Assigns default values to method steps based on current state

Editor mode

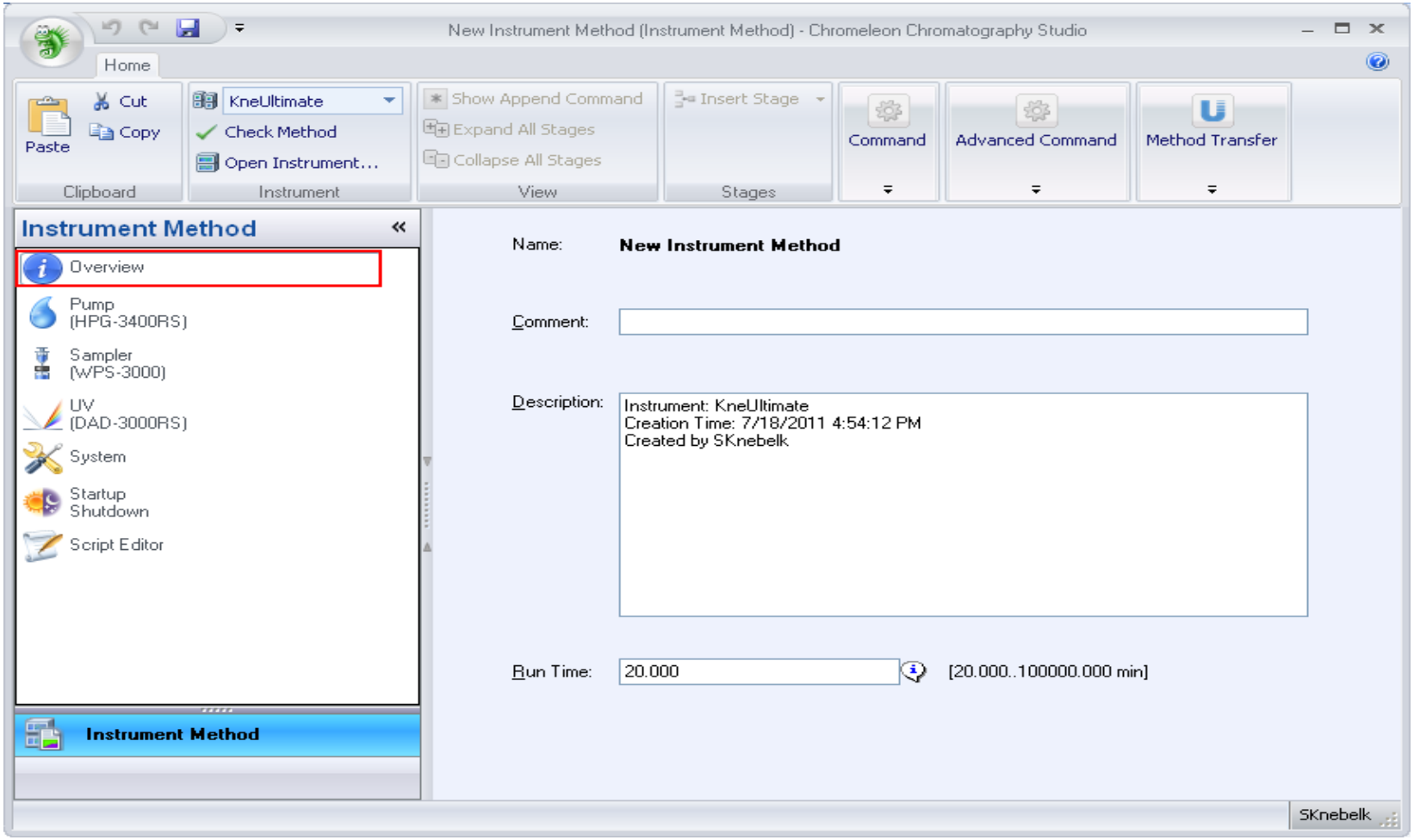
- Allows modification of existing methods
- **Does not require an instrument connection**
- Accepts empty input fields (they represent a missing method step)
- Must be prepared for modifications made by the user in the script view
- Displays UI for advanced settings
- Is embedded in the Chromeleon Studio

Prerequisites for starting the IME Wizard:

- **Check that the instrument controller is running with an instrument containing the desired device**
- Check whether a plug-in with a corresponding driver ID for the selected device is available
- Open the Chromeleon Console and select an device.
- Select “Create” from the menu bar and select “Instrument method”.



Instrument Method Editor – Overview Page

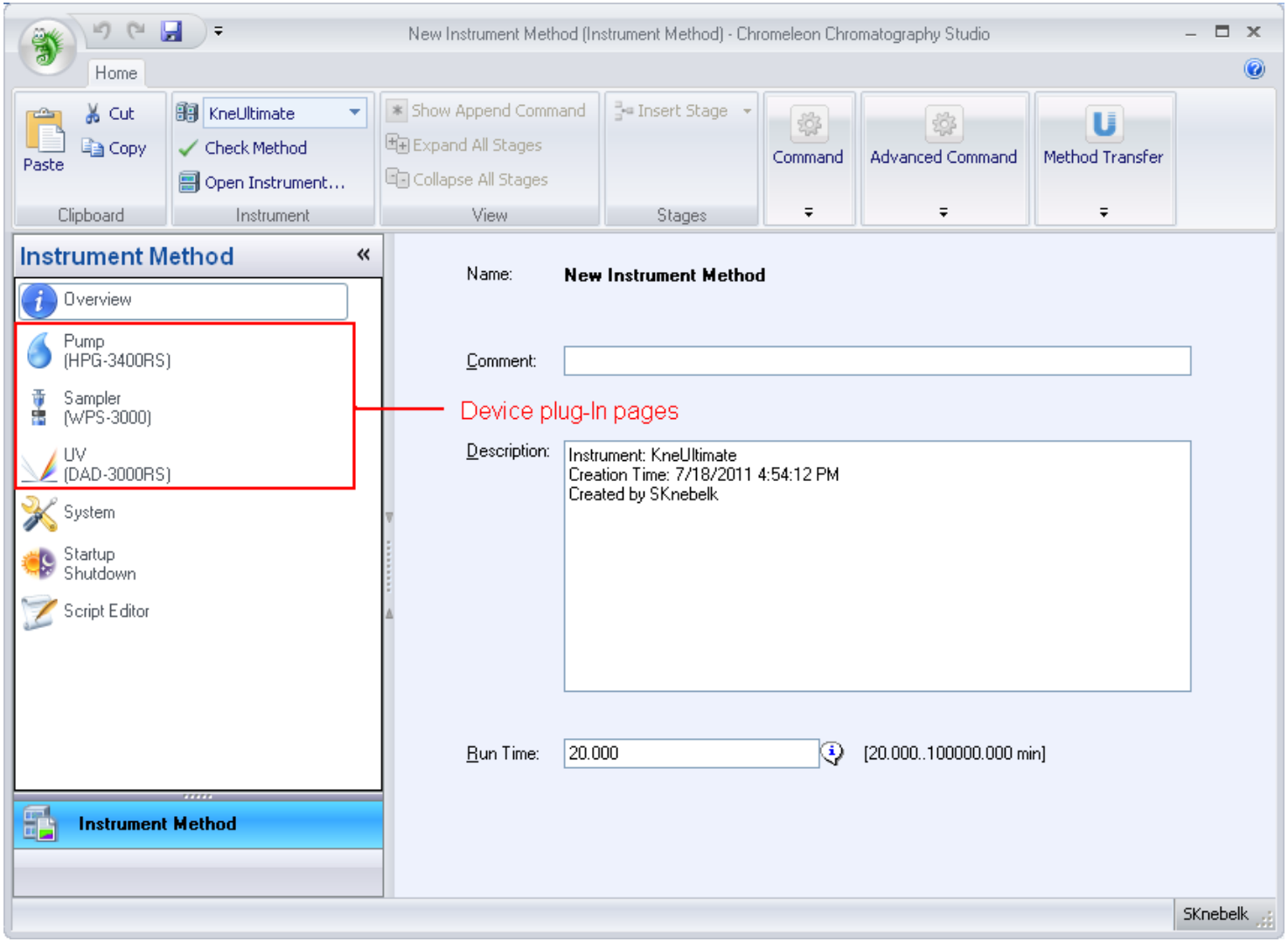


Modify

- Comment
- Description
- Run Time

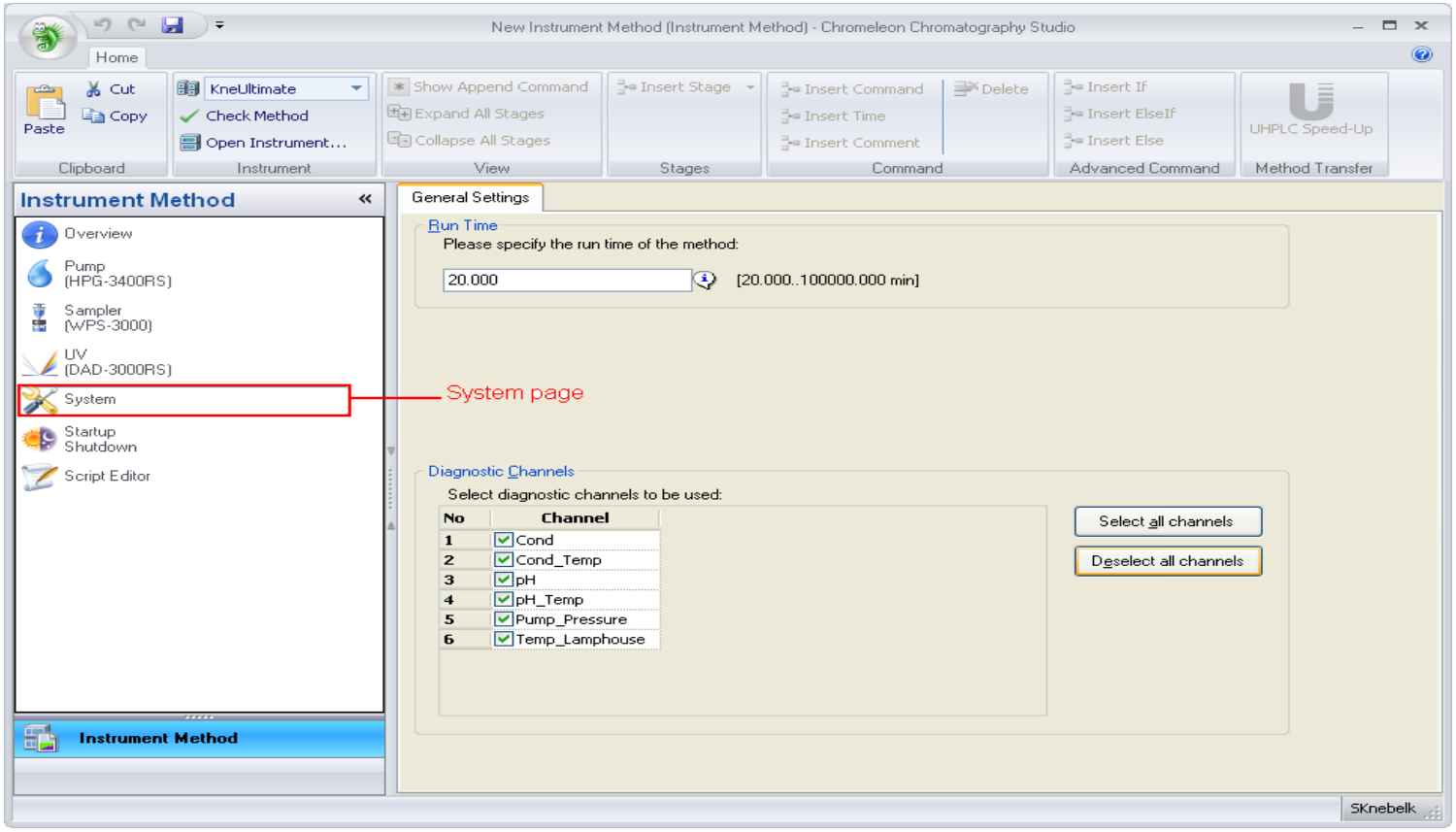
ThermoFisher SCIENTIFIC

Instrument Method Editor – Device Plug-In Pages



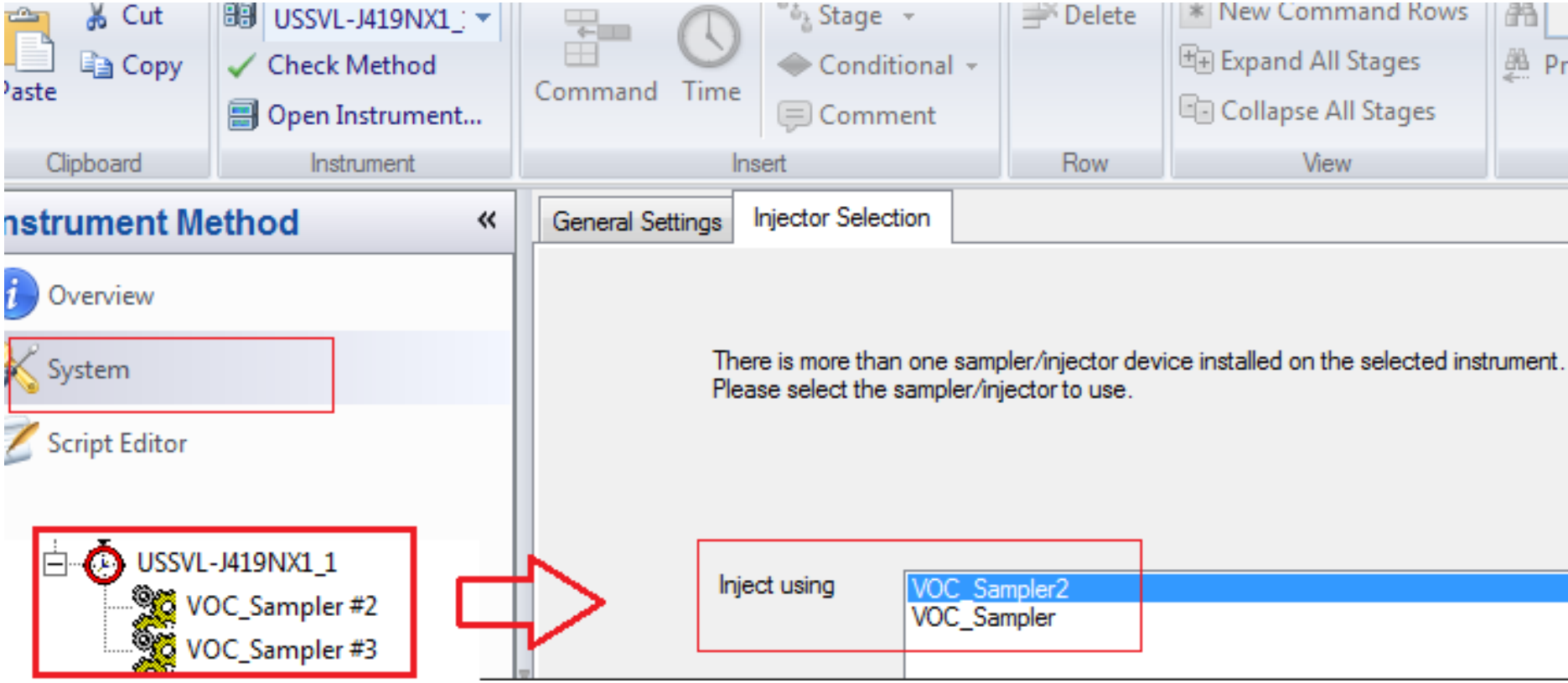
- Modify device specific settings

Instrument Method Editor – System Pages



- Modify run time of the method
- Select/Deselect diagnostic channels for acquisition
- If there is >1 injector on the instrument an extra injector selection tab page is displayed

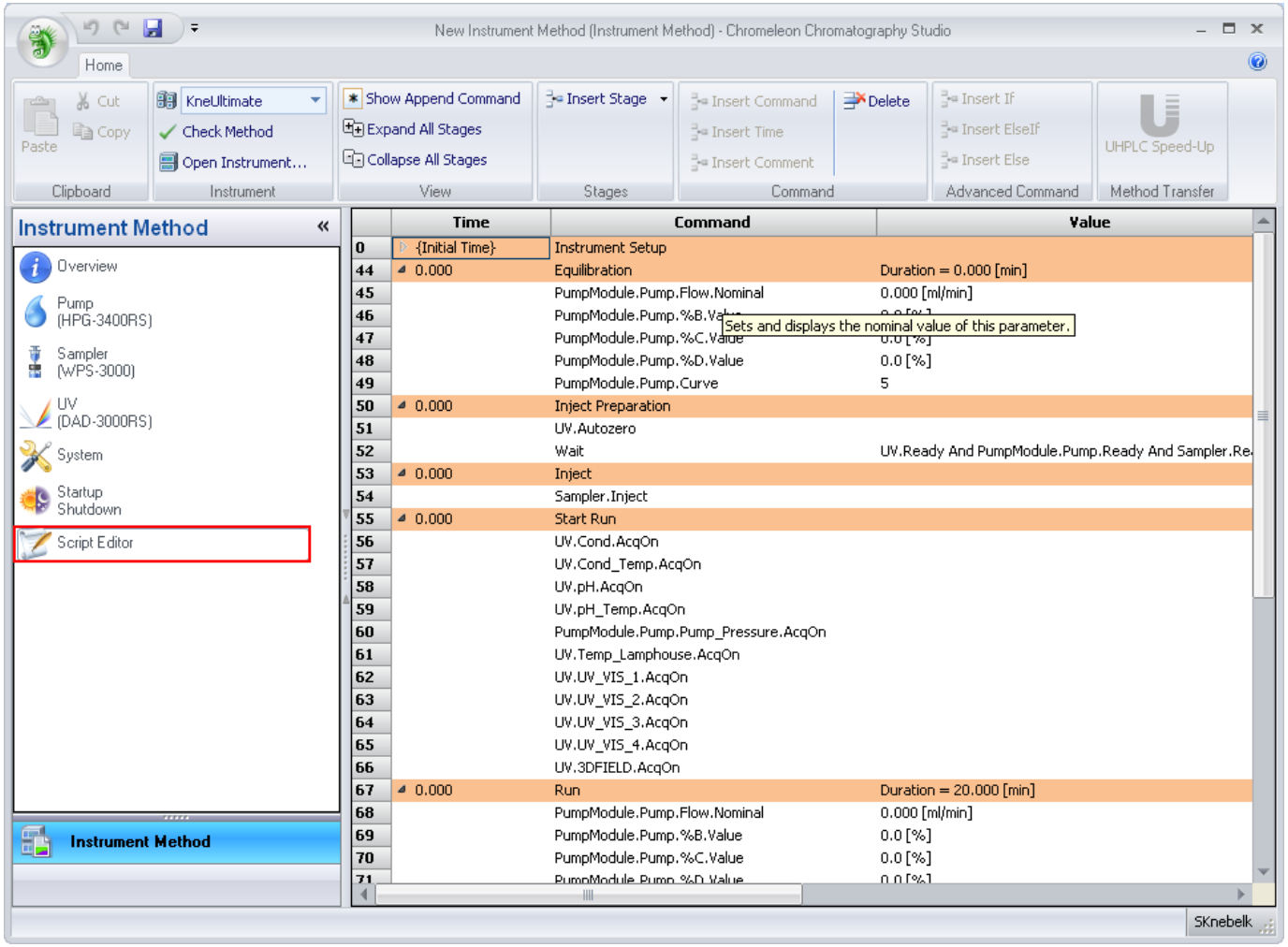
Two Injector Example



ThermoFisher SCIENTIFIC

Instrument Method Editor – Script Editor

- Modify method manually
- Allows the user to create more complex methods



ThermoFisher SCIENTIFIC

Instrument Method Editor

Using the instrument method editor results in a instrument method consisting of:

- the Chromeleon symbol table
- the method (stages, time stamps, commands, property assignments…)

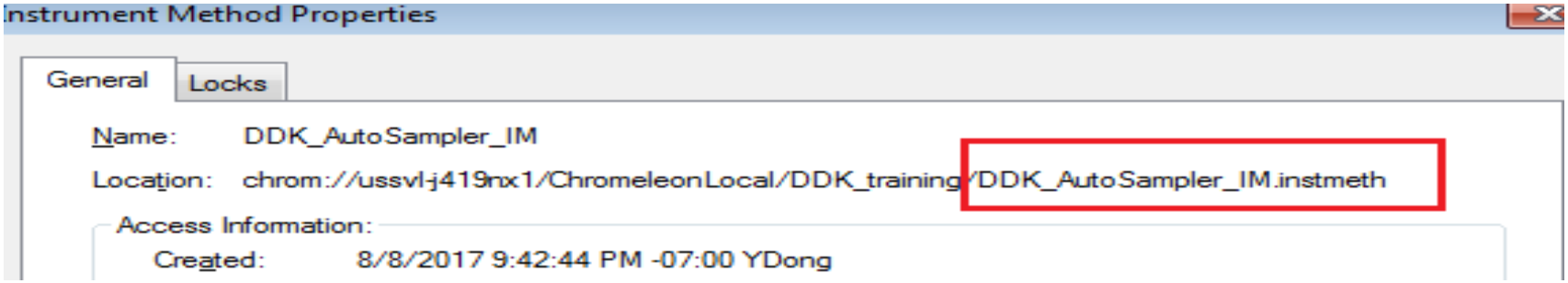
Instrument Method Structure

A CM7 instrument method is a structured document that contains

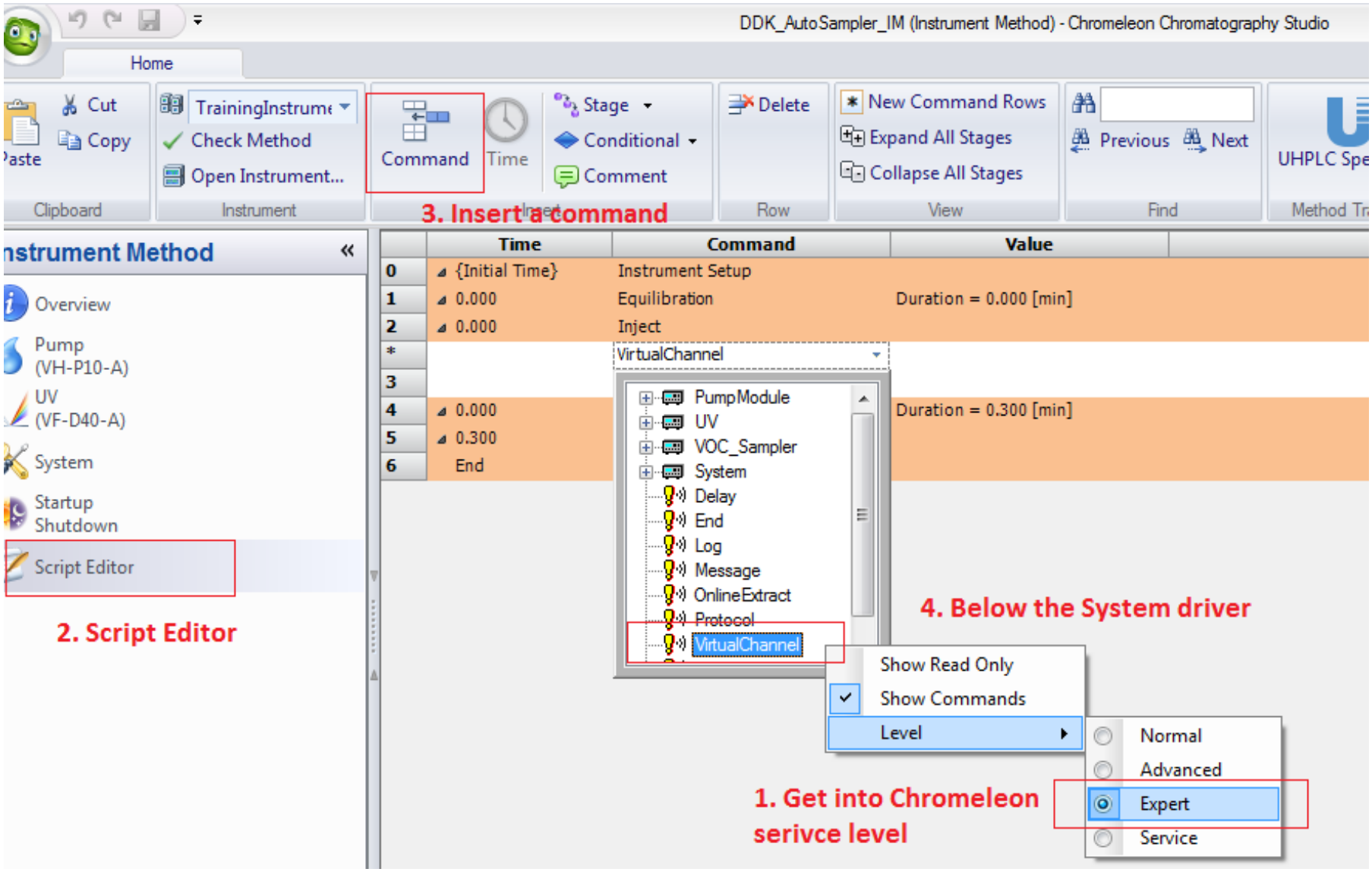
- Stages
- Time steps
- Device property assignments
- Device commands **(including Virtual Channel)**
- Conditional structures (if, elseif, else, triggers)

The CM7 extension of an instrument method is '**.instmeth**'.

The underlying **format is XML**.



Instrument Method Structure



Instrument Method Structure

	Time	Command	Value	Comment
38		PumpModule.Pump.Curve	5	
39	0.000	Inject Preparation		
40		UV.Autozero		
41		Wait	UV.Ready And PumpModule.Pump.Ready	
42	0.000	Inject		
43		Sampler.Inject		
44	0.000	Start Run		
45		UV.Cond.AcqOn		
46		UV.Cond_Temp.AcqOn		
47		UV.pH.AcqOn		
48		UV.pH_Temp.AcqOn		
49		PumpModule.Pump.Pump_Pressure.AcqOn		
50		UV.Temp_Lamphouse.AcqOn		
51		UV.UV_VIS_1.AcqOn		
52		UV.UV_VIS_2.AcqOn		
53		UV.UV_VIS_3.AcqOn		
54		UV.UV_VIS_4.AcqOn		
55		UV.3DFIELD.AcqOn		
56	0.000	Run	Duration = 20.000 [min]	
57		PumpModule.Pump.Flow.Nominal	0.000 [ml/min]	
58		PumpModule.Pump.%B.Value	0.0 [%]	
59		PumpModule.Pump.%C.Value	0.0 [%]	
60		PumpModule.Pump.%D.Value	0.0 [%]	
61		PumpModule.Pump.Curve	5	
62	20.000			
63		PumpModule.Pump.Flow.Nominal	0.000 [ml/min]	
64		PumpModule.Pump.%B.Value	0.0 [%]	
65		PumpModule.Pump.%C.Value	0.0 [%]	
66		PumpModule.Pump.%D.Value	0.0 [%]	
67		PumpModule.Pump.Curve	5	
68	20.000	Stop Run		
69		UV.Cond.AcqOff		

Stage

Time Step

Command / Property Step

Instrument Method Structure

- Stages – Defines sections within an instrument method.
  Creates an order for steps with the same execution time.
- Time Steps – Defines time steps within a stage.
- Command Steps – Defines a device command
- Property (assignment) Steps – Assigns a new value to a device property
- Conditional structures – If / ElseIf / Else / Trigger
- Unsupported conditional structures – While

Instrument Method – Stages

The instrument method is grouped into several **stages**. For separation methods (these are used for Injections) the following stages exist:

- **Instrument Setup** – Initial time commands
- **Equilibration** – Commands with negative times
  (Define start parameters for gradient and ramps.)
- **Inject Preparation** – Prepare system for Inject command (Wait xx.Ready)
- **Inject** – Injects command (or sequence)
- **Start Run** – Start Data Acquisition
- **Run** – Time tabled commands
  (gradient and ramps steps, wavelength switching, valve position changes)
- **Stop Run** – Stop Data Acquisition
- **Post Run** – Commands executed after data acquisition
  (e.g. prepare system for next injection)

Injection Methods

	Time	Command	Value
0	{Initial Time}	Instrument Setup	
10	-4,000	Equilibration	Duration = 4,000 [min]
11		PumpModule.Pump.Flow.Nominal	0,000 [ml/min]
12		PumpModule.Pump.%B.Value	0,0 [%]
13		PumpModule.Pump.%C.Value	0,0 [%]
14		PumpModule.Pump.%D.Value	0,0 [%]
15		PumpModule.Pump.Curve	5
16	0,000	Inject Preparation	
17		Wait	FLD.Ready And PumpModule.Pump.Ready
18	0,000	Start Run	
19		FLD.FLD_FlowCell.AcqOn	
20		PumpModule.Pump.Pump_Pressure.AcqOn	
21	0,000	Run	Duration = 20,000 [min]
22		PumpModule.Pump.Flow.Nominal	0,000 [ml/min]
23		PumpModule.Pump.%B.Value	0,0 [%]
24		PumpModule.Pump.%C.Value	0,0 [%]
25		PumpModule.Pump.%D.Value	0,0 [%]
26		PumpModule.Pump.Curve	5
27	20,000		
28		PumpModule.Pump.Flow.Nominal	0,000 [ml/min]
29		PumpModule.Pump.%B.Value	0,0 [%]
30		PumpModule.Pump.%C.Value	0,0 [%]
31		PumpModule.Pump.%D.Value	0,0 [%]
32		PumpModule.Pump.Curve	5
33	20,000	Stop Run	
34		FLD.FLD_FlowCell.AcqOff	
35		PumpModule.Pump.Pump_Pressure.AcqOff	
36	End		

The **Instrument Setup** stage has the time **Initial Time** which is represented as negative infinity. The Chromeleon runtime engine will ignore the time span between the initial stage and the following stage, so the clock will start at the time defined by the first time mark of that stage. The stages **Inject Preparation**, **Start Run** and **Run** all have the same time of 0. The duration of a stage is determined by the time step with the greatest time. For the above shown example the **AcqOff** commands will be executed at **~ 24 min** after the start of the injection run (**4 min** for equilibration **+** time needed for injection **+ 20 min** run time after inject).

Injection Methods

The following time steps are pre-defined by the instrument method editor:

```
MethodTime.Initial   double.NegativeInfinity;

MethodTime.Zero      0

MethodTime.Minimum   int.MinValue + 1

MethodTime.Maximum   int.MaxValue
```

Events during Script Execution

Consider the following script:

```
{InitialTime} Sampler.Position = H12
              Sampler.Volume   = 10
0.000         UV.Autozero
              Wait UV.Ready
              Sampler.Inject
              UV_VIS_1.AcqOn
1.000         UV_VIS_1.AcqOff
End
```

It is a stripped down version of a typical instrument method used for an Injection. The following is the list of events that are dispatched to the drivers when this script executes:

Events during Script Execution

```
{InitialTime} Sampler.Position = H12
              Sampler.Volume   = 10
0.000         UV.Autozero
              Wait UV.Ready
              Sampler.Inject
              UV_VIS_1.AcqOn
1.000         UV_VIS_1.AcqOff
End
```

OnBroadcast(SampleStart)	this is an injection run
OnBroadcast(ClockStart)	script clock has started
OnLatch	{InitialTime} commands follow
OnSetProperty(Position, H12)	only to driver owning "Sampler" device
OnSetProperty(Volume, 10)	only to driver owning "Sampler" device
OnSync	{InitialTime} commands complete

Events during Script Execution

```
{InitialTime} Sampler.Position = H12
              Sampler.Volume   = 10
0.000         UV.Autozero
              Wait UV.Ready
              Sampler.Inject
              UV_VIS_1.AcqOn
1.000         UV_VIS_1.AcqOff
End
```

OnLatch	"0.000" commands follow
OnCommand(Autozero)	only to driver owning "UV" device
OnSync	no more "0.000" commands for now
OnBroadcast(Hold)	instrument is now waiting for UV.Ready

Events during Script Execution

```
{InitialTime} Sampler.Position = H12
              Sampler.Volume   = 10
0.000         UV.Autozero
              Wait UV.Ready
              Sampler.Inject
              UV_VIS_1.AcqOn
1.000         UV_VIS_1.AcqOff
End
```

<time passes while detector performs autozero operation>

OnBroadcast(Continue)	Waiting ended, execution resumes
OnLatch	more "0.000" commands follow
OnCommand(Inject)	only to driver owning "Sampler" device
OnSync	no more "0.000" commands for now
OnBroadcast(Hold)	instrument is now waiting for inject response
OnBroadcast(Inject)	an injection is in progress

Events during Script Execution

```
{InitialTime} Sampler.Position = H12
              Sampler.Volume   = 10
0.000         UV.Autozero
              Wait UV.Ready
              Sampler.Inject
              UV_VIS_1.AcqOn
1.000         UV_VIS_1.AcqOff
End
```

<time passes while sampler performs injection>

OnBroadcast(InjectResponse)	inject response
OnBroadcast(Continue)	Waiting ended, execution resumes
OnCommand(AcqOn)	only to driver owning "UV_VIS_1" device
OnSync	"0.000" commands complete
OnBroadcast(ZeroCrossed)	script clock is now at a time > 0

Events during Script Execution

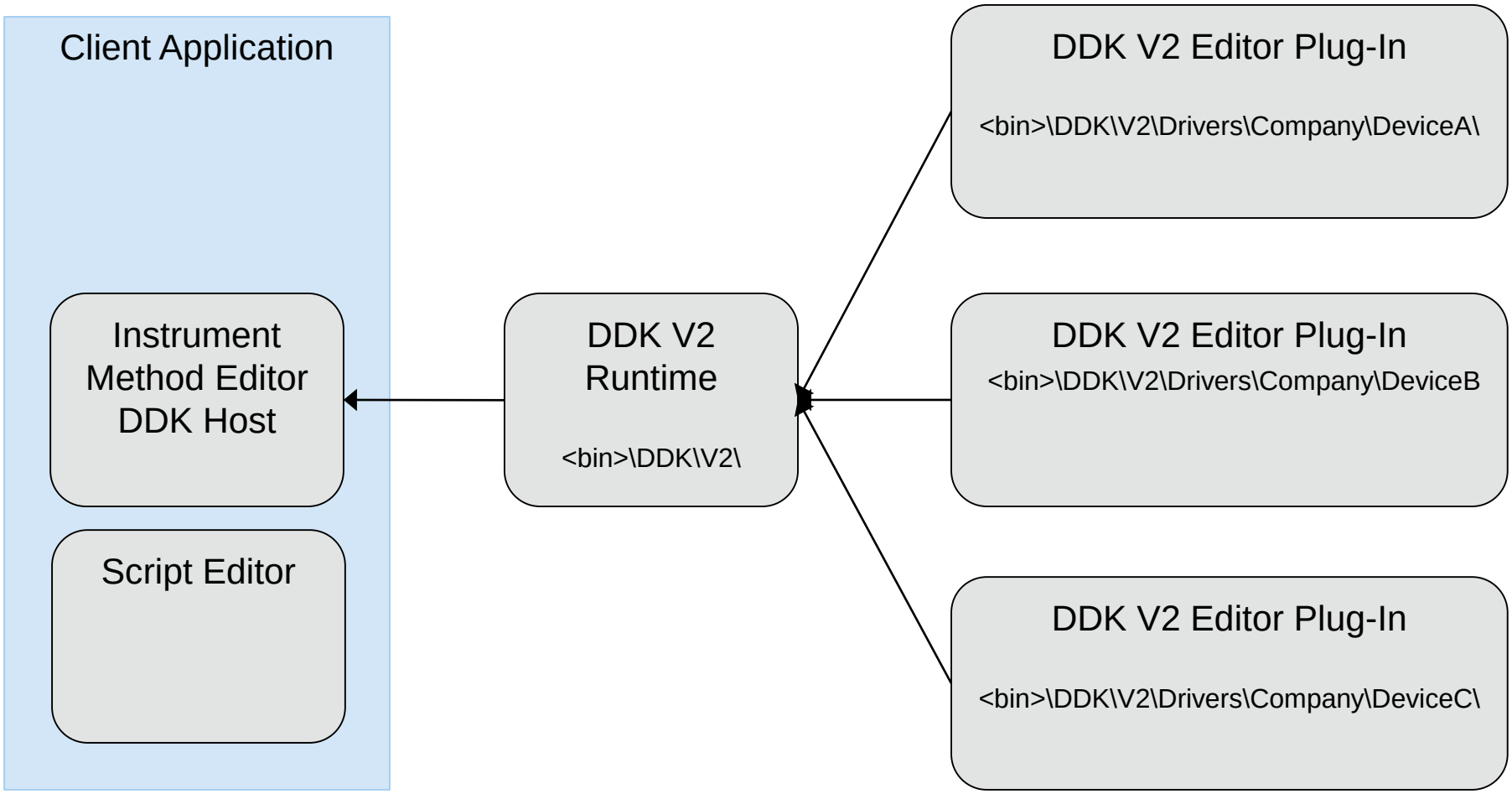
```
{InitialTime} Sampler.Position = H12
              Sampler.Volume   = 10
0.000         UV.Autozero
              Wait UV.Ready
              Sampler.Inject
              UV_VIS_1.AcqOn
1.000         UV_VIS_1.AcqOff
End
```

<1 min pass>

OnLatch	"1.000" commands follow
OnCommand(AcqOff)	only to driver owning "UV_VIS_1" device
OnSync	no more "1.000" commands for now

OnBroadcast(ClockStop)	script is complete, script clock stopped
OnBroadcast(SampleEnd)	injection run is complete

Instrument Method Editor Architecture



DDK V2 Runtime Assemblies

- Dionex.Chromeleon.DDK.V2.Symbols
  Chromeleon symbols and types

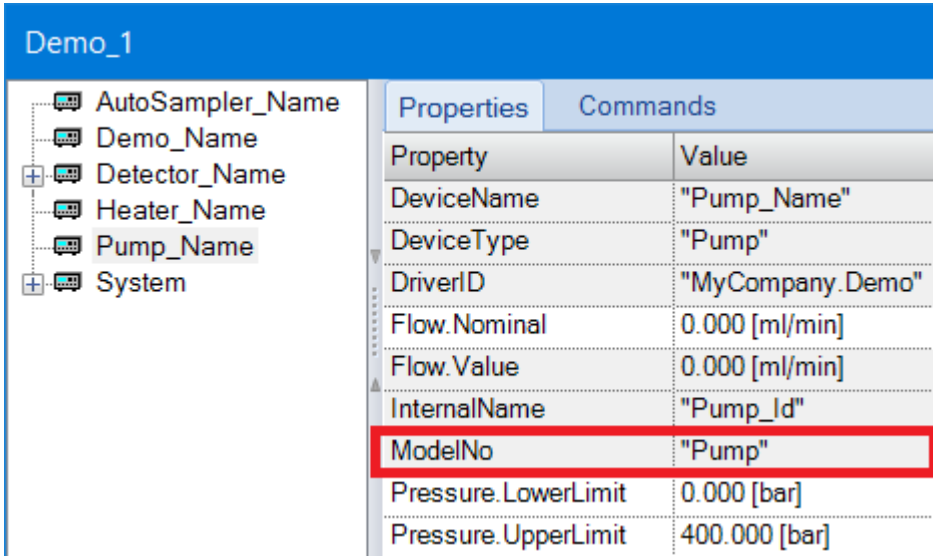
- Dionex.Chromeleon.DDK.V2.InstrumentMethodEditor
  Plug-in structure, component model, method steps

- Dionex.Chromeleon.DDK.V2.InstrumentMethodEditor.Components
  Reusable simple and complex components

Plug-In Enumeration And Assignment

- Plug-ins refer to a driver via the `DriverID` attribute. The `DriverID` must be set to the value of the driver's **ModuleNo** property
- The editor plug-in assembly must contain an object `class PlugIn` that is derived from
  `Dionex.Chromeleon.DDK.V2.InstrumentMethodEditor.IInitEditorPlugIn`
  and has an attribute of type
  `Dionex.Chromeleon.DDK.V2.Driver.DriverID`
- The editor plug-in assembly must be signed using **DriverSignature.exe**
- If a driver with matching `DriverID` is in the instrument configuration, the plug-in will be loaded.

```
[DriverID("Pump")]
public class PlugIn : IInitEditorPlugIn
```



Important Interfaces

IInitEditorPlugIn

- Allows to register system components
- Allows to access the device specific part of the symbol table

Implement for each plug-in (the `class` must be public)

Add driver ID attribute

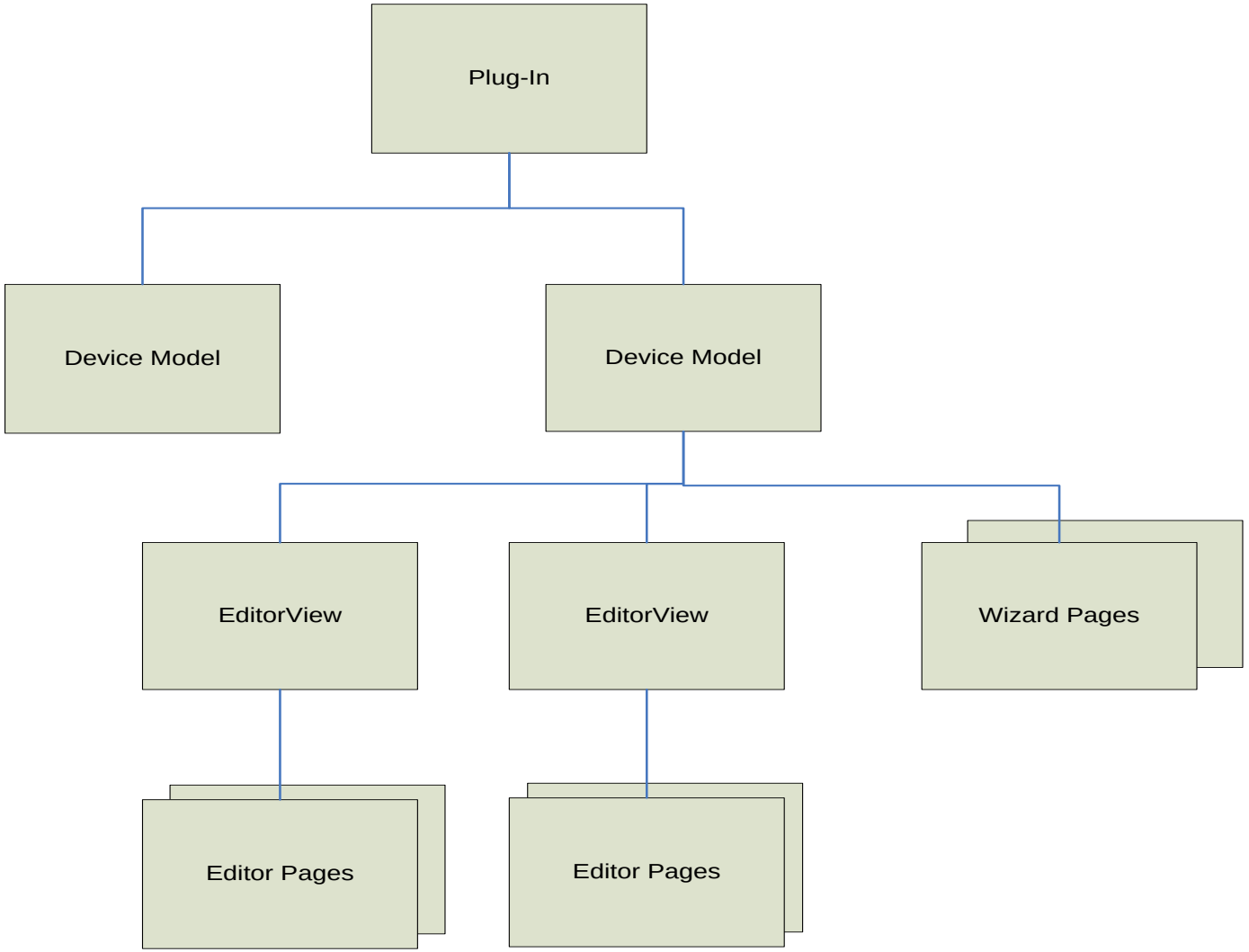
```
internal static class ModuleNo
{
    public const string Pump = "Pump";
    public const string DualPump = "DualPump";
}

[DriverID(ModuleNo.Pump)]
[DriverID(ModuleNo.DualPump)]
public class PlugIn : IInitEditorPlugIn
{
    void IInitEditorPlugIn.Initialize(IEditorPlugIn plugIn)
    {
        // navigate through the symbol table and create device pages
        // register system components
```

Important Interfaces - cont'd

- IInitPage
  - Implement interface once for each device page
  - Allows to register for events (**PageEnterEvent, PageLeaveEvent, PageValidationEvent, WizardFinishedEvent**)
  - Allows to access system components
  - Provides access to the current method object
  - Represents a page in the DDK framework
  - Page can be added to the wizard sequence or to a an editor device view

```
public partial class DevicePage : UserControl, IInitPage
{
    public DevicePage()
    {
        InitializeComponent();
    }

    public void Initialize(IPage page, IEditMethod editMethod)
    {
        // create all controls needed
    }
}
```

Component Structure of a Plug-In



Plug-In Creation

```
[DriverID("ModuleNo")]
public class PlugIn : IInitEditorPlugIn
{
    void IInitEditorPlugIn.Initialize(IEditorPlugIn plugIn)
    {
        // Create a device model
        IDeviceModel deviceModel = plugIn.DeviceModels.Add(plugIn.Symbol, Resources.DeviceIcon);

        // Create a page and register it
        IPage page = deviceModel.CreatePage(new DevicePage(), plugIn.Symbol.Name, plugIn.Symbol);

        // Create an editor device view and add your page
        IEditorDeviceView view = deviceModel.EditorDeviceViews.Add(EditorViewOrder.MiscDetectorViews);
        view.Pages.Add(page);

        // Add your page to the wizard pages
        deviceModel.WizardPages.Add(page, WizardPageOrder.MiscDetectorPages);
    }
}
```

Plug-In Creation

- Create a `UserControl`
- Derive it from `IInitPage`
- Add some components and / or subscribe page editing events

```
public partial class DevicePage : UserControl, IInitPage
{
    public DevicePage()
    {
        InitializeComponent();
    }

    public void Initialize(IPage page, IEditMethod editMethod)
    {
    }
}
```

Introduction of reusable components

- Simple Init Time components
- Enabling / Disabling controller
- Constraint Checking components
- System components
- Complex components
- Method steps handling

Editing Simple Init Time Properties

Drag‘n Drop one of the predefined controls onto your plug-in page:

- PropertyStepTextBox
  a text box that edits an init time property step
- PropertyStepCheckBox
  a check box that edits an init time property step
- PropertyStepComboBox
  a combo box that edits an init time property step
- PropertyRangeLabel
  a label that displays the allowed range of a property

  Specify the property path relative to the page‘s symbol in the property dialog of the control.

Controller Classes

A Controller connects an initial time property step with a Windows.Forms.Control

It handles

- Data binding between the property step value
- Displaying information about the property
- Displaying input errors and constraint errors in a specific error provider icon
- Issuing an error on page validation

Following controllers are available:

- PropertyStepTextController
  Connects the Text property of a Windows.Forms.Control with the value of an initial time property step

- PropertyStepComboBoxController
  Same as PropertyStepTextController but also handles the list box part of a Windows.Forms.ComboBox

- PropertyStepCheckBoxController
  Controls a boolean type property step using the CheckState of the Windows.Forms.CheckBox.

- PropertyRangeTextContontroller
  Displays the symbol property range using the text property of a Windows.Forms.Control

Enabling / Disabling Components

Use the `EnableController` to enable / disable Controls AND instrument method step generation

The following example creates an Enable Controller that enables / disables temperature settings depending on the Temperature Control Check Box state:

```
public void Initialize(IPage page, IEditMethod editMethod)
{
    m_EnableController = new EnableController(page.Component,
                                              m_TempControlCheckBox.Controller);

    m_EnableController.ControlledItems.AddRange(new Control[]
                                                {
                                                    m_TextBoxLowerLimit,
                                                    m_TextBoxUpperLimit,
                                                    m_TextBoxNominal
                                                });
}
```

Component Constraint Checking

DDK offers some classes that facilitate constraint checking between property step controllers.

Those classes handle

- Checking the constraint every time one of the value changes
- Consider enable state of the controller objects
- Setting a constraint error message which is displayed in an error provider icon
- Issuing a constraint error message when the page is being validated

There are

- Simple multi purpose constraint checking classes

  `BinaryConstraintController`
- Constraint checking components for specific tasks

  `MinMaxNominalController`

Constraint Checking: **BinaryConstraintController**

The `BinaryConstraintController` check constraints between two values.

Example: The following code checks constraints between two values given by two property step controller objects:

```
PropertyStepTextController m_3DFieldMaxWavelengthController
PropertyStepTextController m_3DFieldMinWavelengthController
```

Used constraints are:

- a predefined constraint `BinaryConstraintController.GreaterThanConstraint`
- and a user defined constraint `ExceedsMinWavelengthRange`

```
public static bool ExceedsMinWavelengthRange(double first, double second)
{
    return first >= second + MinWavelengthDiff;
}

public void Initialize(IPage page, IEditMethod editMethod)
{
    var wavelengthLimitsChecker = new BinaryConstraintController(m_Page.Component,
                                                                 m_Page.Component.Symbol,
                                                        m_3DFieldMaxWavelengthController,
                                                        m_3DFieldMinWavelengthController);
    wavelengthLimitsChecker.Constraints.Add(
        new BinaryConstraint(BinaryConstraintController.GreaterThanConstraint,
                                                                    Resources.InvalidWavelengthLimits));

    wavelengthLimitsChecker.Constraints.Add(
        new BinaryConstraint(ExceedsMinWavelengthRange, Resources.InvalidWavelengthRange));
}
```

Constraint Checking Components

Constraint Checking Components are predefined components for typical constraints checking tasks.

Available components are

- `MinMaxController`
  Checks, that maximum value > minimum value

- `MinMaxNominalController`
  Checks, that maximum value > minimum value and minimum value <= nominal value <= maximum value.

System Components

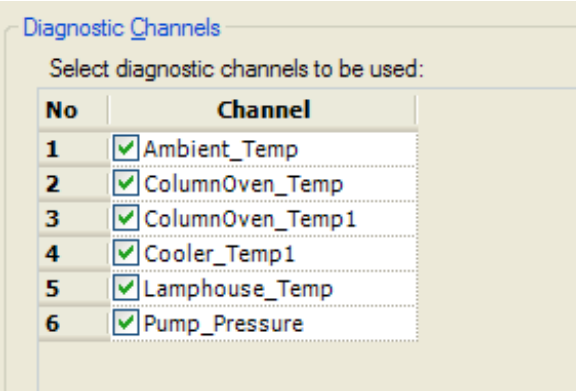
- System components control functions that are common to more than one plug-in.
- Provide means to communicate between plug-ins.
- Examples for system components:
  Control diagnostic channel selection and overall run time,
  Selecting an injector, selecting a GC-side, handling the Wait.Ready command …
- System components can be accessed via

```
Dionex.Chromeleon.DDK.V2.InstrumentMethodEditor.Components
```

System Components: Data Acquisition

- Register all **diagnostic** channels with the system component `IChromatographySystem.DataAcquisition.Channels`

- Registered channels will be displayed in the **Diagnostic Channels** (with `m_Channel.NeedsIntegration = false`) grid. Acquisition Commands will be created automatically if the channel is selected.



System Components: Data Acquisition

```
[DriverID("Detector")]
public class PlugIn : IInitEditorPlugIn
{
    private IDetector m_Detector;

    void IInitEditorPlugIn.Initialize(IEditorPlugIn plugIn)
    {
        m_Detector = plugIn.System.DataAcquisition.Detectors.Add(plugIn.Symbol, true);

        foreach (ISymbol symbol in plugIn.Symbol.ChildrenOfType(SymbolType.Channel))
        {
            if (symbol.AuditLevel <= AuditLevel.Advanced)
            {
                continue;
            }

            if (IsDiagnosticChannel(symbol))
            {
                m_Detector.Channels.Add(symbol);
            }
        }

        IDeviceModel deviceModel = plugIn.DeviceModels.Add(plugIn.Symbol, Resources.Detector);
        IPage page = deviceModel.CreatePage(new DetectorPage(m_Detector), plugIn.Symbol.Name,
                                                                                plugIn.Symbol);
        IEditorDeviceView view = deviceModel.EditorDeviceViews.Add(EditorViewOrder.CDDetectorViews);
        view.Pages.Add(page);
        deviceModel.WizardPages.Add(page, WizardPageOrder.CDDetectorPages);
    }
}
```

System Components: IReadyForInjectDeviceCollection

- Register devices that need to prepare for Injection.
  Use
  `IChromatographySystem.IInjectorSubsystem.WaitForInjectDevices`

- The system will automatically add an xxx.Ready clause for this device to the wait command just before Inject:

39		Pump.%C.Value	0,0 [%]
40		Pump.%D.Value	0,0 [%]
41		Pump.Curve	5
42	0,000	Inject Preparation	
43		Wait	ColumnOven.Ready And Sampler.Ready
44		UV.Autozero	
45		Wait	UV.Ready
46	0,000	Inject	
47		Sampler.Inject	
48	0,000	Start Run	

Complex Device Components

Handles UI and step creation for typical Chromatography devices like pumps, detectors, samplers etc.

They can be

- A complete device page
  (i.e. pump gradient page)
- Parts of a device page
  (pump table component)
- A component w/o UI
  (i.e. special constraint check components)

Complex Device Components

Namespace for components

`Dionex.Chromeleon.DDK.V2.InstrumentMethodEditor.Components`

Assembly

`Dionex.DDK.V2.InstrumentMethodEditor.Components`

Device Specific Plug-Ins: Pumps

Pump plug-ins should first register with the DDK framework.

```
void IInitEditorPlugIn.Initialize(IEditorPlugIn plugIn)
{
    IDeviceModel pumpModel = plugIn.DeviceModels.Add(plugIn.Symbol, DeviceIcon.Pump);
    IDevice deviceSymbol = plugIn.Symbol as IDevice;

    IPumpDescription pumpDescription = plugIn.System.PumpSubsystem.CreatePumpDescription(deviceSymbol);
    plugIn.System.PumpSubsystem.Pumps.Add(pumpDescription);
}
```

Once registered, the system automatically performs the following checks:

- The sum of all eluents must not exceed 100
- There should be no gradient during inject
- There should be no curve before inject.

Device Specific Plug-Ins: Pumps

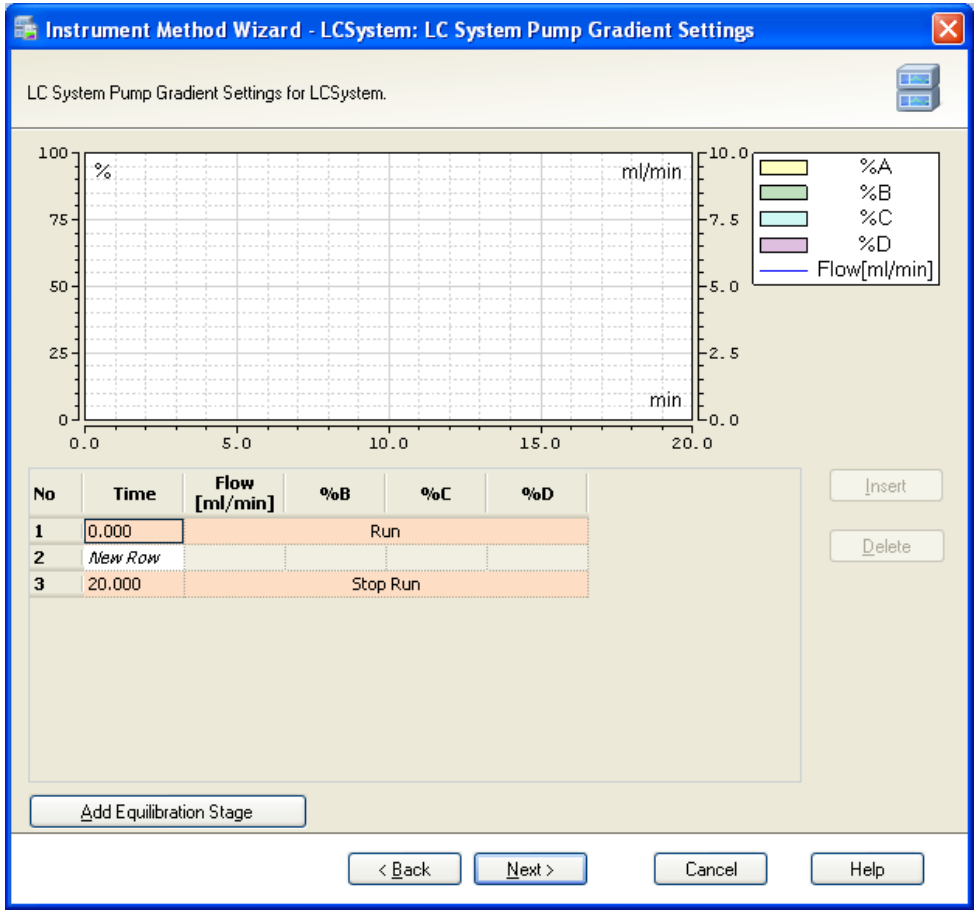
The pump description object `IPumpDescription` can be used to create a pump specific component:

A grid component for editing pump gradients plus a plot control for displaying the defined gradient `UserControl`:
`Dionex.Chromeleon.DDK.V2.InstrumentMethodEditor.Components.PumpGradientComponent`

```
[DriverID("Pump")]
public class PlugIn : IInitEditorPlugIn
{
    private IPumpDescription m_PumpDescription;

    void IInitEditorPlugIn.Initialize(IEditorPlugIn plugIn)
    {
        IDevice pump = plugIn.Symbol as IDevice;
        m_PumpDescription = plugIn.System.PumpSubsystem.CreatePumpDescription(pump);
        //Inform the system that this page handles a pump.
        plugIn.System.PumpSubsystem.Pumps.Add(m_PumpDescription);
    }
}
```

Device Specific Plug-Ins: Pumps

```
public void Initialize(IPage page, IEditMethod editMethod)
{
    PumpGradientComponent comp = new PumpGradientComponent();
    comp.Initialize(m_PumpDescription, page.Component);
    comp.Dock = DockStyle.Fill;
    Controls.Add(comp);
}
```



ThermoFisher SCIENTIFIC

System Components: BasicInjectorComponent

This component does the following

- It registers the sampler with the framework, so it will appear on the sampler selection list if there is more than one sampler in the configuration.
- Writes (or removes) the inject command.
- Adds (or removes) the xxx.Ready clause in the system wait command.
- Component is included in

`Dionex.Chromeleon.DDK.V2.InstrumentMethodEditor.Components.BasicInjectorComponent`

39		Pump.%C.Value	0,0 [%]
40		Pump.%D.Value	0,0 [%]
41		Pump.Curve	5
42	0,000	Inject Preparation	
43		Wait	ColumnOven.Ready And Sampler.Ready
44		UV.Autozero	
45		Wait	UV.Ready
46	0,000	Inject	
47		Sampler.Inject	
48	0,000	Start Run	

Device Specific Plug-Ins: Column Compartments

To control the temperature input and the user set limits for the temperature column compartment plug-ins may use the component

`Dionex.Chromeleon.DDK.V2.InstrumentMethodEditor.Components.MinMaxNominalController`

If the column compartment has additional valves that should be switched during runtime, the plug-in may use the component

`Dionex.Chromeleon.DDK.V2.InstrumentMethodEditor.Components.CommandListControl`

Device Specific Plug-Ins: Detectors

Detector plug-ins should use the

```
Dionex.Chromeleon.DDK.V2.InstrumentMethodEditor.Components.BasicDetectorComponent
```

Typical code:

```
void IInitEditorPlugIn.Initialize(IEditorPlugIn plugIn)
{
    IDeviceModel deviceModel = plugIn.DeviceModels.Add(plugIn.Symbol, DeviceIcon.UvDetector);
    m_DetectorComponent = new BasicDetectorComponent(deviceModel);
```

The component:

- Registers with the system's detector collection in order to be listed on the system page.
- Enables and disables the device component on selection / deselection of the device.
- If the device has a Ready property, registers this device with the injector sub system in order to create a .Ready clause in the system wait command

Device Specific Plug-Ins: Detector Channel Grid

In order to display and edit initial detector channel conditions, most detectors use the component

`Dionex.Chromeleon.DDK.V2.InstrumentMethodEditor.Components.ChannelGridControl`

The grid allows

- switching a channel on/off, which will add / remove AcqOn and AcqOff commands.
- editing parameters for each channel (Typically: wavelength)
- setting individual AcqOn and AcqOff times. However this is rarely the case, so these columns can optionally be hidden.

Device Specific Plug-Ins: Detector Channel Grid (2)

To initialize the grid, the plug-in needs a `IChannelDescription` for each channel which is edited by the grid.

Example code:

```
private IEnumerable<IChannelDescription> RegisterChannels(IEditorPlugIn plugIn,
                                                          IDetector     detector)
{
    List<IChannelDescription> result = new List<IChannelDescription>();

    IEnumerable<IChannel> channels = plugIn.Symbol.Children.OfType<IChannel>().
                                     Where(channel =>
                                             channel.AuditLevel >= AuditLevel.Normal &&
                                             channel.Child("Wavelength") != null);
    foreach (IChannel channel in channels)
    {
        IChannelDescription channelDescription = detector.Channels.Add(channel);
        result.Add(channelDescription);
    }
    return result;
}
```

Device Specific Plug-Ins: Detector Channel Grid (3)

Channel grids often need specific validation code.

For this purpose ChannelGridControl offers events that are raised if any of the grid values has changed.

In order to display a validation error first obtain a symbol value object for the property:

```
ChannelGridControl.GetSymbolValue(string propertyName, int channelIndex)
```

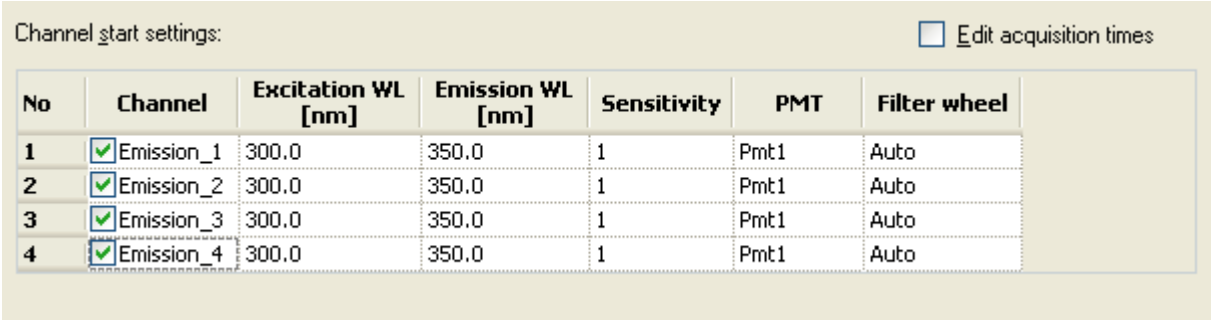
Then use the returned ISymbolValue object to set the constraint error:

```
ISymbolValue.ConstraintError.SetError(string errorMessage,
                                      ConstraintErrorSeverity severity)
```

Don‘t forget to reset the constraint error once the user resolved the conflict.

Device Specific Plug-Ins: Detector Channel Grid (4)

```
private void CreateChannelGrid()
{
  var channelDesc = RegisterChannels(m_PlugIn, myDetectorSymbol);
  var channelGridColumnInfo = CreateChannelGridColumnInfo();
  m_ChannelGrid.Init(page.Component, channelDescriptionList, channelGridColumnInfo);
}

private IList<ChannelGridColumnInfo> CreateChannelGridColumnInfo()
{
  return new List<ChannelGridColumnInfo>
  {
    new ChannelGridColumnInfo("Excitation WL", Resources.Detector_ExWavelength, true),
    new ChannelGridColumnInfo("Sensitivity",   Resources.Detector_Sensitivity , true),
    . . .
  };
}
```



No	Channel	Excitation WL [nm]	Emission WL [nm]	Sensitivity	PMT	Filter wheel
1	Emission_1	300.0	350.0	1	Pmt1	Auto
2	Emission_2	300.0	350.0	1	Pmt1	Auto
3	Emission_3	300.0	350.0	1	Pmt1	Auto
4	Emission_4	300.0	350.0	1	Pmt1	Auto

Method Steps Handling

- A method consists of several stages `IEditMethod.Stages`
- Each stage consists of several time steps `IStage.TimeSteps`
- Each time step consists of several method steps `ITimeStep.SymbolSteps`
- Method steps are property steps or command steps

Method Steps Handling

- How to create a new method step:
  - Solution 1
    - Create a new step:

```
IEditMethod.CreateStep(ISymbol symbol, IEditorComponent component)
```

    - Get/Create the time step for which you want to add the new symbol step:

```
IStage stage = IEditMethod.Stages.GetStage(SeparationMethodStage.Run);
ISymbolStep timestep = stage.TimeSteps.GetOrCreateTimeStep(MethodTime.Zero);
timestep.SymbolSteps.Add(timestep);
```

  - Solution 2

```
IStepLink stepLink = editMethod.CreateStepLink(m_FlField,
                                               page.Component,
                                               SeparationMethodStage.Run);
```

  - Differences:
    - Solution 1 allows you to define a time step
    - Solution 1 can be executed at any time, solution 2 must be executed during `IInitPage.Initialize`

Method Steps Handling

How to find a method step for a symbol

```
public void Initialize(IPage page, IEditMethod editMethod)
{
  foreach (var timeStep in editMethod.Stages.GetAllTimeSteps())
  {
    if (timeStep != null && timeStep.SymbolSteps != null)
    {
      var step = timeStep.SymbolSteps.FirstOrDefault(elem => elem.Symbol == symbol);
      if (step != null)
        break;
    }
  }
```

- How to change the value of a property step
  - Get the ISymbolValue of the property step: IPropertyStep.SymbolValue
  - Call the ValidateValue function:

```
IPropertyStep propertyStep;
string userInput;
ParseResult parseResult = propertyStep.SymbolValue.ValidateValue(userInput);
if (parseResult.Success)
    propertyStep.SymbolValue.BindingValue = userInput;
else
    propertyStep.SymbolValue.ConstraintError.SetError(parseResult.ErrorMessage);
```

Things to remember when writing plug-Ins

- Create a page for each device
- Provide help for each device page
- Register system components like injectors and detectors
- Register diagnostic channels (e.g. Flow, Pressure, Temperature)
- Be aware of method changes in editor mode
- Do not rely on order of method steps
- Methods created via the Wizard must not issue warning or errors in ready check
- Do not change the method without user action in editor mode
- Make sure that plug-In is added to the setup and get distributed
- Make sure that your plug-In assembly/assemblies are signed

- CmDDKV2Examples: ExampleLCSystem.EditorPlug Walk Through

Day 4 Afternoon Agenda

- ePanels

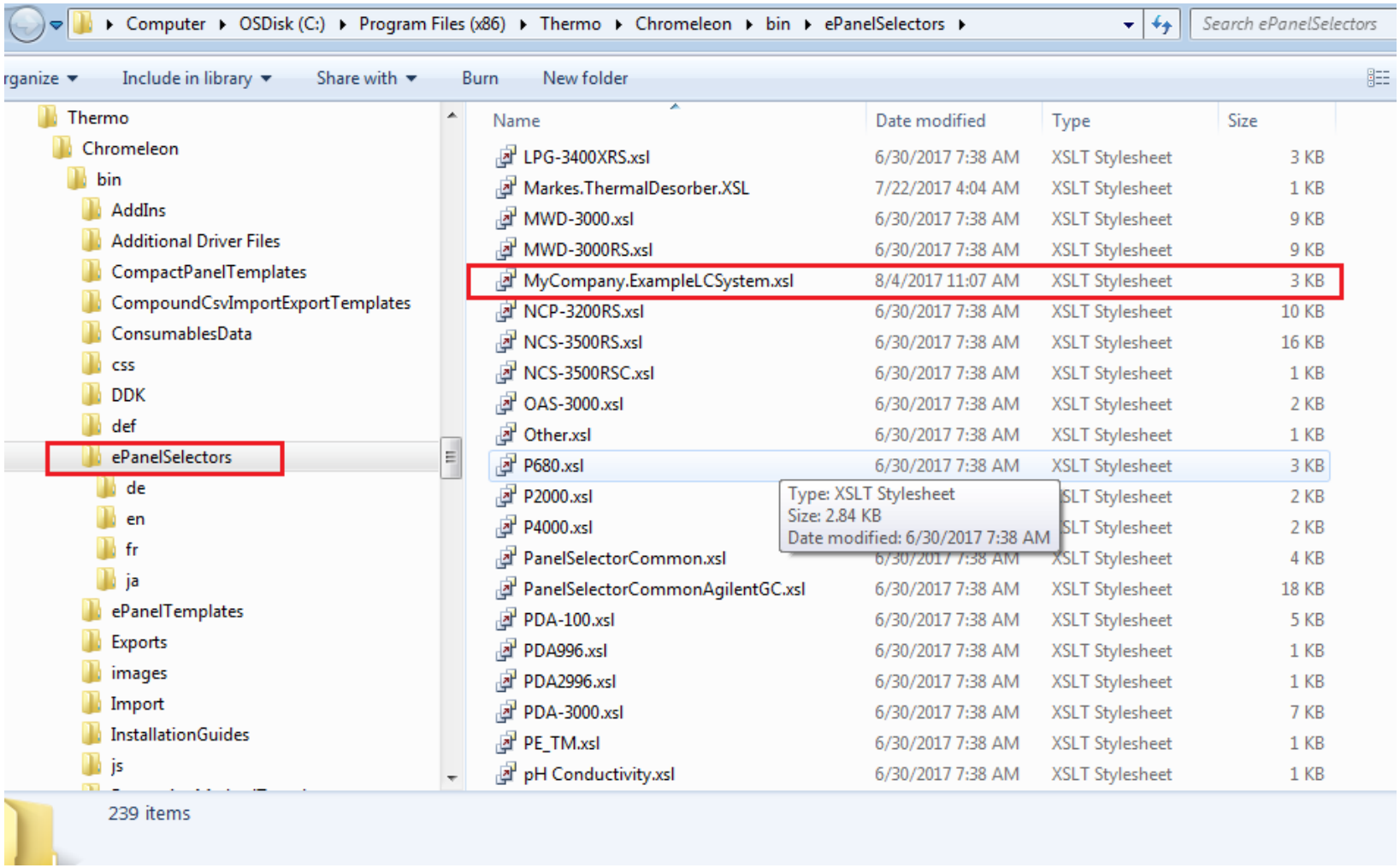
ePanels – Selector File

- **ePanels** are the customizable graphical user interface elements for controlling instruments, and for monitoring instrument parameters and signal data. Chromeleon provides a set of ePanels for each instrument within the Instruments category. The ePanel set is defined by so-called panel selectors (.xsl file extension).

  Chromeleon\bin\ePanelSelectors

ePanel Selector files are XSL files. Every supported module has an associated ePanel selector file named to match either the **DriverId** property value. They use an XML description from the symbol tree of an instrument as input. The XSL file transforms this input XML to a set of XML nodes. The nodes are used by the Chromeleon Client to select the appropriate ePanels and to set the ePanels' device macros.

ePanel' Selector File – Cont'd



ePanel' Selector File – Cont'd

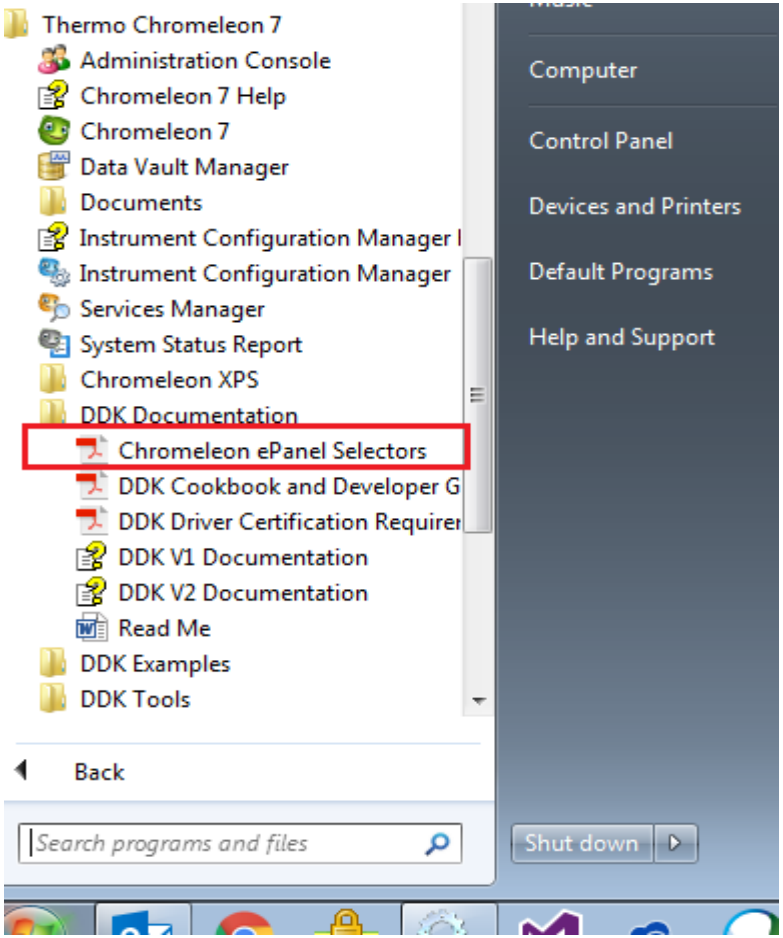
The selector file should use UTF-8 encoding and needs to declare a parameter called "**lang**" for localization purposes

```
<?xml version="1.0" encoding="utf-8"?>
<xsl:stylesheet version="1.0" xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
  <xsl:param name="lang"/>
  <xsl:param name="resourceNode"/>
  <xsl:variable name="Resources" select="$resourceNode"/>

  <xsl:include href="PanelSelectorCommon.xsl"/>

  <xsl:template match="//Symbols/Device[Property[@Name='DeviceType' and
                                                 @Value='Demo']]">
 </xsl:template>
</xsl:stylesheet>
```

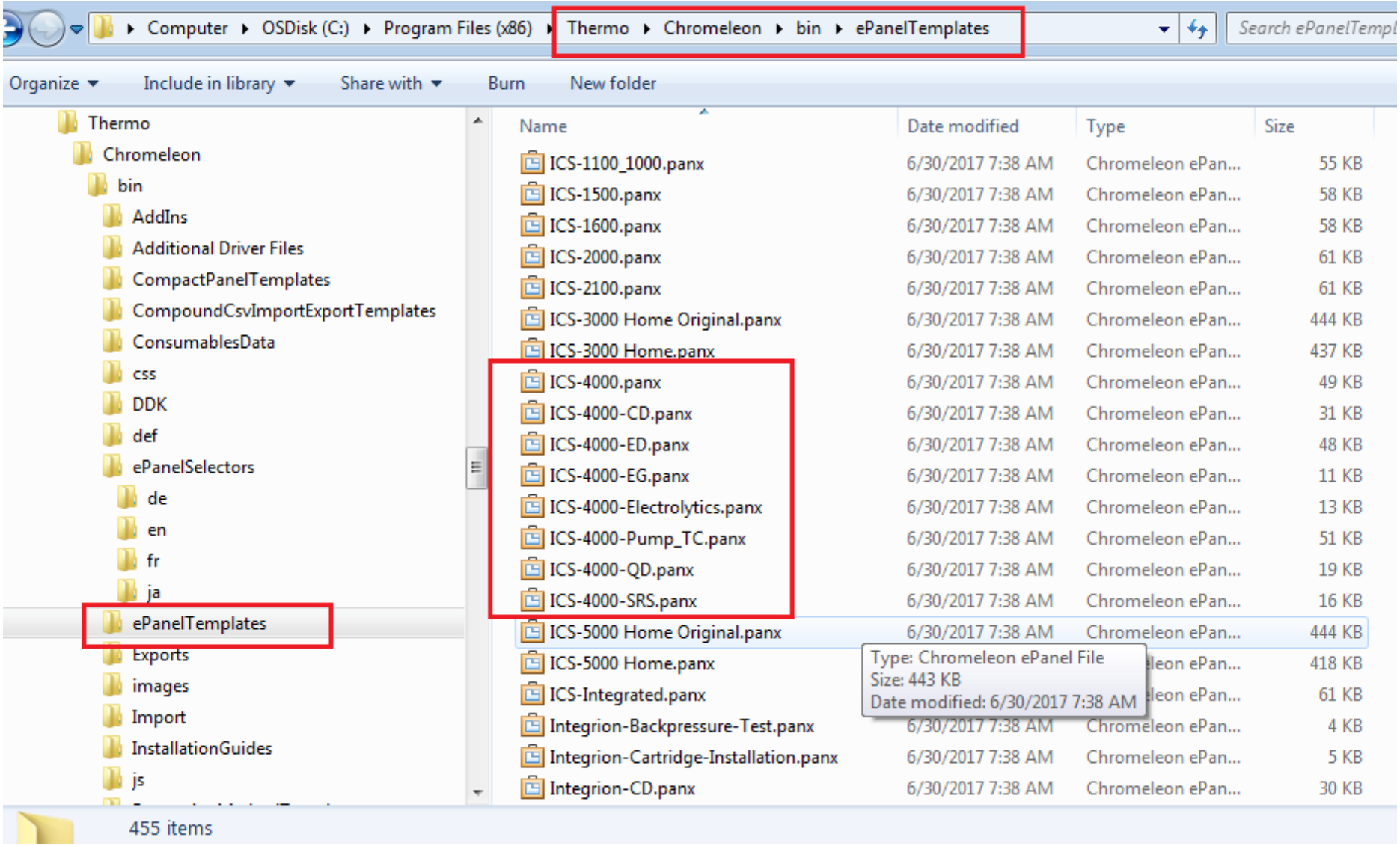
How Chromeleon ePanels work?

- For each module, Chromeleon will run the associated XSL file on the Symbol Table XML. Some of the possible output nodes include:
- **DevicePanels** – defines the Device ePanels that will be displayed in the Console for the auto generated ePanel set.
- **StartupPanels (aka SmartX panels)** – defines the Startup ePanels displayed in the Startup tab.
- **HomePanels** – defines the Pods displayed on the Home tab. A Pod is a child node of the home panels' Panel node:

Check the **4. ePanel selection engine overview** in Chromeleon ePanel Selectors.pdf file for details



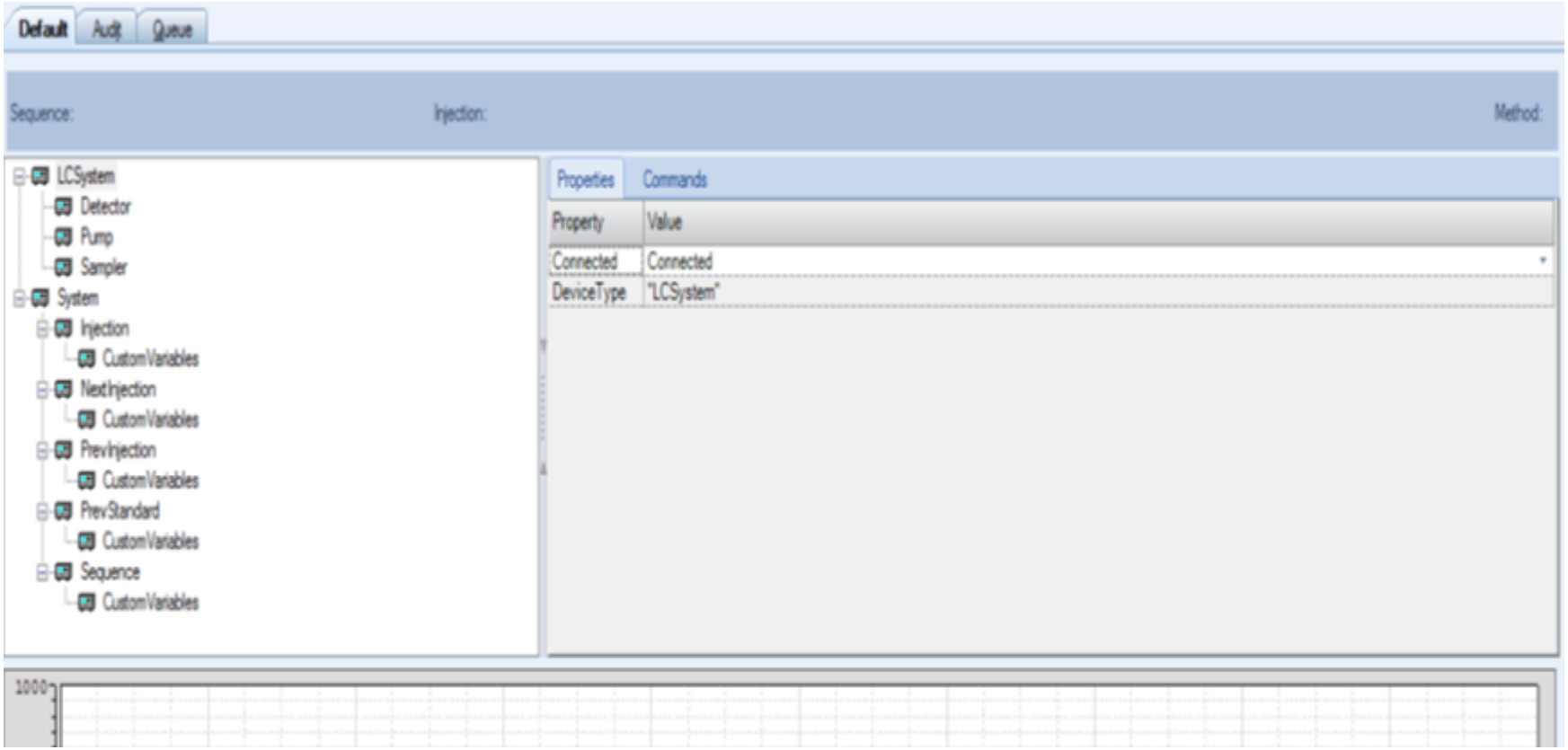
ePanel' PANX Files



ThermoFisher SCIENTIFIC

Default Panels

If one or more devices have no associated selector file then a ‘Default’ panel is added as the rightmost dynamic tab (before the Audit and Queue tabs which are always present)



<Autogenerated> Panels

- <**Panel**> child nodes of the <**DevicePanels**> node describe individual Device ePanels, each of which are presented on one tab of the <Autogenerated> ePanel set. The text on these tabs is determined by the caption of the main panel contained therein. To ensure unique captions in case a particular ePanel is present multiple times in the ePanel set, you can use the predefined macro $0 for the caption, which evaluates to the corresponding device name
- The **Priority** value (a number) controls the ordering and is evaluated such that ePanels with lower numbers appear to the left (i.e. Priority =”10” appears left of Priority = “20”). If two ePanels specify the same priority, tab order depends on the appearance of the corresponding devices in the symbol table (which is not guaranteed).
- Home ePanel and its pods (see reference document)

Macros for Editable Device Names

- **External ePanel macros**
- Chromeleon ePanels support macro replacement (e.g. in captions and property links) to allow ePanels designed such that changing device names in Chromeleon Instrument Configuration Manager will not break ePanel functionality. Users are free to change the default device names provided by a driver; especially, there may be a need to do so when more than one instance of a given driver is added to the same instrument to resolve duplicated device names.
- ePanels can define internal macros (stored within the PANX file), which will resolve to the same value even if that ePanel will be added more than once to an ePanel set.
- ePanels can also reference external macros, names and values for which are also defined in the XML output created by the ePanel selector. Each <**Macro**> node has a **Name** attribute which defines the macro's name, referenced in the ePanel designer as $Name; and a **Value** attribute which defines the replacement value. For the example given above, $Main would resolve to "LCSystem" when the ePanel is initialized.

Macros for Editable Device Names – Cont'd

- External macros can be used where ever ePanels accept a path to a device symbol present in the symbol table, for instance in the following places:
  - Caption
  - Link
  - ShowIf expression
  - EnableIf expression
  - Scripts for Buttons (commands, properties, value expressions and command parameter expressions)
  - Script for Close

ExampleLCSystem ePanel Walk Through