Prevas

Plugin Architectures

Customizable application with plugin architecture

What is a plugin?

From Wikipedia:

In computing, a plug-in (also called plugin, addin, add-in, addon, add-on, snap-in or snapin, but see also extension) consists of a computer program that interacts with a host application (a web browser or an email client, for example) to provide a certain, usually very specific, function "on demand". Add-on is often considered the general term comprising plug-ins, extensions, and themes as subcategories.

- A plugin is a set of methods that are dynamically loaded into a (binary) application.
- The plugins expands the functionallity and/or customizies the user interface.

Prevas

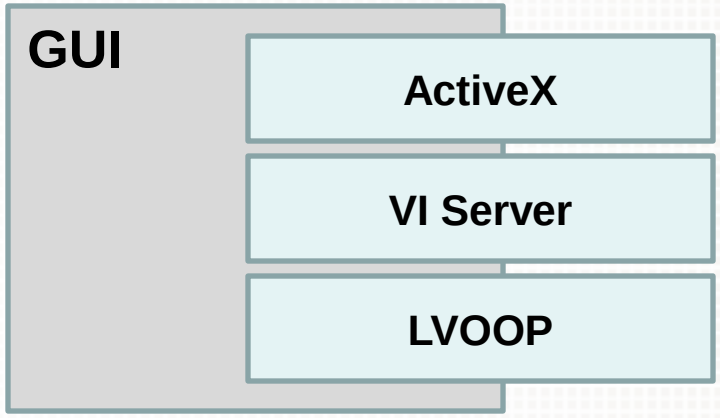
Why use Plugins

- Easier to maintain.
  - Many applications can use the same base.
  - The basics can be more or less static for a long time, since the specific functionallity sits in the plugins.
- Reduces cost
  - The main difference between projects will be the contents of the plugins, and many plugins might be reused between projects.
- Easy to customize
  - Customers can customize the application to meet their specific needs
- Protecting IPs
  - Company IPs are protected in a binary application, while still allowing plugins to benfit from them through a well defined interface.

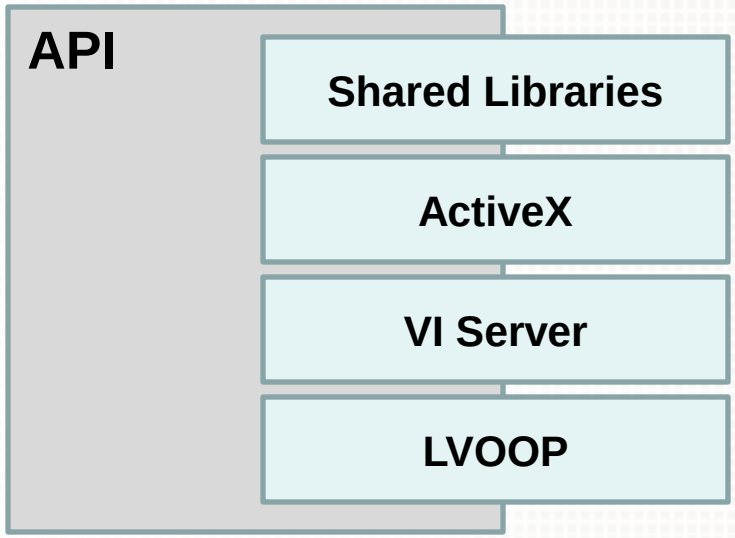
Plugin types for LabVIEW

- Shared Libraries
- ActiveX
- VI Server
- LVOOP

LabVIEW Plugins



”Modifies” or adds new panels to an existing GUI



Adds new functionallity to existing driver layers.

Shared Libraries

Pros

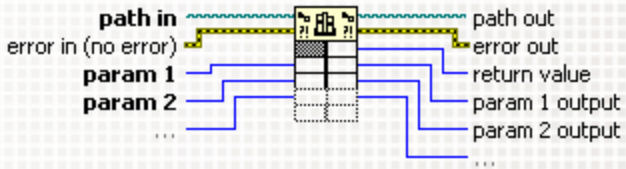
- Support plugins from many other program languages
- Many instrument drivers are delivered as DLL's

Cons

- Dll's written in other languages require other environments to maintain.
- LabVIEW Dll's are hard-linked to a specific LabVIEW version
- OS specific
- No Plugin Panel support
- No type checking if the function prototype changes.
  (only runtime error)

Note:

In LabVIEW < 8.5 the DLL had to be loaded from start, and could not be unloaded



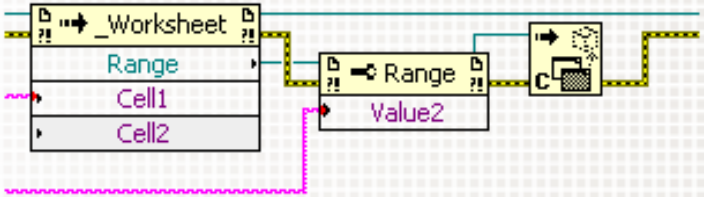
ActiveX

Pros

- Support plugins from other program languages
- Can add panel views from the plugin using ActiveX containers
- Supported by ”most” major Windows applications.

Cons

- Require other environments for maintenance.
- Not full control of memory loading/unloading.
- Can be difficult to know what methods are needed, and in what order.



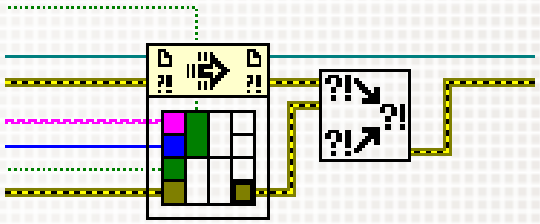
VI Server

Pros

- Custom UI Panels can easily be loaded into the application.
- Available on more than the Windows platform.
- Works on LabVIEW RT
- Even works between targets.

Cons

- More difficult to debug
- Inline operations are more difficult.



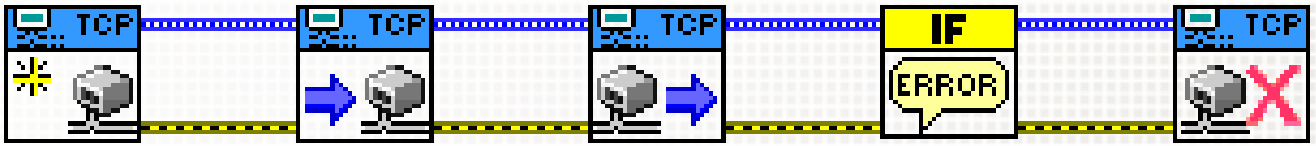
LVOOP

Pros

- Methods can overlay existing methods easily.
- Good protection of private items, (attribute and methods)
- Easy to make inline operations on data
- True parallel execution
- Edit time type checking
- Available on more than the Windows platform, from LV2009 even on LV-RT

Cons

- Can be hard to unload a lvclass plugin from memory.
- Not always easy to debug, e.g. reentrant methods

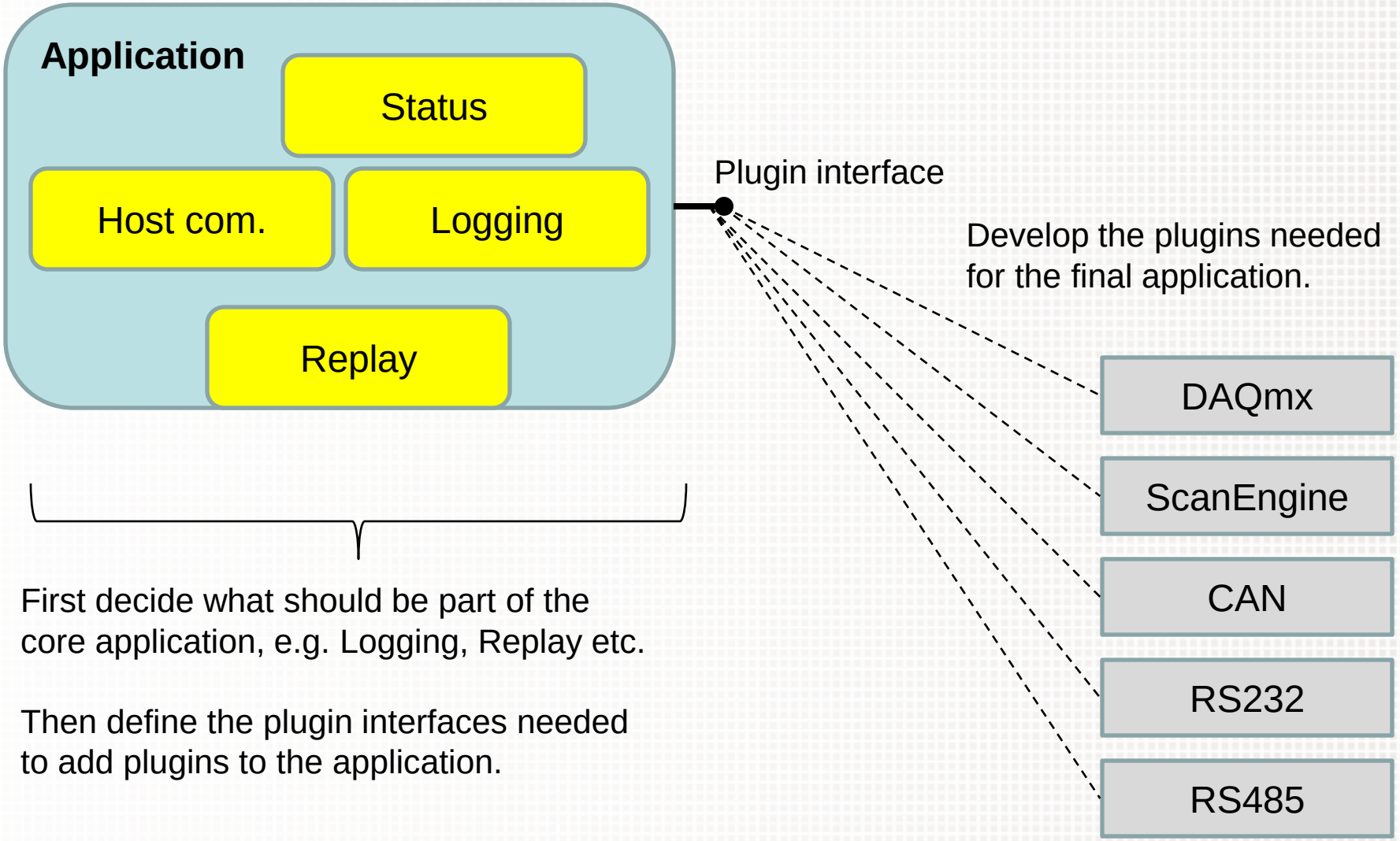


LVOOP vs. VI Server

some differences

Action	LVOOP	VI-Server (call
Skipped interface method	Load the plugin as normal, once called it will use the parent implementation instead	Must be handled as the plugin is loaded, or by creating an empty implementation.
Missing required plugin method	Tracked at edit time by using a property in the lvclass.	This is handled at run-time, and should return a File not exist error.
Parallel execution (reentrant methods)	All plugins must have the same setting (reentrant) for a interface method, but parallel calls are really parallel.	To get true parallel execution we would have to load the method N-times to allow N parallel calls. Also requires housekeeping of the loaded references.
VI prototype checking	If the plugin implementation differs from the interface implementation, the VI is not executable (found at edit time).	Will be detected when the plugin is loaded/called
Loading the plugin	Only loads the .lvclass file with the method ”Get LV Class Default Value”	Each method must be loaded with a separate call to Open-VI-Reference, and multiple times for reentrant methods.

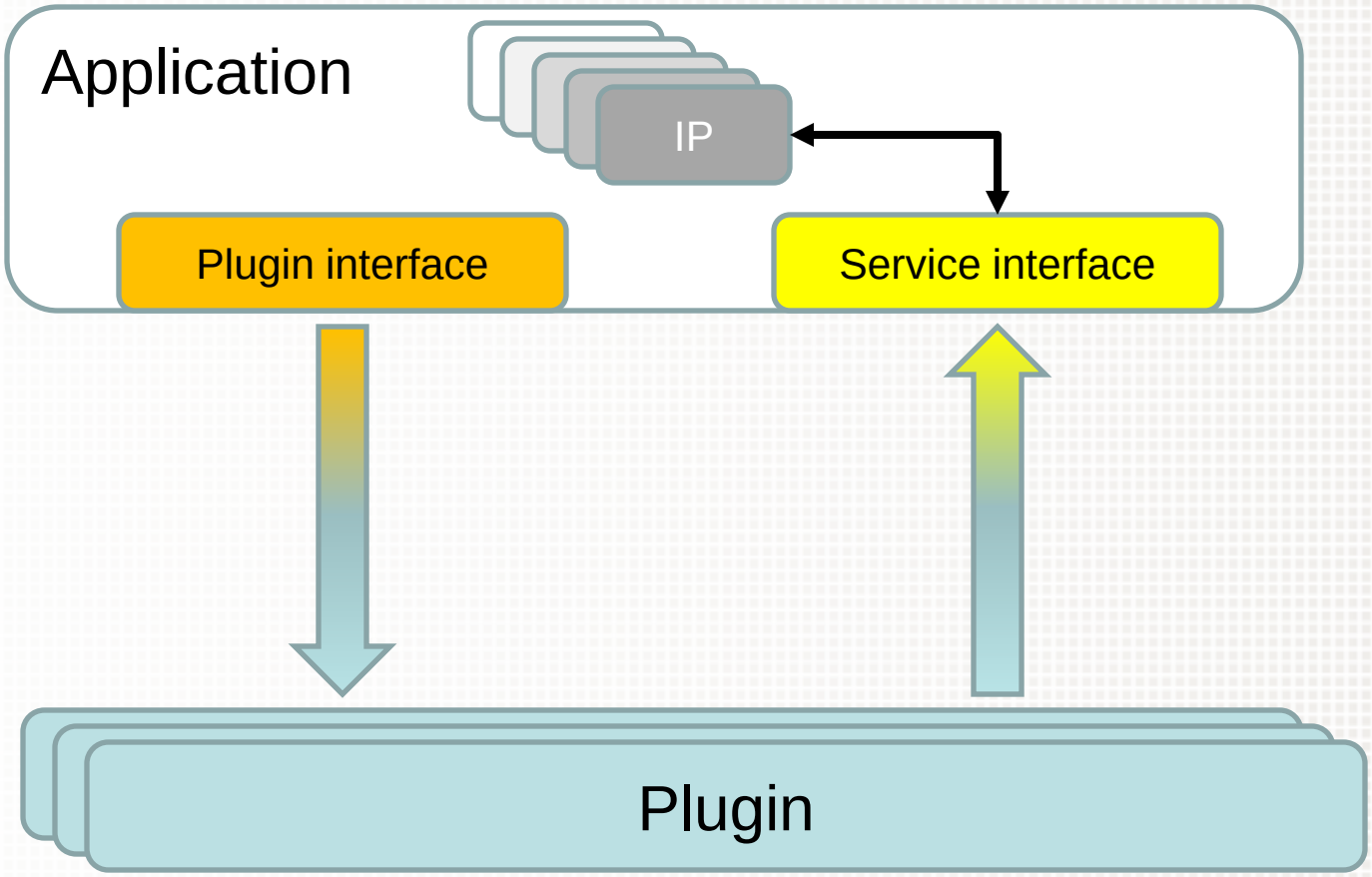
Creating an application



First decide what should be part of the core application, e.g. Logging, Replay etc.

Then define the plugin interfaces needed to add plugins to the application.

Plugin architecture

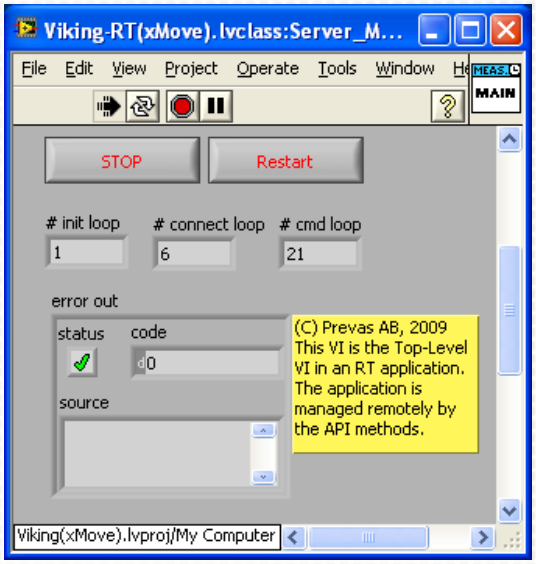


Example

Measurement and control software

Host application

Meas server



Out-of the box

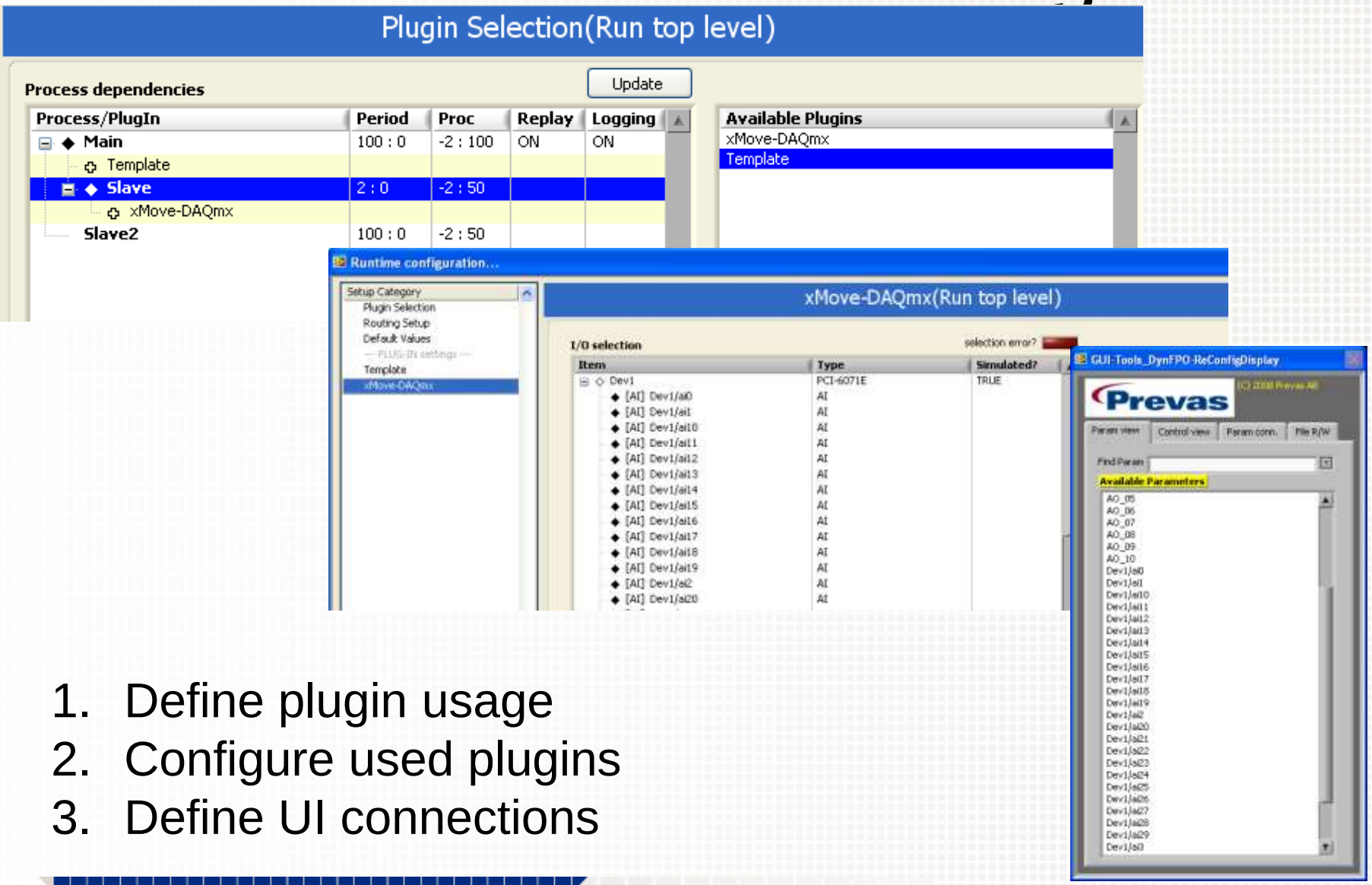
- Host-Server communication
- Logging & Playback
- Plugin configuration
- Possible to save configuration to disk.

Example cont.

Plugins

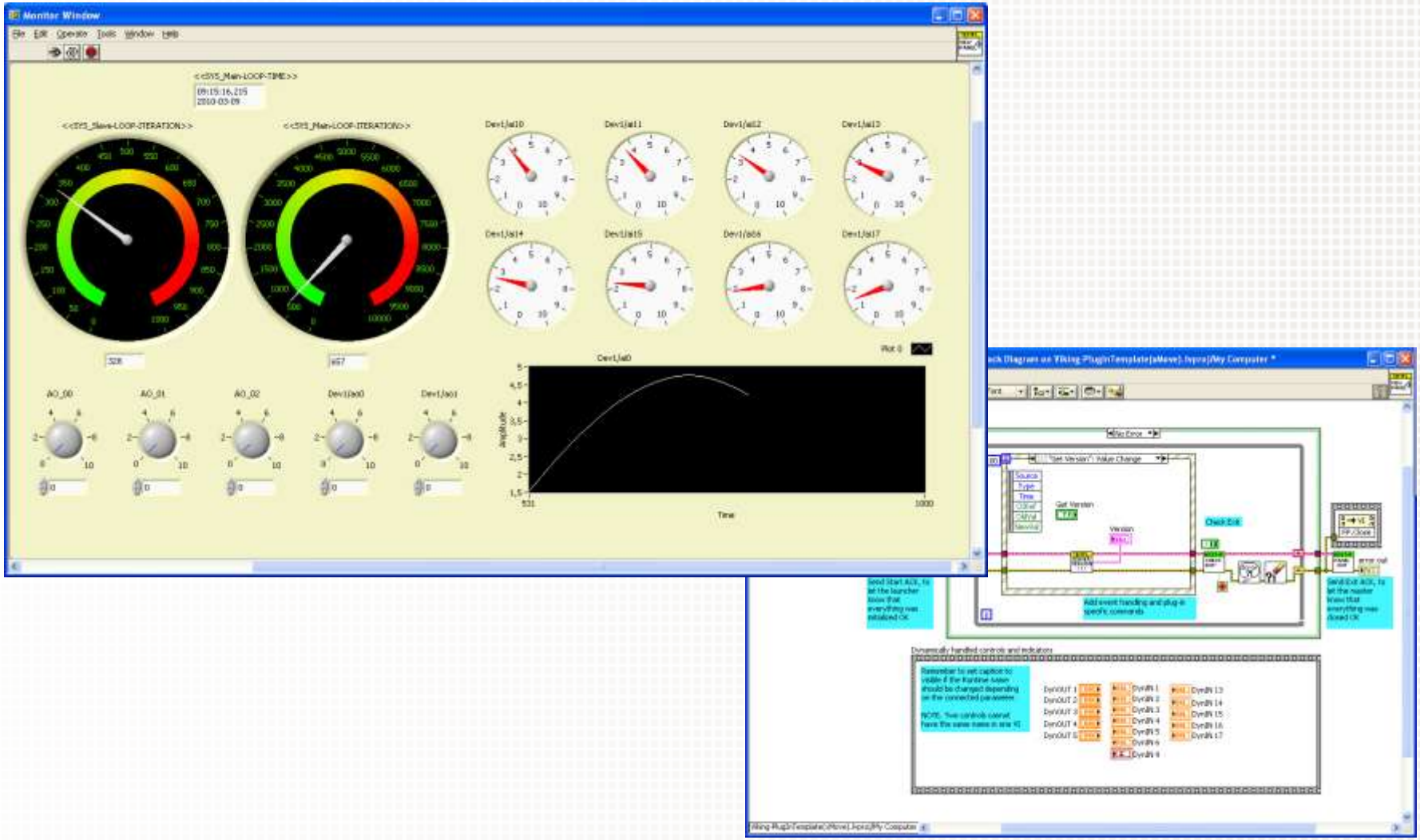
- The platform does not support any HW in the basic application. Instead it is supposed to load plugins for all HW necessary for the current application.
- Plugins are realized as a set of LabVIEW classes, and they have one server part and one GUI part.

Runtime configuration

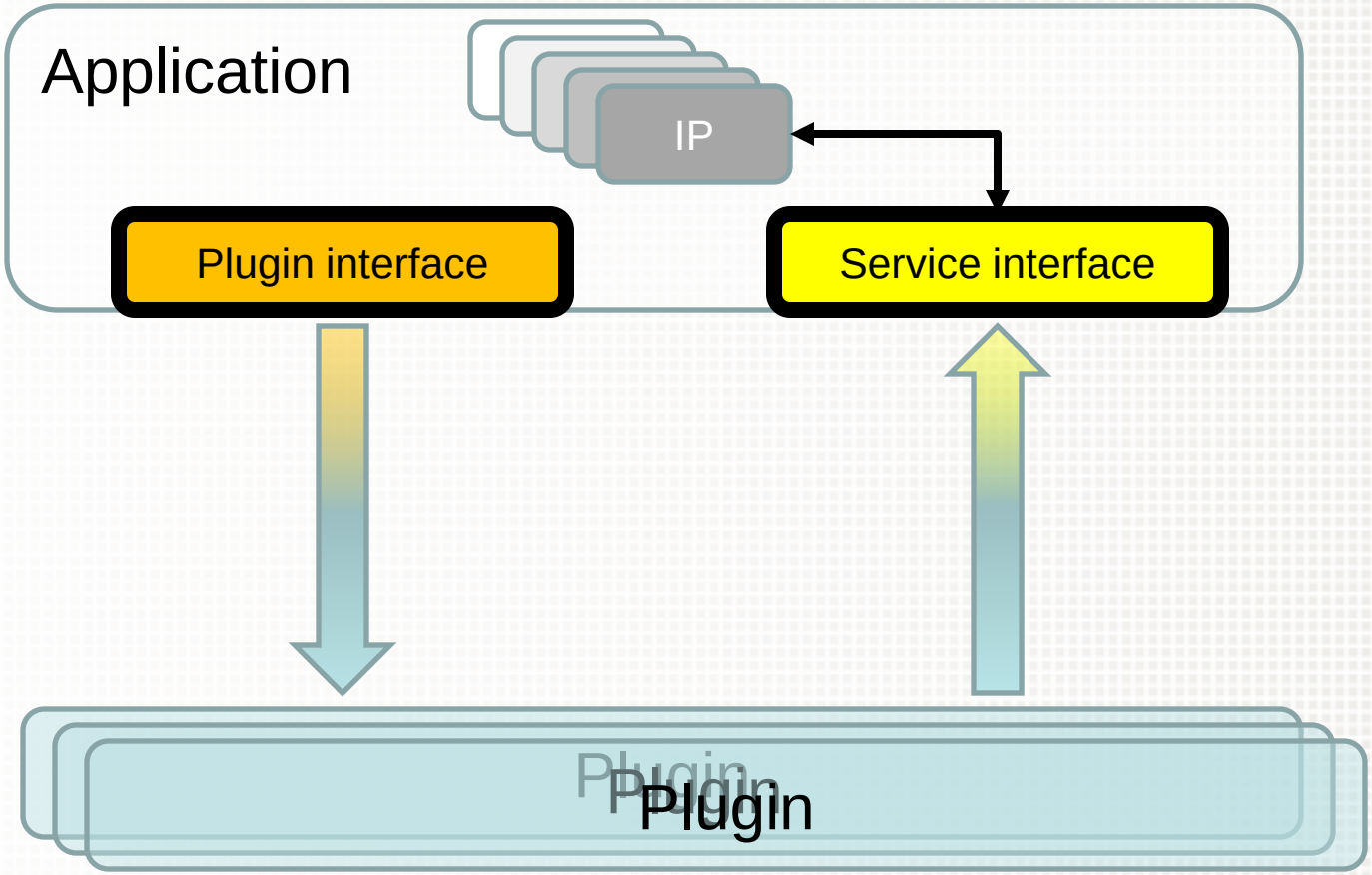


1. Define plugin usage
2. Configure used plugins
3. Define UI connections

Runtime-ViewPanel

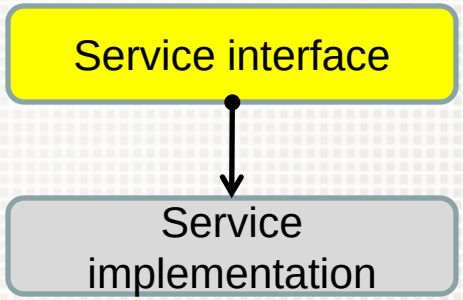


Plugin architecture Details

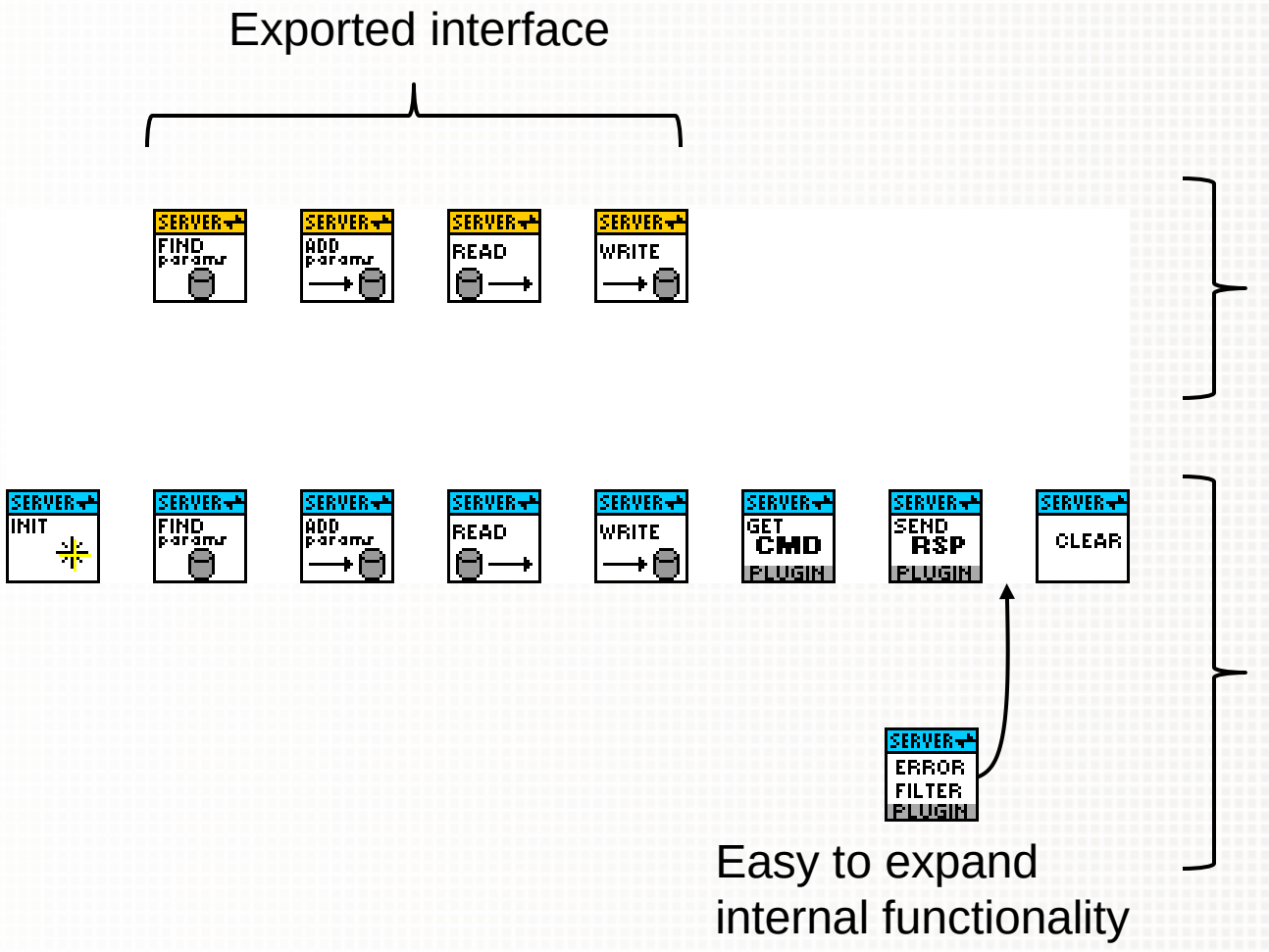


LVOOP Service interface

- A Plugin uses the parent implementation of the Service interface
  - This ensures that the IPs are protected, since they are never exposed to the plugin.
  - The Parent implementation can be more or less empty, but it is a good idea to prepare this in a way that it can be used to test the plugin offline, i.e. outside of the application.
- The application uses a lvclass that inherits from the Service interface.
  - This means that the server overrides the interface implementation with the current implementation in the server. Allowing the Server to expand functionality without breaking the plugin.

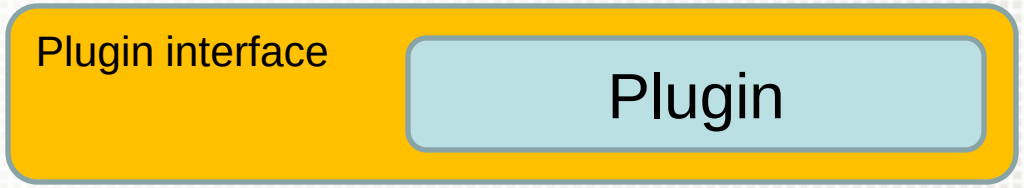


Service interface example

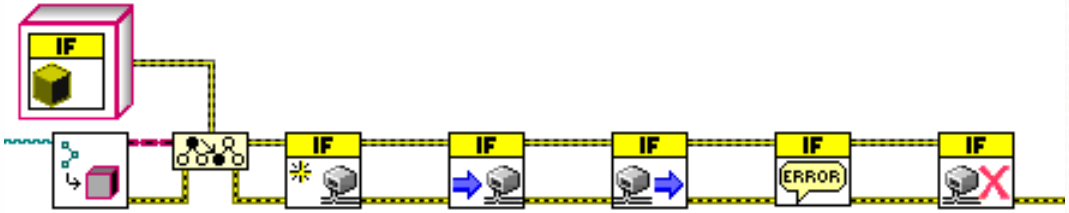


LVOOP Plugin interface

- A Plugin inherits from the Plugin interface class, meaning it can use all public methods of this class, but also that the application itself only needs to know about the interface class.
- The application only calls the ”empty” Plugin interface class methods. If a plugin is loaded into the application, each of these methods will be “replaced” with plugin specific version.
- In the end this means that the application is protected from changes in the plugin implementations, and that the application lifecycle is longer.



Plugin usage



Get LV Class Default Value.vi

The application uses the Plugin interface class methods

Plugin specific class is loaded

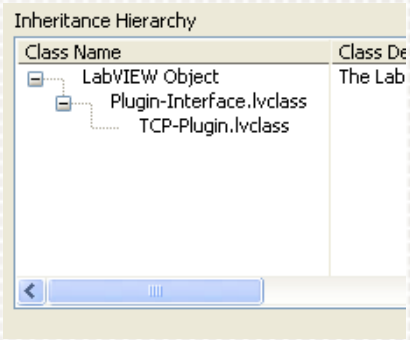
If a plugin is loaded the plugin methods will override the interface methods. If a interface method is not present in the plugin class, the interface method will be used.

Creating a LVOOP plugin

Create plugin project

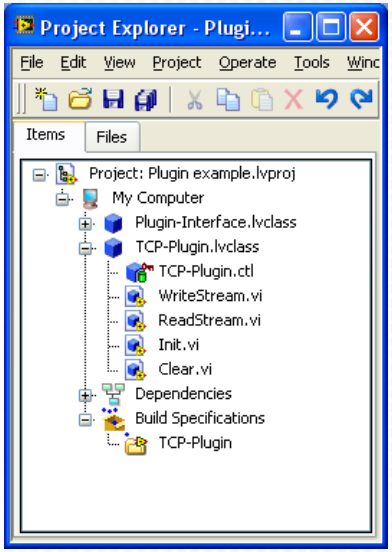
Add interface class

Create plugin class



Edit inheritance

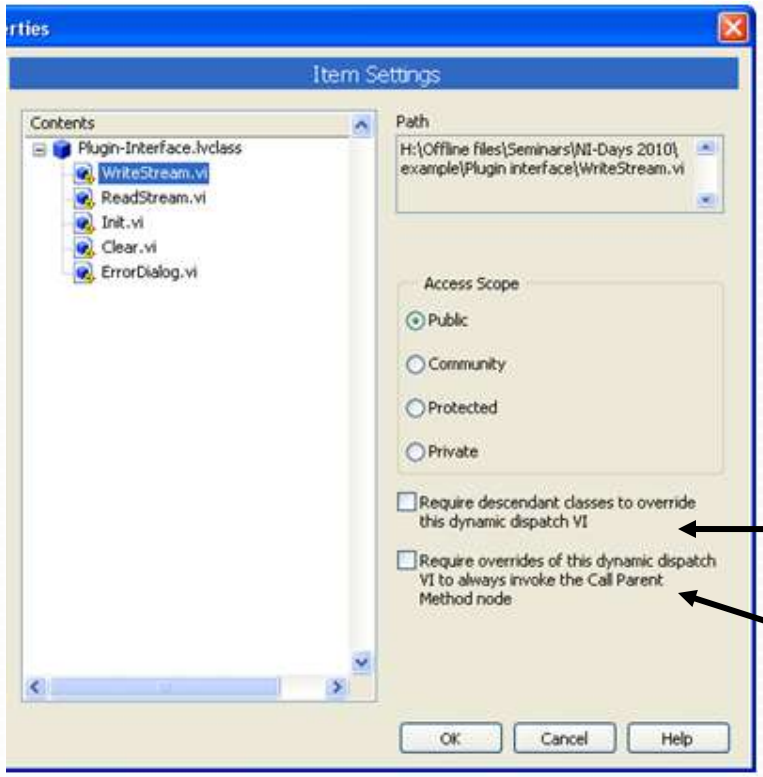
Add plugin specific code and edit appearance



Remember:

Create source distribution for the plugin on the targets where it is intended to be used!

lvclass settings



Specify the scope of the methods in the plugin.

Detect missing methods at edit time.

Set if we require the parent class (interface) method to be executed, e.g. to handle common plugin information.

Questions

Thanks

Johan.Sandquist@prevas.se