How to write a CodeXL Plug-In

Gilad Yarnitzky, Doron Ofek

June 14, 2015

Contents

[How to write a CodeXL Plug-In 1](#_Toc422060478)

[CodeXL Plug-Ins and CodeXL Framework 2](#_Toc422060479)

[Perforce Repository 2](#_Toc422060480)

[Creating the Plug-In Project 2](#_Toc422060481)

[Create a new Visual Studio Project 2](#_Toc422060482)

[Set Project Settings 3](#_Toc422060483)

[Create Plug-In skeleton 4](#_Toc422060484)

[Header File for Export definitions 4](#_Toc422060485)

[Export the 3 Mandatory C Functions 5](#_Toc422060486)

[Deriving from QObject or QObject derived class 6](#_Toc422060487)

[Extending CodeXL 7](#_Toc422060488)

[Extending the CodeXL Menu 7](#_Toc422060489)

[Extending the CodeXL Global Options 7](#_Toc422060490)

[The afGlobalSettingsPage Interface: 8](#_Toc422060491)

[Extending the CodeXL Project Settings 8](#_Toc422060492)

[The afProjectSettingsExtension Interface: 9](#_Toc422060493)

[Handling Events 9](#_Toc422060494)

[Adding Views 10](#_Toc422060495)

[Commands 13](#_Toc422060496)

[Inheriting the Interfaces 13](#_Toc422060497)

[Registering the Class 15](#_Toc422060498)

[Tree handler 15](#_Toc422060499)

[Execution mode 16](#_Toc422060500)

[Linux build with SCons 17](#_Toc422060501)

[SConstruct file 18](#_Toc422060502)

[SConstruct file 20](#_Toc422060503)

[Implement Find? 21](#_Toc422060504)

[Open file 21](#_Toc422060505)

# CodeXL Plug-Ins and CodeXL Framework

CodeXL supports expansion of its functionality via Plug-Ins. A Plug-In is a DLL (or SO on Linux) that can be loaded by the CodeXL application and interacts with CodeXL Framework using the framework classes API.

CodeXL Framework is a collection of classes. Types of framework classes:

* **Interfaces**  
  Plug-In classes inherit interface classes and implement their methods.
* **Base Classes**  
  Plug-In classes inherit base classes, use and extend their functionality.
* **Managers**  
  Plug-In classes register themselves with framework managers. This causes the framework to activate the Plug-In classes when events occur.
* **Utils**Plug-In instanciate and use Utils classes.

# Perforce Repository

Developer Tools source files are stored in this Perforce repository:

stlvswp4p1.amd.com, port 1660

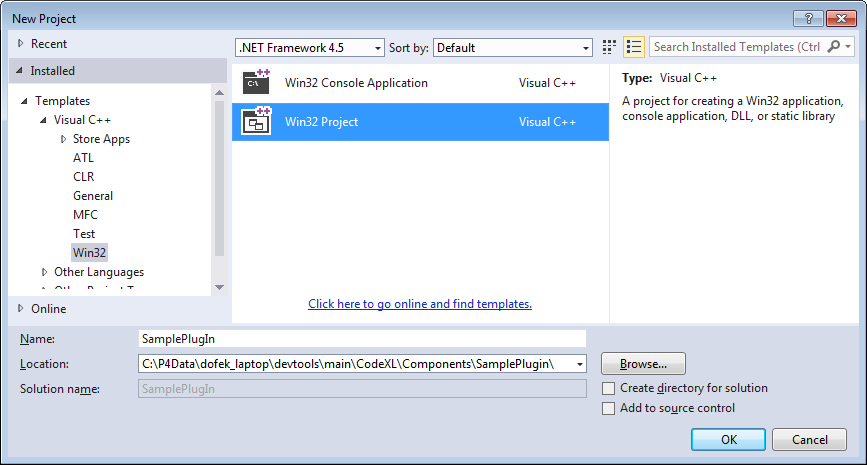
# Creating the Plug-In Project

## Create a new Visual Studio Project

Use Visual Studio 2013.

Create a new project of type Win32.

If you’re working in the DevTools Perforce repository, the path for the project should be in a new folder under the \devtools\main\CodeXL\Components folder, such as “\devtools\main\CodeXL\Components\SamplePlugIn”.



In the Project’s Application Settings, select

* Application Type = DLL
* Additional Options
  + Check ‘Empty Project’
  + Uncheck ‘SDL’

## Set Project Settings

Create a new props file for the Plug-In project. The props file should contain the following:

* Define the CommonDir macro  
  This is the path to the \devtools\main\Common folder, starting with the current project directory, i.e. $(ProjectDir)\*path\_to\_common\_folder*
* Import the CodeXLComponent props file

See the sample props file below. The areas that require editing are highlighted:

<?xml version="1.0" encoding="utf-8"?>

<Project DefaultTargets="Build" ToolsVersion="4.0" xmlns="http://schemas.microsoft.com/developer/msbuild/2003">

<PropertyGroup Label="UserMacros">

<CommonDir>$(ProjectDir)\..\..\..\Common</CommonDir>

</PropertyGroup>

<ItemGroup>

<BuildMacro Include="CommonDir">

<Value>$(CommonDir)</Value>

</BuildMacro>

</ItemGroup>

<ImportGroup Label="PropertySheets">

<Import Project="$(ProjectDir)\..\CodeXLComponent.props" />

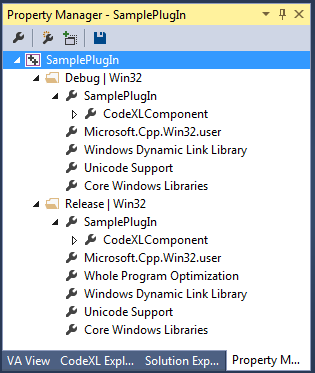
</ImportGroup>

</Project>

Place this file in the same folder as the project file. For example: CodeXL\Components\SamplePlugIn\SamplePlugIn.props

Add the props file to the project using Right-click -> “Add Existing Property Sheet…”

Your project’s Property Manager tree should look like this:



# Create Plug-In skeleton

## Header File for Export definitions

Create a new header file (e.g. SamplePlugInExport.h) for defining dll export macros. Place the file in the same folder as the project file, and add the file to the project.

See the following example:

//=============================================================

// (c) 2014 Advanced Micro Devices, Inc.

//

/// \author Doron Ofek

/// \version $Revision: $

/// \brief Sample Plug-In export macros

//

//=============================================================

// $Id: $

// Last checkin: $DateTime: $

// Last edited by: $Author: $

// Change list: $Change: $

//=============================================================

#ifndef \_\_SAMPLE\_PLUGIN\_EXPORT\_H\_

#define \_\_SAMPLE\_PLUGIN\_EXPORT\_H\_

// Under Win32 builds - define: SAMPLE\_PLUGIN\_API to be:

// - When building SamplePlugIn.dll: \_\_declspec(dllexport).

// - When building other projects: \_\_declspec(dllimport).

#if defined(\_WIN32)

# if defined(AMDTSAMPLEPLUGIN\_EXPORTS)

# define SAMPLE\_PLUGIN\_API \_\_declspec(dllexport)

# else

# define SAMPLE\_PLUGIN\_API \_\_declspec(dllimport)

# endif

#else

# define SAMPLE\_PLUGIN\_API

#endif

#endif // \_\_SAMPLE\_PLUGIN\_EXPORT\_H\_

## Export the 3 Mandatory C Functions

Each CodeXL plug-in must export 3 C functions, as follows

* **CheckValidity**  
  Signature: int CheckValidity(gtString& errString)  
  Purpose: Verify that the plug-in’s prerequisites are met. This is the first function that is called.  
  Description: This function checks if the preconditions that the plug-in relies on are met. For example, plug-ins that require OpenCL should verify that OpenCL is installed.  
  This function should return zero if successful. Otherwise an error code should be returned and a description of the error should be assigned to the errString parameter.
* **Initialize**  
  Signature: void initialize()  
  Purpose: Create objects that interact with the CodeXL framework manager objects.  
  Description: Typical initializations that occur in this function:
  + Create and register the plug-in's specific main menu actions creator
  + Initialize backend modules specific to this plug-in
  + Create and register the plug-in's specific node in the global settings page
  + Create and register the plug-in's specific node in the project settings page
  + Create and register the plug-in's specific execution mode
  + Create event observers and register them with framework managers
  + Create, init and register the plug-in's specific views creator
* **InitializeIndependentWidgets**  
  Signature: void initializeIndependentWidgets()  
  Purpose: Initialize other items after main window creation.  
  Description: This function initializes all the widget items that are not registered with the creators mechanism. These widgets are responsible for their own callbacks and strings.

# Deriving from QObject or QObject derived class

In your project you will need to derive from QObject in order to do this you will need to do the following steps

* Add the source and header file to the project and save the project file.
* Derive from the needed class
* Add the Q\_OBJECT to the class definition

class AC\_API pluninDerivedClass: public QObject

{

Q\_OBJECT

}

* The class needs have a “Moc” build step. This needs to be done in release and debug. Open the project vcxproj in a text editor and replace the

<ClInclude Include="Include\pluninDerivedClass.h" />

with

<CustomBuild Include="Include\pluninDerivedClass.h ">

<Command>"$(AMDTCommonExt)Qt\5.3\bin\moc.exe" "Include\%(Filename).h" -o "tmp\moc\_$(Platform)$(Configuration)\moc\_%(Filename).cpp"</Command>

<Message>moc creation</Message>

<Outputs>tmp\moc\_$(Platform)$(Configuration)\moc\_%(Filename).cpp</Outputs>

<AdditionalInputs>$(AMDTCommonExt)Qt\5.3\bin\moc.exe;src\%(Filename).cpp</AdditionalInputs>

</CustomBuild>

This is assuming Qt5.3 is used. If a more advanced version is used, replace it with the current version that is used.

* Add the “Moc generated file to the project, in the **vcxporj** file where all the other source files are added to the project add the following line

<ClCompile Include="tmp\moc\_$(Platform)$(Configuration)\moc\_pluninDerivedClass.cpp" />

* Add the “Moc generated file to the project, in the **filters** file where all the other source files are added to the project add the following line

<ClCompile Include="tmp\moc\_$(Platform)$(Configuration)\moc\_acSourceCodeView.cpp">

<Filter>Generated Files</Filter>

</ClCompile>

In Linux this is handled in the SConscript file and is described in the [Linux build with Scons](#_Linux_build_with) section

# Extending CodeXL

## Extending the CodeXL Menu

Create a class that is derived from afActionExecutorAbstract that will be the “Actions Creator” class

Register the class in the with afQtCreatorsManager

The code will look something like that:

// Create the main menu actions creator:

pluginMenuActionsExecutor\* pActionsCreator = new pluginMenuActionsExecutor;

// Register the actions creator:

afQtCreatorsManager::instance().registerActionExecutor(pActionsCreator);

All the virtual functions of the afActionExecutorAbstract must be implemented:

virtual void handleTrigger(int actionIndex): Handle the actual execution of the command.

virtual void handleUiUpdate(int actionIndex): Handle the UI updates of the command. If the command is enabled, disabled, visible and checked.

All the virtual functions of the afActionCreatorAbstract (base class of afActionExecutorAbstract) must be implemented:

virtual bool actionText(int actionIndex, gtString& actionText, gtString& tooltip, gtString& keyboardShortcut): Sets the command text string, tooltip and command short cut.

virtual gtString menuPosition(int actionIndex, afActionPositionData& positionData): Defines the place of the commands in the menu using the afActionPositionData. The command can be set relative to other commands already created before this command.

virtual gtString toolbarPosition(int actionIndex): Position of the command in a toolbar.

virtual void groupAction(int actionIndex): In case several commands need to be grouped in this component it should be executed in this command. (hardly used).

## Extending the CodeXL Global Options

The global options are options that are not specific to the current project.

In order to add a page to the global options the following steps needs to be taken:

* The afGlobalSettingsPage needs to be derived.
* Its interfaces needs to be implemented.
* When creating the plugin the new options page needs to be registered.

When implementing the new options page remember that the afGlobalSettingPage is derived from QGroupBox and all your widgets and layouts have a base widget to be placed into.

### The afGlobalSettingsPage Interface:

virtual void initialize() : Initialize the settings page. Create all the needed controls and place them in the needed layouts and give the settings page the needed look.

virtual gtString pageTitle() : Give the page a title. This must be a unique name and not match any other existing page titles.

virtual gtString xmlSectionTitle() : Give the xml section name where the data of the page will be saved.

virtual bool getXMLSettingsString(gtString& projectAsXMLString) : Build the xml string that represents the data in the settings page.

virtual bool setSettingsFromXMLString(const gtString& projectAsXMLString) : Build the settings page data from the xml strings.

virtual void loadCurrentSettings() : Load he current settings. This is not the same as the setSettingsFromXMLString, which is restoring the data from the xml file when the application is loaded. The loadCurrentSettings is loading the data from the application data.

virtual void restoreDefaultSettings() : Restore the data if the user pressed cancel in the options dialog.

virtual bool saveCurrentSettings() : Save the current settings when the user pressed ‘ok’.

virtual bool IsPageDataValid() : Validate that the data in the page is valid before accepting the ‘ok’. If the return value is false the dialog will not close. In this function you will need to give the user what needs to be fixed in order to solve the problem.

The registration is done in the plugin [void initialize()](#_Export_the_3)function:

// Create and register the global settings page:

m\_ppluginOptionsPage = new pluginOptionsPage;

afGlobalVariablesManager::instance().registerGlobalSettingsPage(m\_ppluginOptionsPage);

## Extending the CodeXL Project Settings

The project settings are specific the current open project.

In order to add a page to the project settings the following steps needs to be taken:

* The afProjectSettingsExtension needs to be derived.
* Its interfaces needs to be implemented.
* When creating the plugin the new project settings page needs to be registered.

When implementing the new options page remember that the afProjectSettingsExtension is derived from QGroupBox and all your widgets and layouts have a base widget to be placed into.

Many of the functions are similar to the afGlobalSettingsPage interface.

### The afProjectSettingsExtension Interface:

virtual void initialize() : Initialize the settings page. Create all the needed controls and place them in the needed layouts and give the settings page the needed look.

virtual gtString ExtensionXMLString() : Give the xml section name where the data of the page will be saved.

virtual gtString ExtensionTreePathAsString() : Returns the position of the settings page in the dialog. This is done in a level representation, where each level is marked by a ‘,’. For example:

|  |  |
| --- | --- |
|  | The “Genera and Optimization” item is defined as:  "Analyze, Kernel/Shader Build Options, General & Optimization" |

virtual bool GetXMLSettingsString(gtString& projectAsXMLString) : Build the xml string that represents the data in the settings page.

virtual bool SetSettingsFromXMLString(const gtString& projectAsXMLString) : Build the settings page data from the xml strings.

virtual void RestoreDefaultProjectSettings(): Restore the data if the user pressed cancel in the options dialog.

virtual bool AreSettingsValid(gtString& invalidMessageStr) : Validate that the data in the page is valid before accepting the ‘ok’. If the return value is false the dialog will not close. In this function you will need to give the user what needs to be fixed in order to solve the problem.

virtual bool SaveCurrentSettings(): Save the current settings when the user pressed ‘ok’.

virtual bool RestoreCurrentSettings(): Restore the data if the user pressed cancel in the options dialog.

virtual bool ShouldAddToProjectSettingDialog() : In settings page is a base class for other pages and should not appear in the setting dialog this should return false.

## Handling Events

CodeXL has an event mechanism to handle apEvent derived events.

There are many type of events, some are generic events that the plugin might want to handle, like main object tree item selection and activation events, and some are more specific that the plugin is less likely to address like specific profiling events.

The handling of events is done in a class derived from apIEventsObserver, and implementing its interface:

virtual void onEvent(const apEvent& eve, bool& vetoEvent) : The function that will handle all the events that are passed in the system. You must be aware that all the events will pass throw this function and must handle only those you mean to handle.

virtual const wchar\_t\* eventObserverName() const : give a unique name to your event observer.

Once the event handle class is defined it needs to register. Usually this is done in the constructor of the class so it can be done in the following way (passing the “\*this” parameter)

The second parameter in the registration is the priority of the observer. The higher priority handlers get the event first to handle it and then the lower priority handlers. As the handlers take action they can veto the event in order to prevent other handlers from taking farther action on it.

These are the levels of priority:

AP\_PERSISTENT\_DATA\_MANAGER\_EVENTS\_HANDLING\_PRIORITY,

AP\_API\_TO\_SPY\_CONNECTOR\_EVENTS\_HANDLING\_PRIORITY,

AP\_GLOBAL\_VARIABLES\_MANAGER\_EVENTS\_HANDLING\_PRIORITY,

AP\_PERFORMANCE\_COUNTERS\_READERS\_EVENTS\_HANDLING\_PRIORITY,

AP\_APPLICATION\_FRAMEWORK\_EVENTS\_HANDLING\_PRIORITY,

AP\_APPLICATION\_COMPONENTS\_MANAGERS\_EVENTS\_HANDLING\_PRIORITY,

AP\_APPLICATION\_COMPONENTS\_EVENTS\_HANDLING\_PRIORITY,

// Register to recieve debugged process events:

apEventsHandler::instance().registerEventsObserver(\*this, AP\_PROCESS\_DEBUGGER\_EVENTS\_HANDLING\_PRIORITY);

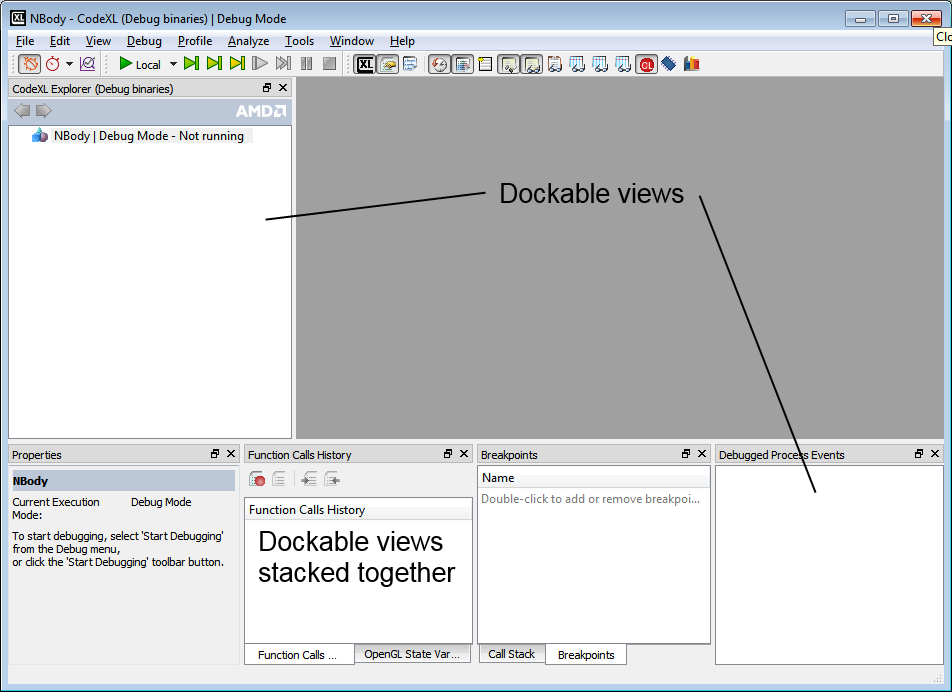
Setting a veto on an event:

In the implementation of the interface void onEvent(const apEvent& eve, bool& vetoEvent) you can set the vetoEvent to true when handling a specific event. This will veto farther action on the event

## Adding Views

A plugin can have as many views as it wants. There are two types of views

Dockables views: Views can be docked at the sides of the main client area or float. When they are docked they can be docked together and selectable by a tab control, or one next to the other. These views are automatically created when CodeXL is opened. They can be hidden or shown based on the active component but they always exist.



MDI views: Views that are only created when a specific document is requested to be opened by CodeXL and will be visible in the client area.

The views are created using a view creator objects that derives from afQtViewCreatorAbstract, which derive from afViewCreatorAbstract. Both interfaces need to be implemented.

A creator can create several views and each interface will have an index that will help identify which view that interface is referring to.

A plugin might have several views creators. This can help organize the code logic. There is no limit how many views can be created in a view creator and how many views creators can be in a plugin.

afQtViewCreatorAbstract & afViewCreatorAbstract interfaces that need and can be implemented:

void titleString(int viewIndex, gtString& viewTitle, gtString& viewMenuCommand) : Get the title of the of the view and the command title associated with the view.

gtString executionMode(int viewIndex) : The views show/hide menu item are grouped together by the “execution” mode such as “Debug”, “Profile”, Etc. If no mode is defined, then the view menu item appears directly under the “View” menu. This interface returns to which group this view should be added to. Can be not implemented.

gtString modeMenuPosition(int viewIndex, afActionPositionData& positionData) { (void)(viewIndex); (void)(positionData); return L""; } : In case an execution mode is supplied, a sub menu can be also provided for the execution mode menu. Can be not implemented. positionData not used, for future use.

bool addSeparator(int viewIndex) : Should a separator be added after the view command.

gtString associatedToolbar(int viewIndex) : Set the associated toolbar with the view. When the view will be visible, this toolbar will be displayed if it is not. If there is no toolbar associated return an empty string.

afViewType type(int viewIndex) : Defines if the view is MDI or Dockable. Return AF\_VIEW\_dock or AF\_VIEW\_mdi.

int dockArea(int viewIndex) : Return the initial docking area of the view based on the values of afDockingAreaFlag.

gtString dockWith(int viewIndex) : If the dock with viewIndex is a dockable view and we initially want to dock it with another view then the return value is the view it will dock with. If the string will have a prefix of “-“ the view will open below the indicated view. Can be not implemented.

bool isDynamic() : does the view creator create views that are MDI views. If this is true then NO dockable views can be created by this views creator! So it is highly recommended not to mix dockable views creation with mdi views creation.

QDockWidget::DockWidgetFeatures dockWidgetFeatures(int viewIndex) : return the dock widget features of the dockable view. Recommended to return QDockWidget::AllDockWidgetFeatures

QWidget\* createQtWidgetWrapping(int viewIndex) : If there is a need to create a wrapping QWidget for an MDI view then use this function. Can be used if the MDI creates different QWidgets that needs to be accessed by a single known wrapper QWidget.

bool createQTWrapping(int viewIndex, QMainWindow\* pMainWindow) : For future use. MUST return true. Implement something like this:

{ (void)(viewIndex); (void)(pMainWindow); return true;};

bool createViewContent(int viewIndex, QWidget\*& pContentQWidget, QWidget\* pQParent) : Create the view content into pContentQWidget. This is the core of the creation of the creator, where the QWidget itself is created.

QSize initialSize(int viewIndex) : Get initial size of the view for dockable views. However if the controls will require larger size, then the value will be overwritten.

bool visibility(int viewIndex) : Get the initial visibility flag for dockable views.

bool initiallyActive(int viewIndex) : make sure that the view is active (raised and visible in case it is with overlapped with sibling views.

int amountOfViewTypes() : Get the number of views this view creator creates.

QPixmap\* iconAsPixmap(int viewIndex) : Get the icon of the view.

const gtString CreatedMDIType() const : Get the MDI to match the apMDIViewCreateEvent::CreatedMDIType so when opening an MDI view it will be known it belongs to this creator. More will be explained in the [Open file](#_Open_file) section.

The next two functions refer to command handling that will be explained in more detail in the [Commands](#_Commands) section.

void handleTrigger(int viewIndex, int actionIndex) : Handle the command execution that relate to the view.

void handleUiUpdate(int viewIndex, int actionIndex) : Handle the command update the relate to the view.

Once the view creator is defined it needs to register like all the other interfaces. This is done in a similar way to a view creator manager but with an extra action:

// Create a debug views creator:

m\_pPluginViewCreator = new pluginViewCreator;

// Initialize (initialize the buffer that holds the views icons)

m\_pPluginViewCreator->initialize();

// Register the views creator:

afQtCreatorsManager::instance().registerViewCreator(m\_pPluginViewCreator);

## Commands

Commands are the added to the main menu. There are two types of commands, both of the commands appear in the main menu in the same way and will be implemented by the same interface.

Global commands: Commands that the state will be checked regardless of the current active view (view that has the keyboard focus). If the command is enabled and executed then the execution will be carried out directly not going through the current active view or through the current active view creator

Local Commands: Commands that the state of the view will be checked according to the current active view. A good example would be “Select”. If the current active view does not enable select that this command will be disabled. The Execution will be handled by the current active view. So a “Paste” command will be executed in the current active view.

In order to create Global commands the following steps needs to be done:

* Derives from afActionExecutorAbstract (that derives from afActionCreatorAbstract).
* Implement its interfaces
* Register the class

If you want to create Local commands then the following steps need to be done:

* Derives from afActionCreatorAbstract. (notice the difference)
* Implement its interfaces
* In the views creator class connect between the views creator and the command creator

### Inheriting the Interfaces

afActionCreatorAbstract interface:

virtual void populateSupportedCommandIds() : This interface adds the ids of the supported command to the m\_supportedCommandIds member. Simply add the ids using push\_back command. For example:

// fill the vector of supported command ids:

m\_supportedCommandIds.push\_back(ID\_CUT);

m\_supportedCommandIds.push\_back(ID\_COPY);

From now on the index will refer to the index in the m\_supportedCommandIds vector so in the above example ID\_CUT will have actionIndex = 0 and ID\_COPY will have actionIndex = 1.

In all the interfaces the actionIndex is used, if you want to get the commandID then you can use the int actionIndexToCommandId(const int actionIndex) const; API that will convert the action index to the command ID

bool actionText(int actionIndex, gtString& actionText, gtString& tooltip, gtString& keyboardShortcut) : Get the command text, tooltip and accelerator of the command.

gtString menuPosition(int actionIndex, afActionPositionData& positionData) : Get the menu position. The string is composed of menu levels separated by “/”. For example if command is “start” in “Debug” menu then the return value should be “Debug”. However if the command is in a submenu “Actions” in the “Debug” menu then the return value would be “Debug/Actions”.

afActionPositionData& positionData Gives additional control over the positioning of the menu item:

afCommandPosition: Is the command in the Start/Middle/End block of the menu.

afCommandSeparatorType: Should the command have separator

Should the command come before another action: This is identified by two parameters:

gtString m\_beforeActionMenuPosition: the menu position of the command

gtString m\_beforeActionText: the command text

virtual gtString toolbarPosition(int actionIndex) : Get the name of the toolbar name that the command belongs to. If no string was supplied (empty string) then the command will not be added to any toolbar.

void groupAction(int actionIndex) : If you need to group actions together then overwrite this function. Grouped actions are actions that only one of them can be active at a time. Does have to be implemented.

int separatorPosition(int nActionID) : If a toolbar separator needs to be added before/after the toolbar command then overwrite this function. Returnvalue of -1 indicates separator before, return value of 1 indicates separator after. Does have to be implemented.

QPixmap\* iconAsPixmap(int actionIndex, bool& shouldUseInMenu) : Get the pixmap of the command and if it should be used in the menu (it will be used in the toolbar if a toolbar string was supplied).

void initActionIcons() : Overwrite this function if the actions are using one of the predefined icons. If they do, for each command that does use the function to connect between the two: initSingleActionIcon(commands id, frame work icon index); .Does have to be implemented.

afActionExecutorAbstract interface:

void handleTrigger(int actionIndex) : Handles the selection of the action by the user. In this function the actual execution of the command is done.

void handleUiUpdate(int actionIndex) : The updating of the command is done in this action. This means the enable/disable of the command, changing the text if needed, hiding, and any other display manipulation that is allowed on QAction.

To get the QAction associated with the command you simply need to do the following:

QAction\* pAction = action(actionIndex);

And on the pAction you can use any Qt API to set the QAction attributes

### Registering the Class

There is a difference when handling Global commands implementation that needs registering. This is done in the following way:

The registration is done in the same way as other registrations: Create the object and register it:

pluinMenuActionsExecutor\* pActionsCreator = new pluinMenuActionsExecutor;

// Register the actions creator:

afQtCreatorsManager::instance().registerActionExecutor(pActionsCreator);

A Local command objects is not registered but is connected to the views creator in the views creator constructor throw a special member that the view creator has and is done in the following way:

// Create the view actions creator:

\_pViewActionCreator = new pluginLocalActionCreator;

\_pViewActionCreator->initializeCreator();

## Tree handler

The plugin tree handler manages the interaction with the main objects tree.

The basic inheritance is from afApplicationTreeHandler and implementing the following functions:

bool BuildContextMenuForItems(const gtList<const afApplicationTreeItemData\*>, QMenu& menu) : Build the context menu item for the list of afApplicationTreeItemData which are the items in the selected tree nodes. Return true if there is a context menu for those list of tree item data.

afApplicationTreeItemData\* FindMatchingTreeItem(const afApplicationTreeItemData& displayedItemId) : Find an existing matching tree item for the supplied afApplicationTreeItemData.

bool BuildItemHTMLPropeties(const afApplicationTreeItemData& displayedItemId, afHTMLContent& htmlContent) : For the supplied displayedItemId build an htmlcontent. Return true only if this tree handler created html data for the supplied tree item data.

void SetItemsVisibility() : Sets the items visibility in the tree. Passing through all/part of the items and hide/show items based on the active mode or any other criteria.

Usually a tree handler will be an event handler, will implement onEvent(const apEvent& eve, bool& vetoEvent) and will handle tree related events which are:

apEvent::GD\_MONITORED\_OBJECT\_SELECTED\_EVENT : Handle selection of tree item.

apEvent::GD\_MONITORED\_OBJECT\_ACTIVATED\_EVENT : Handle activation of tree item (double click)

Both in files apMonitoredObjectsTreeEvent.h/.cpp

After creating the Tree handler at some stage in its creation (usually in the constructor) it register itself in the application tree. This is done with a code similar to the following code:

afApplicationCommands\* pApplicationCommands = afApplicationCommands::instance();

GT\_IF\_WITH\_ASSERT(pApplicationCommands != nullptr)

{

m\_pApplicationTree = pApplicationCommands->applicationTree();

GT\_IF\_WITH\_ASSERT(m\_pApplicationTree != nullptr)

{

// Should add a signal to main window that the tree was created, and register all tree handler when handling this signal

m\_pApplicationTree->registerApplicationTreeHandler(this);

}

}

## Execution mode

A plugin can add an execution mode (similar to debug, profile and analyze that already exist in CodeXL). In order to do that a class derived from afIExecutionMode needs to be implemented and registered.

gtString modeName() : Get the mode name include the “mode” word.

gtString modeActionString() : The name of the action the mode encompasses (e.g. "Debugging", "Profiling", "Analysis", etc.)

virtual gtString modeVerbString() : The action verb the mode encompasses (e.g. "debug", "profile", "analyze", etc.)

gtString modeDescription() : Mode description for tooltips

bool ExecuteStartupAction(afStartupAction action) : Perform a startup action. Return true iff the mode support the requested action. For possible start up action see the afIExecutionMode.h file

bool IsStartupActionSupported(afStartupAction action) : Return true iff the mode support the requested action:

bool IsStartupActionSupportedWithNoProject(afExecutionCommandId commandId) : Return true iff the execution mode supports the requested action when no project is loaded. Does not need to be implemented.

bool IsRemoteEnabledForSessionType(const gtString& sessionType) : Return true iff the execution mode supports remote host scenario for the requested session type. Does not need to be implemented.

gtString selectedSessionTypeName() : A mode can have more than one session type (for example profiling have several CPU profiling types and GPU profiling types). This method returns the current selected session type. Does not need to be implemented.

void execute(afExecutionCommandId commandId) : Execute the mode command. For the available execution commands see afIExecutionMode.h file.

void updateUI(afExecutionCommandId commandId, QAction\* pAction) : Handle the command update UI. This includes enable/disable, hide/show or other options that QAction allow.

void execute(int sessionTypeIndex) : Execute session type change. Does not need to be implemented.

void updateUI(int sessionTypeIndex, QAction\* pAction) ) : Handle the Session command update UI. This includes enable/disable, hide/show or other options that QAction allow. Does not need to be implemented.

int numberSessionTypes() Get the number of session types. Does not need to be implemented.

gtString sessionTypeName(int sessionTypeIndex) : Get the session name at a specific index.

QPixmap\* sessionTypeIcon(int sessionTypeIndex) : Get the session pixmap icon at a specific index.

int indexForSessionType(const gtString& sessionType) : Get the index for a specific session from a string (reverse of sessionTypeName(int sessionTypeIndex)).

afMainAppWindow::LayoutFormats layoutFormat() : Each mode has a specific layout of windows it uses. This function returns the layout to be used for this mode. At this stage new modes for plugins cannot be added so a layout of the available ones must be used.

gtString ProjectSettingsPath() : The strings that identify the settings for this mode in the XML must be unique and not collide with other modes strings.

bool isModeEnabled(): if this mode is enabled at all. Does not need to be implemented.

gtString HowToStartModeExecutionMessage() : Get the properties view message to start the execution of the mode. Default is a generic message.

Once the interface is implemented then the modes need to be registered:

m\_ppluginExecutionMode = new pluginExecutionMode;

afExecutionModeManager::instance().registerExecutionMode(m\_ ppluginExecutionMode);

## Linux build with SCons

In order to add a plugin to the Linux build you have to do two things

* Create an SConscript file
* Add it to the CodeXL main SConstruct file

### SConstruct file

The SConstruct file is constructed from several sections

Here is a sample SConstruct that is the basis for all SConstruct files and can be expended (the top section which has the Init and Uses might be different as will be explained)

# -\*- Python -\*-

Import('\*')

from CXL\_init import \*

libName = "AMDTPlugin"

env = CXL\_env.Clone()

initBoost(env)

UseBoost(env)

UseTinyXml(env)

UseQt4(env)

UseQScintilla(env)

env.Append( CPPPATH = [

".",

"../",

"./Include",

"./src/",

env['CXL\_commonproj\_dir'],

env['CXL\_common\_dir'] + '/Some in the common dir',

])

moc\_files = Split(

" src/pluginSrc1.h"

+ " src/pluginSrc2.h"

)

sources = \

[

"src/pluginSrc1.cpp",

"src/pluginSrc2.cpp",

]

commonLinkedLibraries = \

[

"AMDTBaseTools",

"AMDTOSWrappers",

"libpthread"

]

# Building moc content

MOC\_Generated = []

for moc\_file in moc\_files:

MOC\_Generated += env.MocBld(moc\_file)

# Contains all linked libraries:

linkedLibraries = commonLinkedLibraries

env.Prepend (LIBS = linkedLibraries)

# Creating object files

objFiles = env.SharedObject(sources + MOC\_Generated)

# Creating shared libraries

soFiles = env.SharedLibrary(

target = libName,

source = objFiles)

# Installing libraries

libInstall = env.Install(

dir = env['CXL\_lib\_dir'],

source = (soFiles))

Return('libInstall')

The top section just include from the main script all the function

# -\*- Python -\*-

Import('\*')

from CXL\_init import \*

Define the library name

libName = "AMDTPlugin"

Clone the environment locally

env = CXL\_env.Clone()

use the different Libraries (in case of boost also init it since it is not globally initialized in the main SConstruct). Here More or less libraries can be defined as used. Most of the libraries do not need to be initialized. In the CodeXL SConstruct the default initialized libraries can be seen.

initBoost(env)

UseBoost(env)

UseTinyXml(env)

UseQt4(env)

UseQScintilla(env)

The include path are defined. Notice how the common project and common directories are used.

env.Append( CPPPATH = [

".",

"../",

"./Include",

"./src/",

env['CXL\_commonproj\_dir'],

env['CXL\_common\_dir'] + '/Some in the common dir',

])

Files that needs to generate moc files are defined:

moc\_files = Split(

" src/pluginSrc1.h"

+ " src/pluginSrc2.h"

)

Source files that need to be compiled as part of the libraries are defined:

sources = \

[

"src/pluginSrc1.cpp",

"src/pluginSrc2.cpp",

]

Libraries that are linked against are defined:

commonLinkedLibraries = \

[

"AMDTBaseTools",

"AMDTOSWrappers",

"libpthread"

]

The compile, link and build stages are set. Those steps are the same with all the SConscript. In the rare case you do not have moc generated files you can remove the sections that refer to moc generated files.

# Building moc content

MOC\_Generated = []

for moc\_file in moc\_files:

MOC\_Generated += env.MocBld(moc\_file)

# Contains all linked libraries:

linkedLibraries = commonLinkedLibraries

env.Prepend (LIBS = linkedLibraries)

# Creating object files

objFiles = env.SharedObject(sources + MOC\_Generated)

# Creating shared libraries

soFiles = env.SharedLibrary(

target = libName,

source = objFiles)

# Installing libraries

libInstall = env.Install(

dir = env['CXL\_lib\_dir'],

source = (soFiles))

Return('libInstall')

### SConstruct file

Adding a plugin to the SConstruct file is done in two parts:

Building the plugin itself.

newPlugin = []

plugin\_Obj = SConscript('Components/plugin/SConscript', variant\_dir=obj\_variant\_dir+'/plugin, duplicate=0)

CXL\_env.Depends(plugin\_Obj, OSWrappers\_Obj + BaseTools\_Obj)

newPlugin += plugin\_Obj

The plugin might have more than one library so we define an array to hold all the created libraries:

newPlugin = []

For each one we build it

plugin\_Obj = SConscript('Components/plugin/SConscript', variant\_dir=obj\_variant\_dir+'/plugin, duplicate=0)

Notice that here the duplicate=0 but in some cases where we depend on code that might need a local copy then the duplicate will need to be duplicate=1

Define its dependencies (in this example it is OSWrappers and BaseTools)

CXL\_env.Depends(plugin\_Obj, OSWrappers\_Obj + BaseTools\_Obj)

And then add it to the new plugin libraries

newPlugin += plugin\_Obj

Once it is build we need to add it to the main build so that the newPlugin will be build and also give it an alias in case we want only to build it and not whole CodeXL

In the end of the file we have:

CodeXL\_Full = \

GpuDebuggingPlugins + \

CpuProfilingPlugins + \

PowerProfilingPlugins + \

GpuProfilingPlugins + \

ThreadProfilingPlugins + \

AMDTSystemInformationHelper + \

AMDTTeaPot + \

AMDTClassicMatMul + \

KernelAnalyzer

So we add our newPlugin to the list

And we add an alias:

Alias( target='PluginName', source=( newPlugin))

## Implement Find?

## Open file

MDI event