# Class Dependency Analyzer

# Developer Guide

**V1.0**

## Table of Contents

# 1. Introduction

This guide is supposed to provide information for developers that intend to write their own extensions to enhance the Class Dependency Analyzer.
The motivation to do this can be

- adding own analysis ideas
- providing new graphical features to visualize dependency information
- exporting the data model to file or database

Chapter 2 of this document provides a general description of the extension mechanism supported by CDA and chapter 3 explains all extension points in detail.

# 2. Extension Mechanism

This chapter gives an overview of the extension mechanism supported by CDA and describes the prerequisites to consider when implementing a plug-in.

## *2.1. Prerequisites*

CDA as well as the library ODEM are implemented utilizing Java 5 language features (e.g. generics, enums, ...). Therefore it is required to use a JDK 1.5 to develop plug-in code.

Use **JDK 1.5** or higher for plug-in implementation.

For plug-in implementation the following JAR files from CDA must be in the compile classpath:

- **pf-odem.jar**
- **pf-cda-xpi.jar**
- **pf.jar** (optionally)

## *2.2. Implementation*

All extension-points are defined either as an *interface* or an *abstract class*. To provide a plug-in for a particular extension-point it is usually sufficient to just implement the interface or extend the abstract class and implement the required methods.
See the Javadoc provided with CDA for an accurate API description.

## *2.3. Plug-in Declaration*

To introduce a plug-in to CDA it must simply be declared in a special plug-in specification file (e.g. "***ui.plugins***") which must be located in the sub-folder **META-INF**.
The extension mechanism of CDA collects all such plug-in specification files on the classpath and automatically loads the declared plug-in classes declared in them. The file itself contains key/value pairs like properties files. The key is an arbitrary identifier for the particular plug-in implementation and the value is the full qualified class name. Optionally the class name can be prefixed by options which are enclosed by square brackets -> [*options*]. Currently only the following option is supported

**1** Defines that the plug-in should be loaded as singleton. That is, only one instance is created and re-used whenever the plug-in gets invoked.

Example 1:
```
   exportXML=[1]org.pf.tools.cda.ext.export.xml.ui.XmlFileModelExporterUIPlugin
```

If a plug-in should get some initialization parameters it is possible to specify those in the same line separated by semicolon (';'). Each key/value pair must be separated by a semicolon from the next setting.

Example 2:
```
   custom1=org.pf.tools.cda.ui.plugin.spi.ASubWindowPlugin;width=800;height=400
```

The plug-in class then must implement the interface `org.pf.plugin.InitializablePlugin` which requires to add *pf.jar* to the classpath. The parameters will then be given as `java.util.Properties` object to the

```
       public void initPlugin( String id, Properties properties )
```

method of the plug-in.

## 2.4. Deployment

In order to make a plug-in implementation available to CDA it must be packaged with all its classes and the metadata file into a JAR file.
That JAR has to be copied to the ***lib/ext*** sub-folder under $CDA_HOME.
With the next start of CDA the plug-in will automatically be found and loaded when required.

# 3. Extension Points

This chapter is a reference of all currently supported extension points. It describes which classes have to be extended and what implementation details have to be considered.

Since version 1.9 the following extension points are supported:

- `org.pf.tools.cda.ui.plugin.export.spi.AModelExporterUIPlugin`
  Allows to provide export capabilities for the data model shown in the tree view.

- `org.pf.tools.cda.ui.plugin.spi.ASubWindowPlugin`
  Allows to provide own functionality executed on a selected element in the tree view.

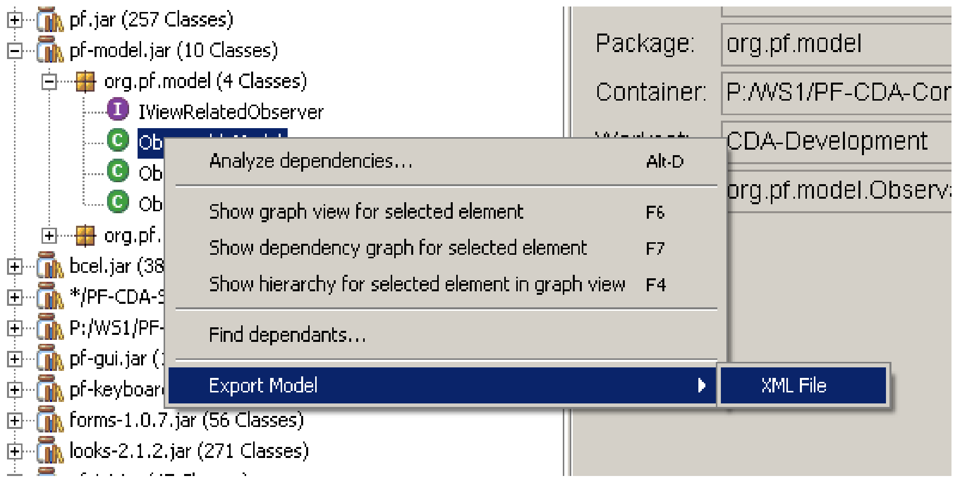
These extension points are explained in the following sub-sections.

## 3.1. org.pf.tools.cda.ui.plugin.export.spi.AModelExporterUIPlugin

The purpose of this extension-point is to allow custom export routines to be plugged-in. The export will be executed starting on a selected element (i.e. workset, container, package, class) in CDA's tree view.
Therefore the plug-in code will be added as a menu-item to the pop-up menus of all elements in the tree view. If the menu Item gets selected the export will be started on the current element.

This is a multi-plug-in extension-point, which means that multiple coexisting implementations are supported.

Below you can see a sample screenshot of the menu item for the XML file export plug-in.


pf.jar (257 Classes)
pf-model.jar (10 Classes)
org.pf.model (4 Classes)
IViewRelatedObserver
Analyze dependencies... Alt-D
Show graph view for selected element F6
Show dependency graph for selected element F7
Show hierarchy for selected element in graph view F4
Find dependants...
Export Model
XML File
bcel.jar
forms-1.0.7.jar (56 Classes)
looks-2.1.2.jar (271 Classes)
Package: org.pf.model
Container: P:/WS1/PF-CDA-Cor
CDA-Development
org.pf.model.Observ

For an implementation of this extension-point follow the steps below:

1. Create a subclass of
   `org.pf.tools.cda.ui.plugin.export.spi.AModelExporterUIPlugin`
   and implement the methods `getPluginProvider(), getPluginVersion()` of interface
   `org.pf.tools.cda.xpi.IPluginInfo`
2. Implement method
   `public String getActionText( Locale locale, IExplorationModelObject object )`
   It must return a non-null value for all given objects for that it supports an export. That means, whenever this method returns null, no menu item is shown in the selected element's pop-up menu for this plug-in.
3. Create a subclass of
   `org.pf.tools.cda.plugin.export.spi.AModelExporter`
   and override the `startXXX()` and `finishXXX()` methods as appropriate.
4. In your subclass of `AModelExporterUIPlugin` (from step 1) implement abstract method
   `public AModelExporter createExporter( PluginConfiguration config )`
   which basically should return a new instance of your `AModelExporter` subclass (from step 3).

That's it.
Don't forget to add the plug-in declaration to the ***ui.plugins*** file in the META-INF sub-directory of your plug-in JAR file. It is recommended to define the [1] option  to specify that the plug-in is treated as a singleton.

A sample plug-in bundling for this extension-point can be found in the ***pf-cda-ext.jar*** which is part of CDA.

## Summary:

| **org.pf.tools.cda.ui.plugin.export.spi.AModelExporterUIPlugin** | |
|---|---|
| UI plug-in | yes |
| Multi plug-in extension-point | yes |
| Singleton | yes |

## *3.2. org.pf.tools.cda.ui.plugin.spi.ASubWindowPlugin*

This is a very general extension-point. It allows to plug-in any custom code to work on the currently selected element in the tree view. Each plug-in provided for this extension-point will create an additional menu-item in the pop-up menus of the elements in the tree view. For which type of element the menu-item will be shown is controlled by the plug-in's method

```
public String getActionText( Locale locale, IExplorationModelObject object )
```

from interface `IPluginActionInfo`. If the method returns a string this string will be used as the menu-item's label. If the method returns null, no menu-item will be shown. All other methods from interface `IPluginActionInfo` have a default implementation in the plug-in's superclasses.

If the menu-item gets selected a new window will be opened. Therefore the plug-in must provide a `java.awt.Frame`. The contents of that frame is completely the responsibility of the plug-in.
The CDA plug-in framework takes care that the new window is handled like all other windows of the application. For closing by a button or menu-item just call method `close()` which is implemented in the superclass.

So creating a plug-in for this extension-point is very simple:

1. Create a subclass of
   `org.pf.tools.cda.ui.plugin.spi.ASubWindowPlugin`
   and implement the methods `getPluginProvider()`, `getPluginVersion()` of interface
   `org.pf.tools.cda.xpi.IPluginInfo`
2. Implement method
   `public String getActionText( Locale locale, IExplorationModelObject object )`
   It must return a non-null value for all given objects for that it supports an export. That means, whenever this method returns null, no menu item is shown in the selected element's pop-up menu for this plug-in.
3. Implement method
   `public Frame createFrame( IExplorationModelObject object )`
   This method is required to create the frame to be shown as separate new window.

Finally bundle all necessary classes together with your META-INF/ui.plugins into a JAR file.

### Summary:

| **`org.pf.tools.cda.ui.plugin.spi.ASubWindowPlugin`** | |
|---|---|
| UI plug-in | yes |
| Multi plug-in extension-point | yes |
| Singleton | no |

### Example:

There is an example plug-in for this extension-point coming with CDA. It is in the sub-folder *samples*. To activate the sample, just copy ***plugin-sample1.jar*** to the {CDA_HOME}/lib/ext folder and re-start CDA.