Class Dependency Analyzer 2.1.0

CDA Developer Guide

Version **1.4**

MDCS

Manfred Duchrow Consulting & Software

Author: Manfred Duchrow

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 or those who just want to use the core functionality of CDA – without GUI – in their own projects.

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 available extension points in detail. In chapter 4 the focus is on the CDA core functionality and the relevant API to utilize it in other contexts.

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.

1.1. Prerequisites

The current CDA version (2.1.0 or higher) is built with Java 8 and requires a Java 8 JVM to execute.

Use **JDK 1.8** or higher for plug-in implementation.

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

- **pf-odem-1.2.0.jar**
- **pf-cda-base-xpi-2.1.0.jar**
- **pf-cda-swing-xpi-2.1.0.jar**
- **pf-base-7.1.0.jar** (optionally)

1.2. 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.

1.3. Plug-in Declaration

To introduce a plug-in to CDA it must 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 (unique) 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.pfsw.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.pfsw.tools.cda.ui.plugin.spi.ASubWindowPlugin;width=800;height=400
```

To receive such initialization parameters, the plug-in class must implement the interface `org.pfsw.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.

1.4. Deployment

In order to make a plug-in implementation available to CDA it must be packaged with all its classes and the metadata *META-INF/ui.plugins* 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 be found automatically 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 CDA version 1.9 the following extension points are supported:

- `org.pfsw.tools.cda.ui.plugin.export.spi.AModelExporterUIPlugin`
  Allows to provide export capabilities for the data model shown in the tree view.
- `org.pfsw.tools.cda.ui.plugin.spi.ASubWindowPlugin`
  Allows to provide own functionality executed on a selected element in the tree view.

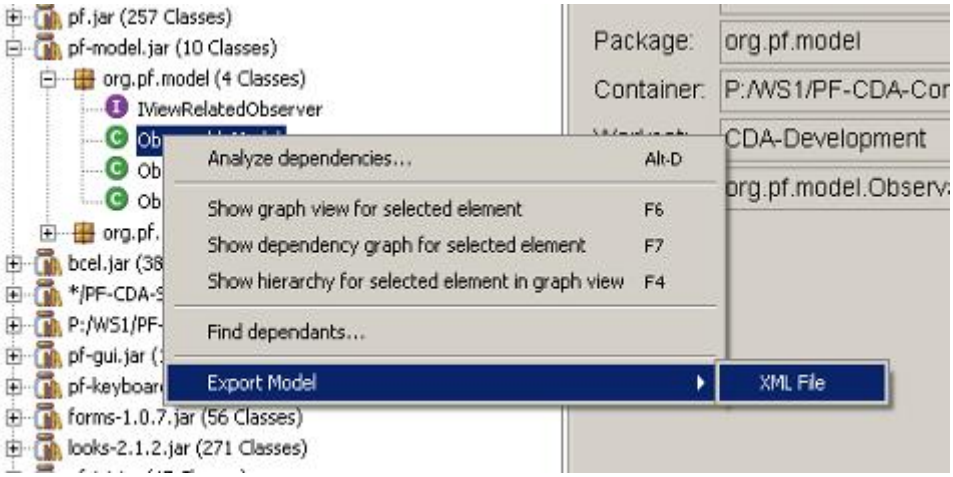
These extension points are further explained in the following sub-sections.

1.5. org.pfsw.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.



1. Create a subclass of
   `org.pfsw.tools.cda.ui.plugin.export.spi.AModelExporterUIPlugin`
2. Optionally implement
   interface `org.pfsw.tools.cda.xpi.IPluginInfo` with the methods `getPluginProvider()`, `getPluginVersion()`
3. 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.
4. Create a subclass of
   `org.pfsw.tools.cda.plugin.export.spi.AModelExporter`
   and override the `startXXX()` and `finishXXX()` methods as appropriate.
5. 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 4).

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-extensions.jar*** which is part of CDA.

Summary:

`org.pfsw.tools.cda.ui.plugin.export.spi.AModelExporterUIPlugin`	
UI plug-in	yes
Multi plug-in extension-point	yes
Singleton	yes

1.6. org.pfsw.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.pfsw.tools.cda.ui.plugin.spi.ASubWindowPlugin`
   and implement the methods `getPluginProvider()`, `getPluginVersion()` of interface
   `org.pfsw.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.pfsw.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.

4 Programmatic Analysis

Sometimes people do not want to use the GUI of CDA. Instead they are interested in utilizing the core functionality of analyzing class dependencies in their own software. Therefore this section provides information about the CDA domain model and the basic knowledge of how to use it in your own software.
For details please refer to the API Javadoc, bundled with the CDA download.

Be aware that there is no guarantee that the API stays stable or backward compatible in any way in future releases of CDA!

4.1 Required Libraries

To do programmatic analyzing put the JARs from lib folder in the distributable

```
pf-cda-2.1.0.zip
```

into the compile classpath or use a dependency manager (see below).

4.1.1 Maven Repository

With build tools like *Apache Maven* or *Gradle* you can simply include the following Maven repository URL:

```
http://pfsw.org/maven/
```

and adding the dependency.

Maven:

```
<dependency>
    <groupId>org.pfsw</groupId>
    <artifactId>pf-cda-core</artifactId>
    <version>2.1.0</version>
<dependency>
```

Gradle:

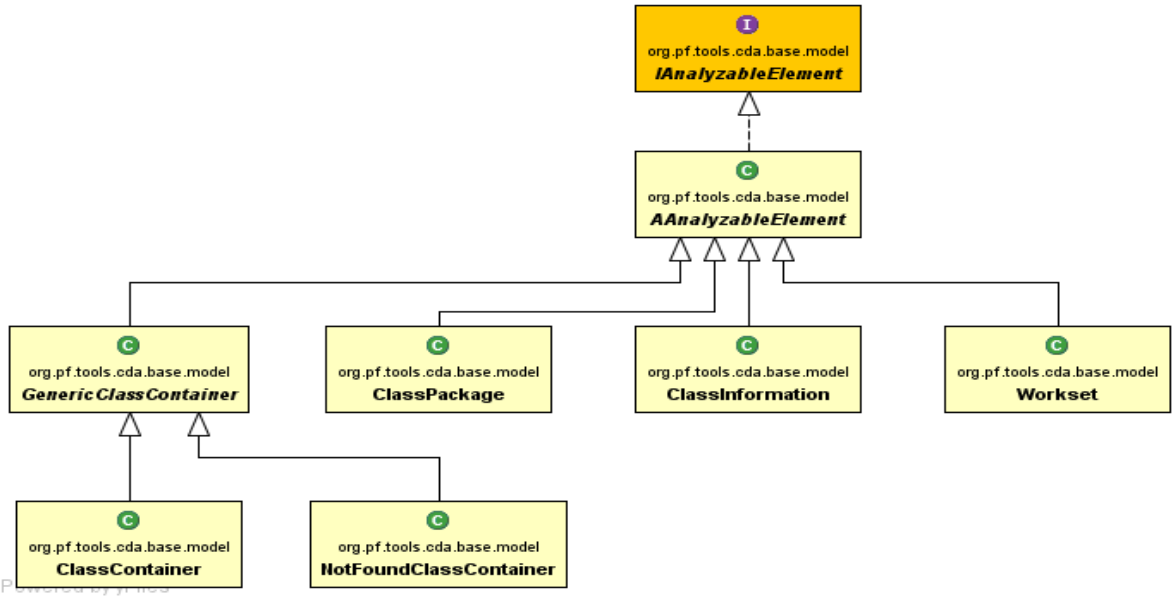
```
compile group: 'org.pfsw', name: 'pf-cda-core', version: '2.1.0'
```

4.2 Domain Model

The model classes of CDA are in **pf-cda-base-2.1.0.jar** in package `org.pfsw.tools.cda.model.base`.
The most important model classes are:

- Workset
- ClassContainer
- ClassPackage
- ClassInformation

They are all subclasses of the abstract class *AnalyzableElement*. See the class diagram below for an overview. A short description for these model classes can be found in the next sub-sections.



4.2.1 Workset

The purpose of a *Workset* is mainly to define the set of class files to be analyzed (similar to a classpath). It supports various types for specifying such files:

- Folder
  All files matching pattern "*.class" in that folder and sub-folders will be included
- Java Archive (**JAR**)
  All files matching pattern "*.class" in such a zipped archive will be included
- Java Web Application Archive (**WAR**)
  All files matching pattern **"WEB-INF/classes/*.class"** in such a zipped archive will be included. Furthermore the contained JAR files under **"WEB-INF/lib"** will also be extracted and loaded automatically.
- OSGi Bundle (**JAR with specific MANIFEST.MF**)
  The manifest attribute "**Bundle-Classpath**" will be evaluated. All JAR files listed there and that are existing inside the bundle will also be extracted and loaded automatically.
- Eclipse .classpath File
  All classes defined by this Eclipse specific classpath definition will be included.
  Attention: Since the format has changed with different Eclipse version this mechanism might not always work properly!

4.2.2 GenericClassContainer / ClassContainer

As the name already indicates, these objects contain the classes. Actually they are not directly containing the classes, but the packages in which then the classes are. This model classes provide methods to find contained classes or packages. The container can also simply be asked to which other containers it has dependencies:

```
GenericClassContainer[] containers = container.getDirectReferredContainers();
```

4.2.3 ClassPackage

Each loaded package is represented by an instance of this class. The separation of packages and

classes allows dependency analysis on different levels. Most other tools can only provide information on class level. With CDA it is possible for example to get all classes a package depends on. The packages a package depends on can be determined by

```
ClassPackage[] packages = pack.getDirectReferredPackages();
```

4.2.4 ClassInformation

All interfaces, classes, enums and annotations are held as instances of *ClassInformation*. The rich API provides a lot of information about the types themselves and about their dependencies.
Some useful methods (see Javadoc for more):

```
public boolean isInterface()
public boolean isEnum()
public boolean isAnnotation()
public boolean isClass()
public boolean isDirectExtensionOf(ClassInformation classInfo)
public boolean isSubclassOf(ClassInformation otherClass)
public ClassInformation[] getReferredClassesArray()
public ClassFileVersion getClassFileVersion()
public boolean refersTo(IAnalyzableElement analyzableElement)
```

4.3 Using CDA Base and Core classes

The general way to work with CDA can be separated into the following phases:

1. Create a *Workset* object initialized with the desired classpath definition.
2. Run the initialization, scanning and pre-analyzation on the *Workset*.
3. Execute a *Processor* over all or a subset of containers, packages or classes that collects elements that match specific criteria.
4. Release resources and clean up temporary files and folders.

4.3.1 Phase 1 – Workset Creation

A *Workset* instance can be created with a few lines of code. Have a look at the following example:

```
Workset workset = new Workset("Sample1");
ClasspathPartDefinition partDefinition = new ClasspathPartDefinition("example-libs/*.jar");
workset.addClasspathPartDefinition(partDefinition);
return workset;
```

You can add as many `ClasspathPartDefinition` objects as you like.

An alternative to create a *Workset* is using the

```
org.pfsw.tools.cda.core.io.workset.WorksetReader
```

It can read workset files (*.ws) created via the (GUI based) CDA tool or with the class

```
org.pfsw.tools.cda.core.io.workset.WorksetWriter
```

Example:

```
workset = WorksetReader.readWorkset(”my-worksets/sample1.ws”);
```

4.3.2 Phase 2 – Workset Initialization

The easiest way to load and analyze the classes specified by a *Workset* is to use the

```
org.pfsw.tools.cda.core.init.WorksetInitializer
```

Example:

```
WorksetInitializer wsInitializer = new WorksetInitializer(workset);
wsInitializer.initializeWorksetAndWait(null);
```

Instead of null an implementor of interface `org.pfsw.tools.cda.core.processing.IProgressMonitor` can be defined to follow the (sometimes lengthy) process of loading and analyzing all classes.

Eventually, after the initialization all classes and their resolved dependencies are in-memory (so ensure to provide enough heap space).

4.3.3 Phase 3 - Analyzing

The last phase is about getting the specific information of interest from the loaded and resolved classes.
Getting a single class is as simple as

```
ClassInformation classInfo = workset.getClassInfo("org.pfsw.tools.cda.base.Module");
```

To get the classes (interfaces, enums and annotations) a class depends on just use

```
ClassInformation[] classes = classInfo.getReferredClassesArray();
```

It is also possible to get all packages the class depends on:

```
ClassPackage[] packages = classInfo.getDirectReferredPackages()
```

More complex or special analysis tasks will be executed with processor objects and corresponding methods in the model classes that follow the *visitor* design pattern.
The general approach is the execution of one of the following methods:

- `processClassContainersObjects(Collection, IClassContainerProcessor)`
- `processClassPackageObjects(Collection, IClassPackageProcessor)`
- `processClassInformationObjects(Collection, IClassInformationProcessor)`

Each is for a different type of domain model class and each requires a type specific processor as parameter. The processor is an object that will be called for each element of the relevant type.

For ready to use implementations of the processor interfaces have a look into the packages

```
org.pfsw.tools.cda.base.model.processing
org.pfsw.tools.cda.core.dependency.analyzer
```

All those processor interfaces are derived from

```
org.pfsw.tools.cda.base.model.processing.IAnalyzableElementProcessor<TAnalyzableElement>
```

So they all can be used in two ways

1. As filter – that is the matches() method of the processor decides whether or not an element is relevant for the current processing.
2. As actor on the elements – that is, each element can be manipulated, collected, counted or whatever necessary.

All the `processClass`***xxx***`Objects(collection, processor)` methods are working in the following way:

1. The process is used for each element of the relevant type (as part of the method name).
2. First the `process(element)` method of the processor gets called.

3. It can do whatever it likes with the given element.
4. If it returns *false* the processing ends immediately – no further elements are processed.
5. If it returns *true* and a collection object (not null) was provided then the `matches(element)` method gets called.
6. This method should not do anything with the given element. It simply should decide whether or not the element matches the criteria of the processor.
7. If the element matches, the method returns true and the element will be added to the collection.

One example for a class processor is

`org.pfsw.tools.cda.core.dependency.analyzer.MainClassDetector<TResultData>`

Its purpose is finding all classes that implement a static main() method. Its `process()` method just always returns *true*. Its `matches()` method only returns true for classes that have a main() method.

4.3.4 Phase 4 – Clean Up

After all work is done, it is strongly recommended to clean up everything by calling

`workset.release()`

That is particularly important if EAR file, WAR files or OSGi bundles have been declared in the workset's classpath definition. In order to analyze such files they are extracted by CDA to temporary folders and files. The above `release()` call will remove all that temporary stuff and frees all other internally used resources.
After a workset object has been released it **must not** be used anymore!

4.3.5 Example Code

Along with CDA there is a zip file that contains an Eclipse project with example code for some variants of programmatic CDA usage. Have a look at the download page.

4.3.5.1 Detect Dependency Cycles

To detect cyclic dependencies inside a workset, you can use the

`org.pfsw.tools.cda.core.dependency.analyzer.CircularDependenciesDetector`

After creating (see 4.3.1) and initializing (see 4.3.2) a workset you can use code similar to that in the example below.

```
CircularDependenciesDetector detector = new CircularDependenciesDetector(workset);
detector.setIgnoreCyclesInSameContainer(false);
detector.setIgnoreCyclesInSamePackageBranch(false);
detector.setIgnoreCyclesInSamePackage(false);
detector.analyze();
CircularDependenciesResult result = detector.getResult();

StringBuilder buffer;
for (ClassInformation[] classInformations : result.getDependencyPaths())
{
  buffer = new StringBuilder();
  for (ClassInformation classInformation : classInformations)
  {
    if (buffer.length() > 0)
    {
      buffer.append(" --> ");
    }
    buffer.append(classInformation.getClassName());
  }
  System.out.println(buffer);
}
```