Codimension Plugins Tutorial

In this tutorial two major topics are discussed. The first part discusses how plugin support is implemented in Codimension. The second part discusses implementation of a simple plugin and what is available for plugins.

Plugin Support Implementation

The shortest answer on the question "what is a Codimension plugin?" is as follows: a Codimension plugin is a Python class implemented in a certain way. Codimension is written in Python and thus its plugins should also be written in Python.

Before some implementation details come up however it makes sense to discuss a few terms Codimension uses while it works with plugins.

At the start time Codimension looks for plugins in two places. The first one is `/usr/share/codimension-plugins/`. The second one is the directory called `.codimension/plugins` located in the user home directory. So on Linux system the latter most probably will be `~/.codimension/plugins`. It is highly recommended that each plugin occupies a designated directory where it keeps all the required files. So on a certain system the related directories structure may look as follows:

```
/usr/share/codimension-plugins/plugin1/...
                               plugin2/...
                               plugin3/...

/home/mike/.codimension/plugins/plugin4/...
                                plugin5/...
                                plugin6/...
```

Depending on a plugin location Codimension splits all the found plugins into two groups: system wide plugins and user plugins. So in the example above `plugin1`, `plugin2` and `plugin3` are system wide plugins while `plugin4`, `plugin5` and `plugin6` are user plugins.

The next pieces which are important for Codimension are a plugin name and a plugin version. A name and a version are stored in a plugin description file (it will be discussed later). That description file is what triggers loading a plugin.

The next important piece is that each plugin can be enabled or disabled (a pair of other terms activated/deactivated is also used further with the same meaning). While loading plugins Codimension initially treats all plugins enabled so a newly installed plugin will be automatically enabled next time Codimension starts. It is possible however that a plugin conflicts with another plugin. Certain types of conflicts can be detected by Codimension automatically and Codimension can disable some plugins to resolve a conflict. The following rules are used for automatic conflict resolution:

- If there is a user and a system wide plugin with the same name then the user plugin wins.
- If there are two plugins with the same name and both of them are either user or system wide then their version is taken into consideration. The higher version wins.
- If names, vaersions and locations of two plugins match then an arbitrary one wins.

There are a few other cases when Codimension disables a plugin automatically. A good example of such a case is when a plugin does not implement the required interface.

An important detail here could be that regardless whether a plugin is enabled or disabled it is instantiated. The plugin class instance will stay in memory till Codimension is closed.

The user is always able to enable or disable plugins manually and in particular resolve detected conflicts the required way if automatic resolution is not what is needed. The manual control of plugin states is done in the plugin manager. The manager user interface is available via main menu **Options->Plugin Manager** menu item as shown below.

Screenshot with the plugin manager

Each plugin can move between the enabled and disabled state an arbitrary number of times within a single Codimension session. This could be illustrated as follows.

Diagram with IDE and plugin states

The last term Codimension intriduces for plugins is a plugin category. Plugins could require different support on the IDE side and a plugin category is the way how to distinguish the required support. For example, a spell checker plugin might need certain support targeted to text editing while a plugin which implements a regular expression visual testing facility does not need text editing support at all. A plugin category defines an interface variation between Codimension and a plugin. The categories come in a form of predefined base classes and each plugin must derive from one of them.

Plugin Files

As it was mentioned above it is highly recommended that a plugin occupies a designated directory. For example, directory content for a plugin may look as follows.

```
/home/mike/.codimension/plugins/pdfexporter/pdfexporter.cdmp
                                            __init__.py
                                            util_functions.py
                                            config_dialog.py
```

The `pdfexporter.cdmp` file contains textual plugin description. The name of the file does not matter, Codimension looks for the .cdmp file extensions. A content of pdfexporter.cdmp file may be similar to the following.

```
[Core]
Name = PDF exporter
Module = .

[Documentation]
Author = Mike Slartibartfast <mike.slartibartfast@some.com>
Version = 1.0.0
Website = http://mike.slartibartfast.homelinux.com/pdfexporter
Description = Codimension PDF exporter plugin
License = GPL v.3
```

The `[Core].Name` value is an arbitrary string however it is better to keep it relatively sort. The `[Core].Module` value is a directory path where Codimension plugin resides. It is recommended that all the plugin files are sitting in a designated directory including the plugin description file and therefore the `[Core].Module` value refers to the very directory it is sitting in. The '.' value is the recommended value for all Codimension plugins.

The `[Documentation]` section has self explanatory values. A plugin can add any values to this section and all of them will be displayed in the **Detailed information** box in the plugin manager dialog when a plugin is selected.

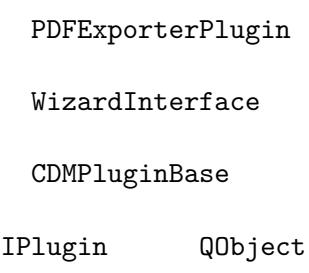
The `__init__.py` file is the one where a plugin class definition must reside. In the example above the plugin also has some utility functions in the `util_functions.py` and a configuration dialog in a separate files. The import these modules from `__init__.py` there is no need in relative import. The `__init__.py` can simply use:

```
# The plugin modules do not require relative import
import config_dialog
from util_functions import designCoastline
```

The Codimension modules are also available for the plugin code. So a plugin can use statements similar to the following:

```
# Importing pixmaps cache from a Codimension module
from utils.pixmapcache import PixmapCache
codimensionLogo = PixmapCache().getPixmap( 'logo.png' )
```

It was mentioned in the previous section that a plugin class must derive from one of the predefined plugin category base class. So a part of the PDF exporter plugin class hierarchy may look as follows:



The `PDFExporterPlugin` class must reside in the `__init__.py` file. This is the class which implements the plugin interface. The plugin developer does not need and should not make any changes in any other classes mentioned on the diagram.

The `WizardInterface` class is a Codimension provided plugin category base class. The class is defined in `codimension/src/plugins/c` The class has a set of member functions some of which have to be implemented by the plugin of this category. The member function documentation strings describe in details what is expected by Codimension. At the time of writing (Codimension v.2.1.0) the `WizardInterface` is the only supported plugin category. When a new plugin category is introduced its base class will appear in the `codimension/src/plugins/categories/` directory. The next anticipated plugin category will serve version control systems.

The `CDMPluginBase` class is a Codimension provided convenience class which simplifies access to the major IDE objects. The class definition resides in the `codimension/src/plugins/categories/cdmpluginbase.py` file. Having `CDMPluginBase` class in the hierarchy makes it possible for a plugin class to use simple to read statements similar to the following:

```
if self.ide.project.isLoaded():
    # The ide has a project loaded
    ...
else:
    # There is no project, the user edits individual files
    ...
```

Access to all the IDE objects should start with:

```
self.ide. ...
```

See the `IDEAccess` class in the `codimension/src/plugins/categories/cdmpluginbase.py` file for a full list of provided IDE objects.

Codimension uses thirdparty library called yapsy to build plugin support on top of it. Yapsy needs to have IPlugin in the plugin class hierarchy and so it is here. A plugin developer should not need to deal with this class however.

The `QObject` class is a PyQt provided class. The class is included into the hierarchy for convenience. Codimension uses QT library for the user interface and therefore QT signals are used quite often. Having `QObject` in the base gives a convenient way to subscribe for signals and to emit them, e.g. a plugin may have the following code:

```
self.connect( self.ide.project, SIGNAL( 'projectChanged' ),
              self.__onProjectChanged )
```

Plugin Example: Garbage Collector Plugin

The idea of an example plugin is quite simple. The Python garbage collector triggers objects collection at pretty much unknown moments and the plugin will make it more predictable. The garbage collector plugin (GC plugin) will call the `collect()` method of the gc Python module when: * a tab is closed * a project is changed * new files appeared in a project * some files are deleted from a project

The `gc.collect()` call provides an information of how many objects were collected as its return value and this could be interesting to see. So a message should be shown somewhere. To make it more user friendly the GC plugin should provide a configuration dialog with options where to show the message: * in the log tab * on the status bar * do not show anything The selected option should be memorized and restored the next time Codimension starts.

Having the requirements at hand let's start implementing the GC plugin with creating a directory where all the plugin files will be located.

```
> mkdir garbagecollector
> cd garbagecollector
```

First, we need the plugin description file, let's call it `garbagecollector.cdmp`. The content of the file will be as follows:

```
[Core]
Name = Garbage collector
Module = .

[Documentation]
Author = Sergey Satskiy <sergey.satskiy@gmail.com>
Version = 1.0.0
Website = http://satsky.spb.ru/codimension
Description = Codimension garbage collector plugin
License = GPL v.3
```

The GC plugin will belong to the wizard plugin category so it must derive from the `WizardInterface` class. The definition of the class must be in the `__init__.py` file.

Miscellaneous

Printing and Logging

Plugins are running in Codimension context so everything what is done in Codimension for the IDE is applicable to plugins. In particular Codimension intercepts printing to stdout and to stderr. If a plugin prints on stdout:

```
print "Hi from plugin"
```

then the message will appear in the log tab in black. If a plugin prints on **stderr**:

```
print >> sys.stderr, "ATTENTION"
```

then the message will appear in the log tab in red.

Codimension also defines a logging handler so that the messages will be redirected to the log tab, for example:

```
import logging
logging.info( "Message" )
```

will lead to a message in the log tab. Codimension can be started with `--debug` option and in this case debug log level will be switched on, otherwise debug log messages are suppressed. E.g.

```
import logging
logging.error( "Error message" )    # Will be shown regardless of the startup options
logging.debug( "Debug message" )    # Will be shown only if Codimension started as:
                                    # >codimension --debug
```

Globals and Settings

When a plugin is activated references to the IDE global data singleton and to the IDE settings singleton are passed to the plugin. Using these sinletons a plugin can get access to pretty much everything in the IDE. It is also possible to cause Codimension crash if important data are misproperly modified.

The `CDMPluginBase` class provides syntactic shugar to simplify access to the most important IDE objects. The other IDE objects could be accessible using direct access to the globals and settings members. If you feel more syntactic shugar should be added to `CDMPluginBase` (or something is not accessible) please feel free to contact Sergey Satskiy at sergey.satskiy@gmail.com.