The eric6 plug-in system

Version 17.12

Table of contents

List of figures

List of listings

1 Introduction

eric 4.1 introduced a plug-in system, which allows easy extension of the IDE. Every user can customize the application by installing plug-ins available via the Internet. This document describes this plug-in system from a user perspective and from a plug-in developers perspective as well.

2 Description of the plug-in system

The eric6 plug-in system is the extensible part of the eric6 IDE. There are two kinds of plug-ins. The first kind of plug-ins are automatically activated at startup, the other kind are activated on demand. The activation of the on-demand plug-ins is controlled by configuration options. Internally, all plug-ins are managed by the PluginManager object. Deactivated autoactivate plug-ins are remembered and will not be activated automatically on the next start of eric6.

Eric6 comes with quite a number of core plug-ins. These are part of the eric6 installation. In addition to this, there are additional plug-ins available via the internet. Those plug-ins may be installed and uninstalled using the provided menu or toolbar entries. Installable plug-ins live in one of two areas. One is the global plug-in area, the other is the user plug-in area. The later one overrides the global area.

3 The plug-in system from a user perspective

The eric6 plug-in system provides the user with a Plug-ins menu in the main menu bar and a corresponding toolbar. Through both of them the user is presented with actions to show information about loaded plug-ins and to install or uninstall plug-ins.

3.1 The Plug-ins menu and toolbar

The plug-ins menu is located under the “Plugins” label in the main menu bar of the eric6 main window. It contains all available user actions and is accompanied by a toolbar containing the same actions. They are shown in the following figures.

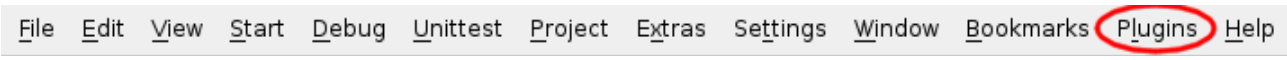

Figure 1: eric6 main menu

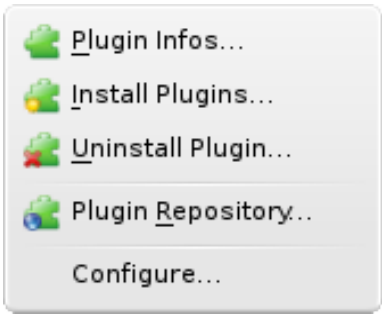

Figure 2: The Plug-ins menu

Figure 3: The Plug-ins toolbar

The “Plugin Infos...” action is used to show a dialog, that lists all the loaded plug-ins and there status. The entry labeled “Install Plugins...” opens a wizard like dialog to install new plug-ins from plug-in archives. The entry, “Uninstall Plugin...”, presents a dialog to uninstall a plug-in. If a plug-in to be uninstalled is loaded, it is unloaded first. The entry called “Plugin Repository...” shows a dialog, that displays the official plug-ins available in the eric6 plug-in repository. The “Configure...” entry opens the eric6 configuration dialog displaying the Plugin Manager configuration page.

3.2 The Plug-in Infos dialog

The “Plugin Infos” dialog shows information about all loaded plug-ins. Plug-ins, which had a problem when loaded or activated are highlighted. More details are presented, by double clicking an entry or selecting the “Show details” context menu entry. An example of the dialog is show in the following figure.

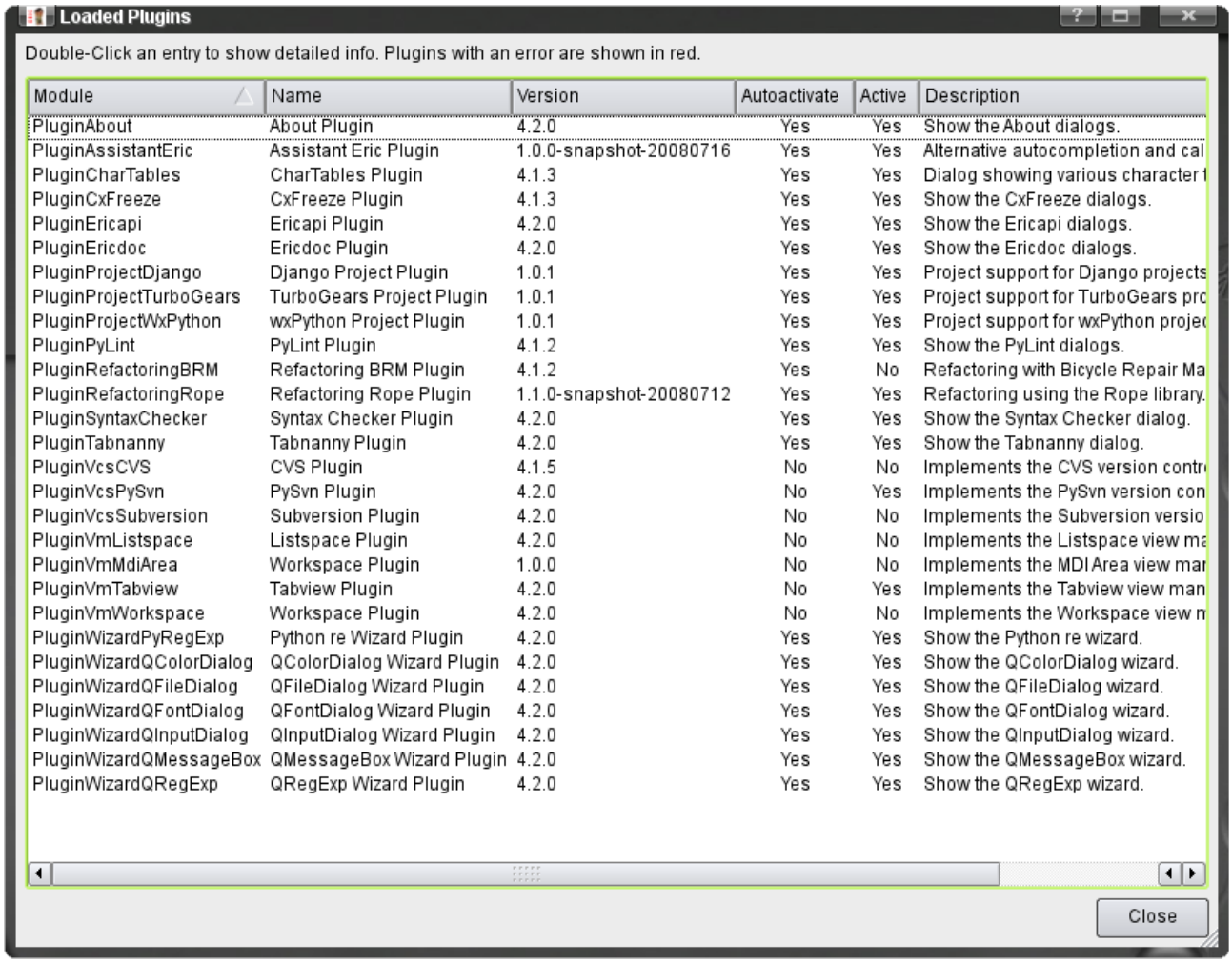

Figure 4: Plug-ins Info dialog

The columns show information as follows.

- Module
  This shows the Python module name of the plug-in. It is usually the name of the plug-in file without the file extension. The module name must be unique.
- Name
  This is the name of the plug-in as given by the plug-in author.
- Version
  This shows the version of the plug-in.
- Autoactivate
  This indicates, if the plug-in should be activated at startup of the eric6 IDE. The actual activation of a plug-in is controlled by the state it had at the last shutdown of eric6.
- Active
  This gives an indication, if the plug-in is active.
- Description
  This column show a descriptive text as given by the plug-in author.

This dialog has a context menu, which has entries to show more details about a selected plug-in and to activate or deactivate an autoactivate plug-in. It is shown below.

Figure 5: Plug-ins Info dialog context menu

Deactivated plug-ins are remembered and will not be activated automatically at the next startup of eric6. In order to reactivate them, the “Activate” entry of the context menu must be selected.

Selecting the “Show details” entry opens another dialog with more information about the selected plug-in. An example is shown in the following figure.

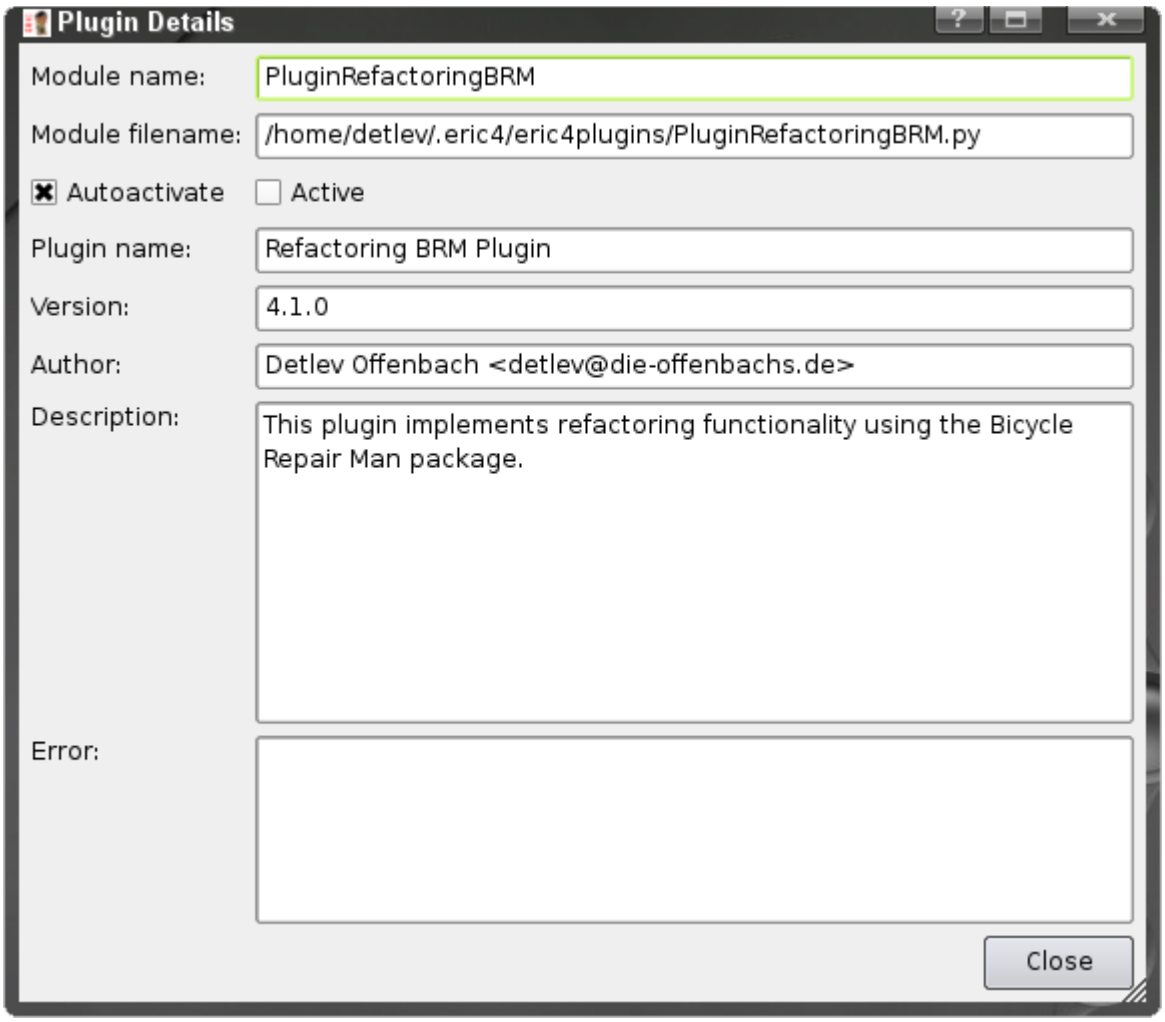

Figure 6: Plug-in Details dialog

The entries of the dialog are as follows.

- Module name:
  This shows the Python module name of the plug-in. It is usually the name of the plug-in file without the file extension. The module name must be unique.
- Module filename:
  This shows the complete path to the installed plug-in Python file.
- Autoactivate
  This indicates, if the plug-in should be activated at startup of the eric6 IDE. The actual activation of a plug-in is controlled by the state it had at the last shutdown of eric6.
- Active
  This gives an indication, if the plug-in is active.
- Plugin name:
  This is the name of the plug-in as given by the plug-in author.
- Version:
  This shows the version number of the installed plug-in. This number should be

passed to the plug-in author when reporting a problem.

- Author:
  This field gives the author information as provided by the plug-in author. It should contain the authors name and email.
- Description:
  This shows some explanatory text as provided by the plug-in author. Usually this is more detailed than the short description displayed in the plug-in infos dialog.
- Error:
  In case a plug-in hit an error condition upon loading or activation, an error text is stored by the plug-in and show in this field. It should give a clear indication about the problem.

3.3 Installing Plug-ins

New plug-ins are installed from within eric6 using the Plug-in Installation dialog. It is show, when the “Install Plugin...” menu entry is selected. Please note, that this is also available as a standalone tool using the `eric6_plugininstall.py` script or via the eric6 tray menu. The user is guided through the installation process by a wizard like dialog. On the first page, the plug-in archives are selected. eric6 plug-ins are distributed as ZIP-archives, which contain all installable files. The “Add ...”-button opens a standard file selection dialog. Selected archives may be removed from the list with the “Remove”-Button. Pressing the “Next >” button continues to the second screen.

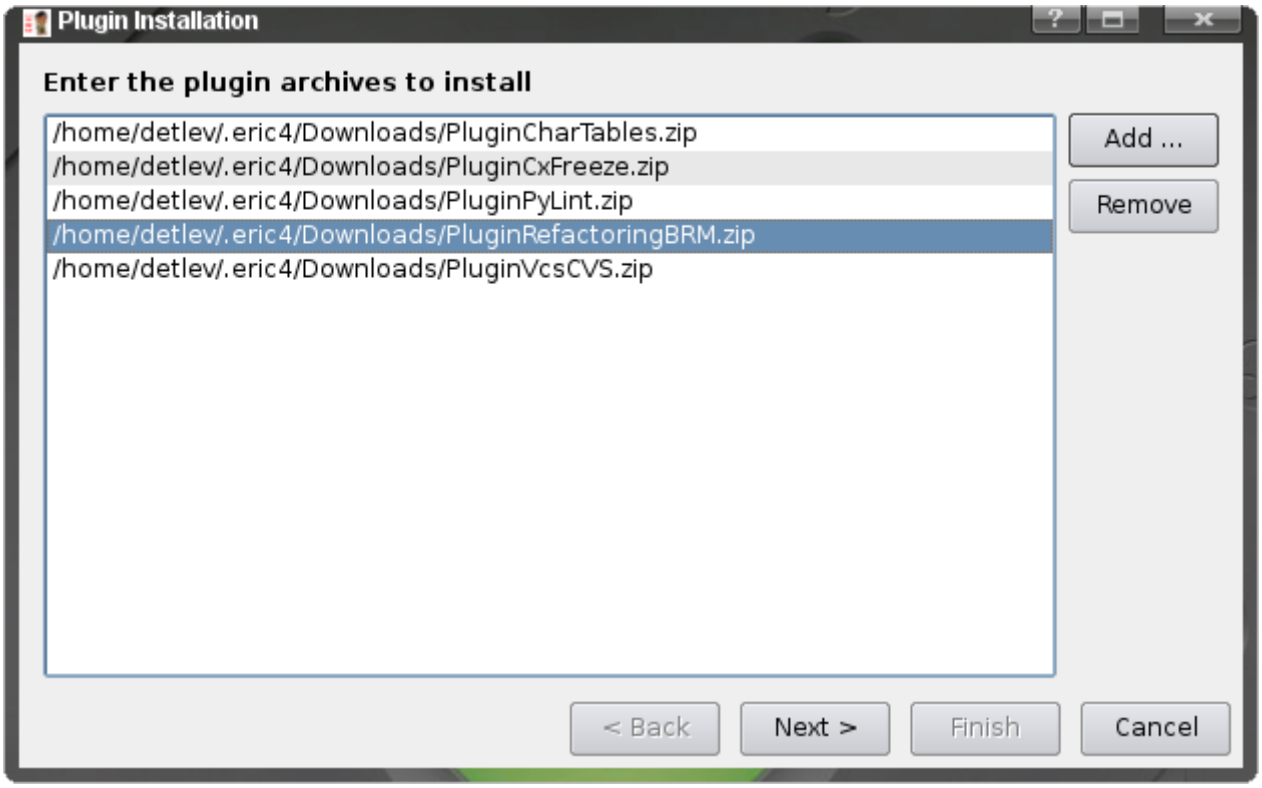

Figure 7: Plug-ins Installation dialog, step 1

The second display of the dialog is used to select the directory, the plug-in should be installed into. If the user has write access to the global eric6 plug-ins directory, both the

global and the user plug-ins directory are presented. Otherwise just the user plug-ins directory is given as a choice. With the “< Back” button, the user may go back one screen. Pressing “Next >” moves to the final display.

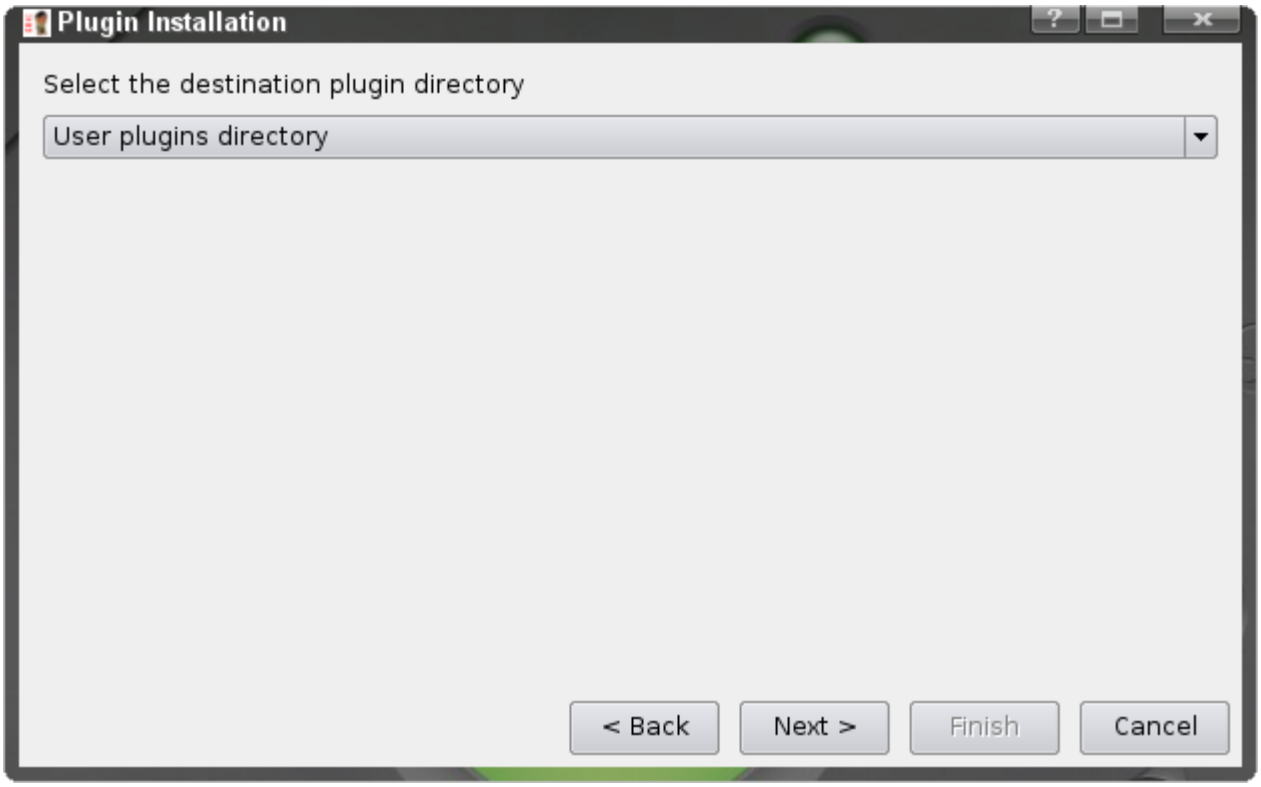

Figure 8: Plug-ins Installation dialog, step 2

The final display of the plug-in installation dialog shows a summary of the installation data entered previously. Again, the “< Back” button lets the user go back one screen. The “Finish” button is used to acknowledge the data and starts the installation process.

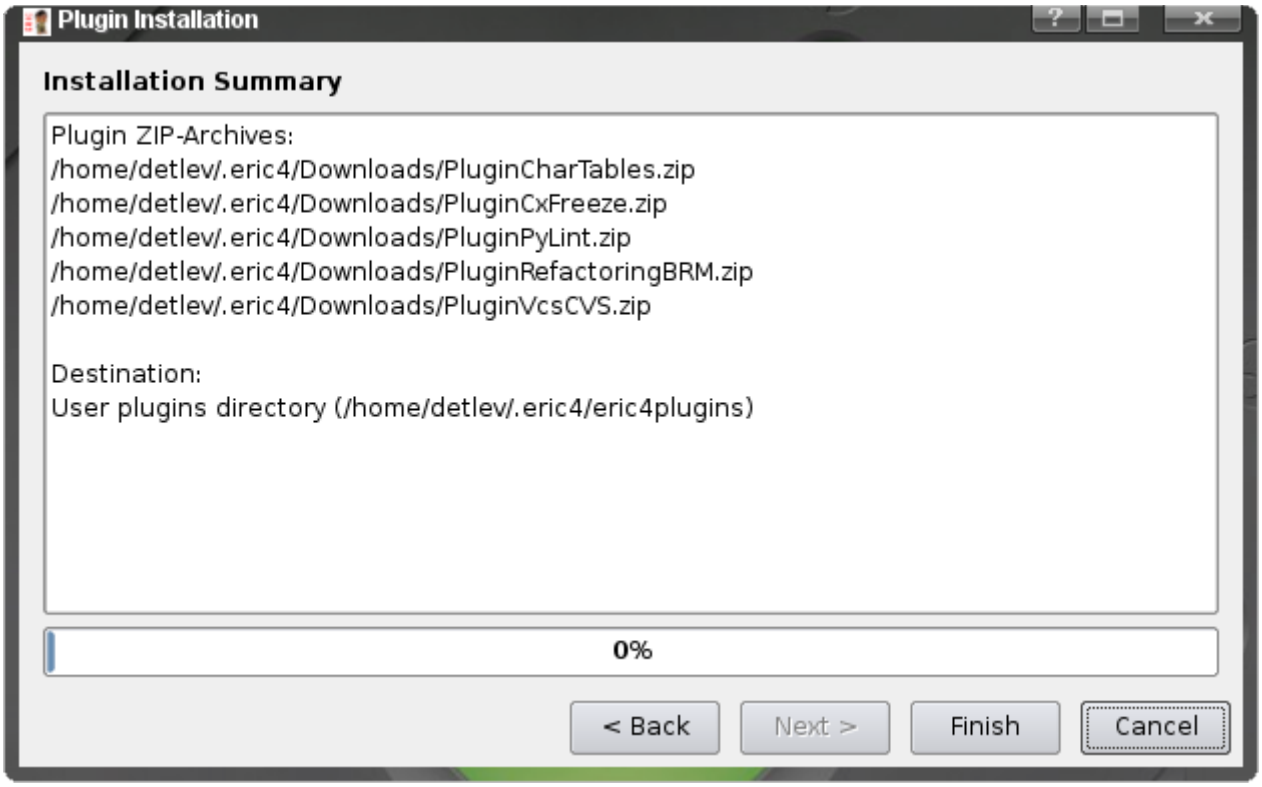

Figure 9: Plug-ins Installation dialog, step 3

The installation progress is show on the very same page. During installation the plug-in archives is checked for various conditions. If the installer recognizes a problem, a message is shown and the installation for this plug-in archive is aborted. If there is a problem in the last step, which is the extraction of the archive, the installation process is rolled back. The installation progress of each plug-in archive is shown by the progress bar.

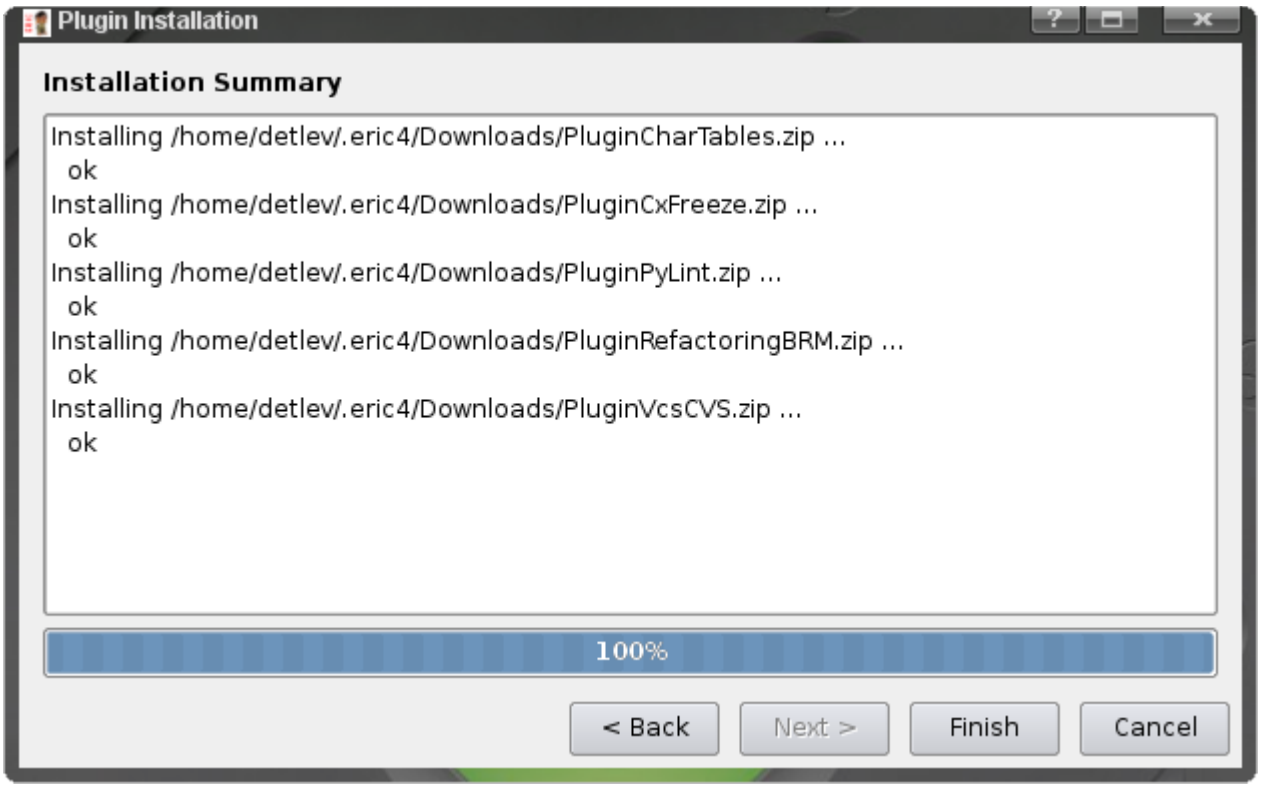

Figure 10: Plug-ins Installation dialog, step 4

Once the installation succeeds, a success message is shown.

Figure 11: Plug-ins Installation dialog, step 5

If plug-ins are installed from within eric6 and are of type “autoactivate”, they are loaded and activated immediately. Otherwise they are loaded in order to add new on-demand functionality.

3.4 Uninstalling Plug-ins

Plug-ins may be uninstalled from within eric6 using the “Uninstall Plugin...” menu, via the `eric6_pluginuninstall.py` script or via the eric6 tray menu. This displays the “Plugin Uninstallation” dialog, which contains two selection list. The top list is used to select the plug-in directory. If the user has write access in the global plug-ins directory, the global and user plug-ins directory are presented. If not, only the user plug-ins directory may be selected. The second list shows the plug-ins installed in the selected plug-ins directory. Pressing the “OK” button starts the uninstallation process.

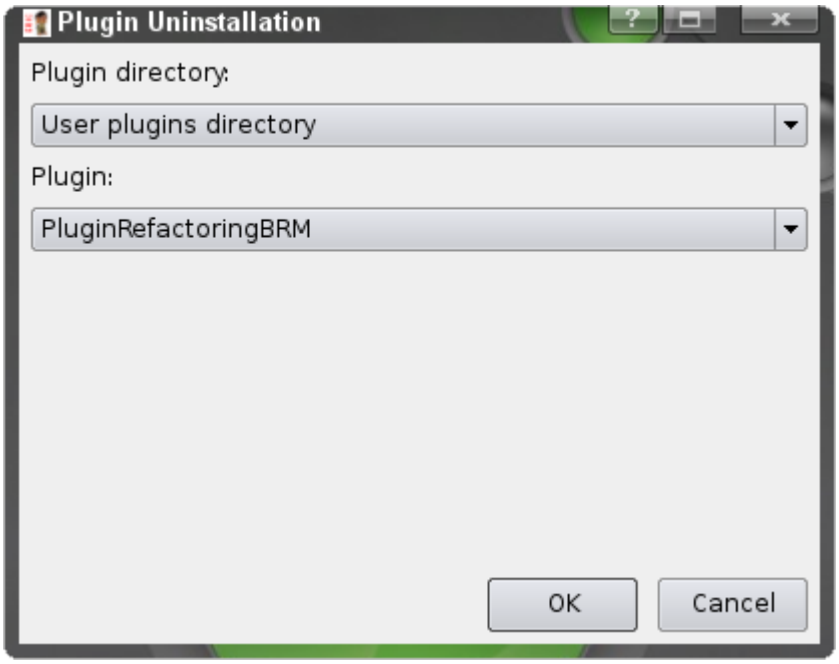

Figure 12: Plug-in Uninstallation dialog, step 1

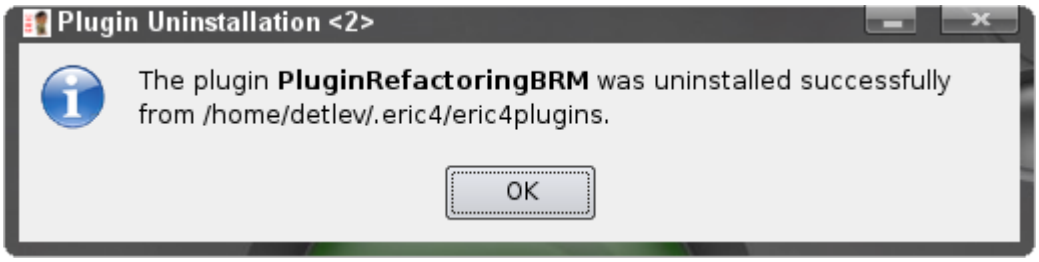

Figure 13: Plug-in Uninstallation dialog, step 2

The uninstallation process deactivates and unloads the plug-in and finally removes all files belonging to the selected plug-in from disk. This process ends with a message confirming successful uninstallation of the plug-in.

3.5 The Plug-ins repository

eric6 has a repository, that contains all official plug-ins. The plug-in repository dialog may be used to show this list and download selected plug-ins.

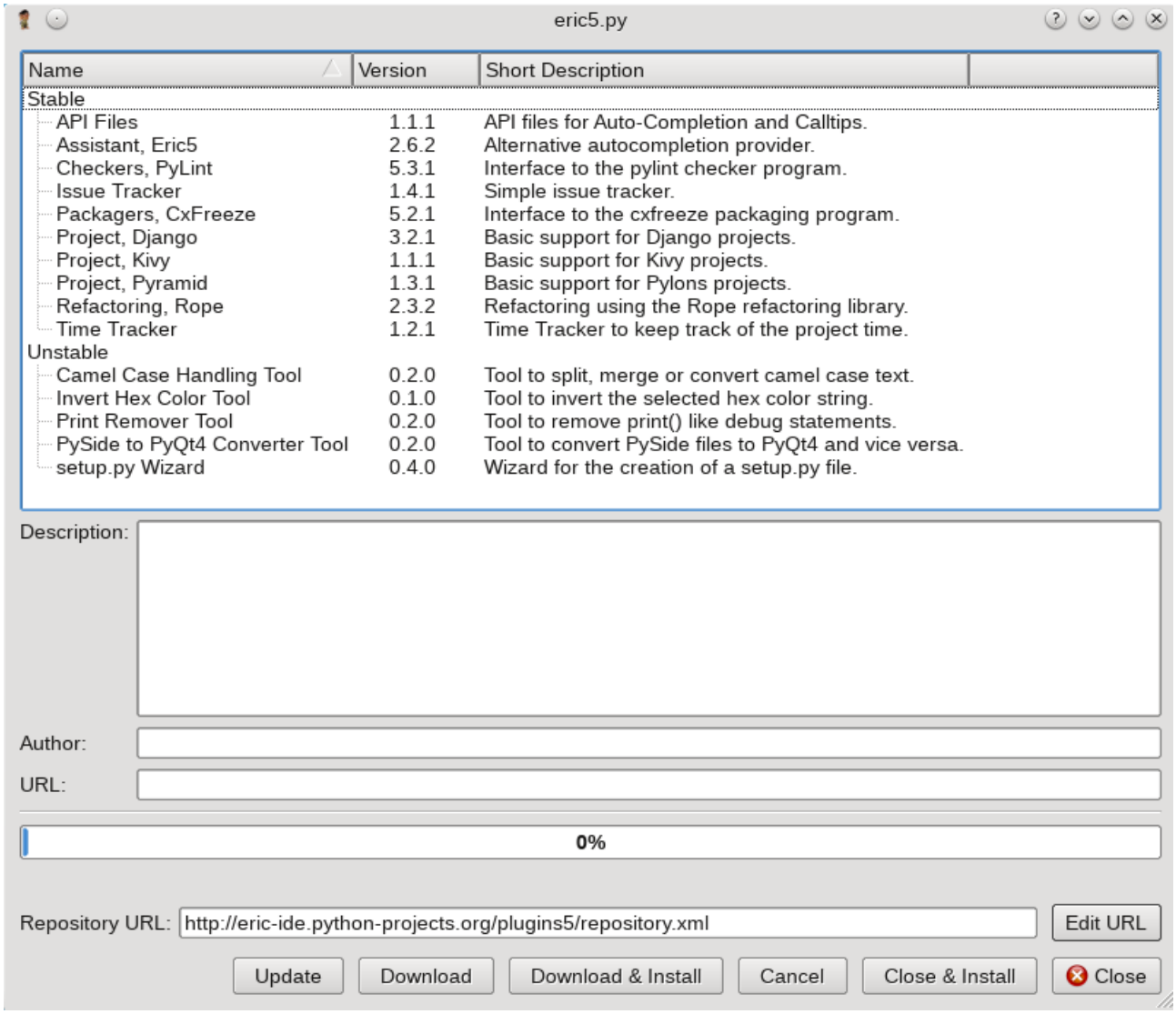

Figure 14: Plug-in Repository dialog

The upper part of the dialog shows a list of available plug-ins. This info is read from a file stored in the eric6 user space. Using the `Update` button, this file can be updated via the Internet. The plug-ins are grouped by their development status. An icon next to the version entry indicates, whether this plug-in needs an update. More detailed data is shown in the bottom part, when an entry is selected. The data shown is the URL of the plug-in, some detailed description and the author of the plug-in. Pressing the `Download` button gets the selected plug-ins from the presented URL and stores them in the user's plug-in download area, which may be configured on the Plug-ins configuration page of the configuration dialog. The `Cancel` button will interrupt the current download. The download progress is shown by the progress bar. Pressing the `Close & Install` button will close this dialog and open the plug-in installation dialog (s. chapter 3.3) The `Download & Install` button download the selected plug-ins, closes the dialog and opens the plug-in installation dialog. The `Repository URL` entry shows the location the repository data is downloaded from. By pressing the `Edit URL` button, this location might be changed by the user in case the location changes and the changed location could not be updated remotely.

4 Eric6 for plug-in developers

This chapter contains a description of functions, that support plug-in development with eric6. Eric6 plug-in projects must have the project type “Eric6 Plugin”. The project's main script must be the plug-in main module. These project entries activate the built-in plug-in development support. These are functions for the creation of plug-in archives and special debugging support. An example of the project properties is shown in the following figure.

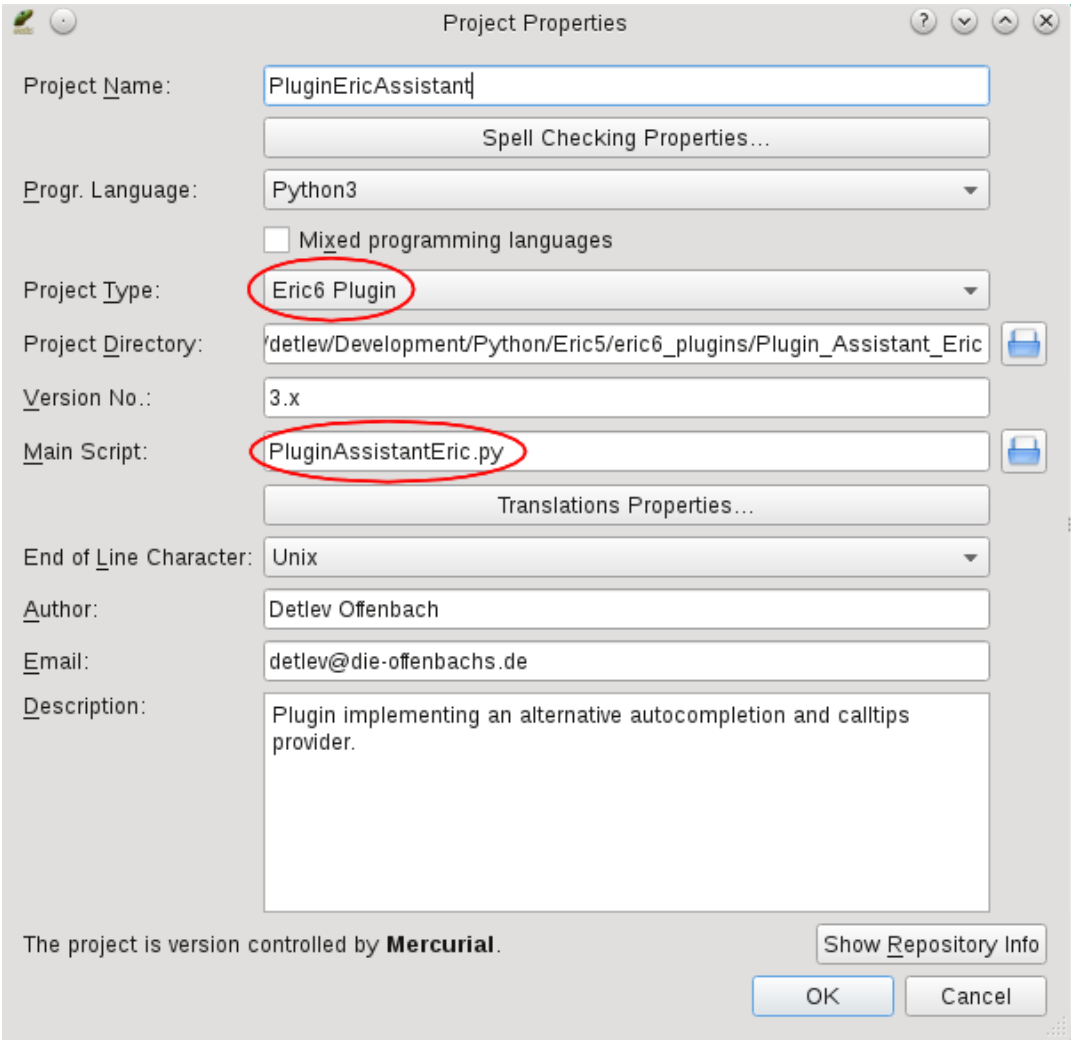

Figure 15: Plug-in specific project properties

To support the creation of plug-in package archives, the Packagers submenu of the Project menu contains entries to ease the creation of a package list and to create the plug-in archive.

Figure 16: Packagers submenu

The “Create package list” entry creates a file called `PKGLIST`, which is used by the archive creator to get the list of files to be included in the plug-in archive. After the `PKGLIST` file has been created, it is automatically loaded into a new editor. The plug-in author should modify this list and shorten it to just include the files required by the plug-in at runtime. The following listing gives an example.

```
AssistantEric/APIsManager.py
AssistantEric/Assistant.py
AssistantEric/ConfigurationPages/AutoCompletionEricPage.py
AssistantEric/ConfigurationPages/AutoCompletionEricPage.ui
AssistantEric/ConfigurationPages/CallTipsEricPage.py
AssistantEric/ConfigurationPages/CallTipsEricPage.ui
AssistantEric/ConfigurationPages/__init__.py
AssistantEric/ConfigurationPages/eric.png
AssistantEric/Documentation/LICENSE.GPL3
AssistantEric/__init__.py
AssistantEric/i18n/assistant_cs.qm
AssistantEric/i18n/assistant_de.qm
AssistantEric/i18n/assistant_en.qm
AssistantEric/i18n/assistant_es.qm
AssistantEric/i18n/assistant_fr.qm
AssistantEric/i18n/assistant_it.qm
AssistantEric/i18n/assistant_ru.qm
AssistantEric/i18n/assistant_zh_CN.GB2312.qm
PluginAssistantEric.py
```

Listing 1: Example of a PKGLIST file

The `PKGLIST` file must be stored in the top level directory of the project alongside the project file.

The archive creator invoked via the “`Create Plugin Archive`” menu entry reads this package list file and creates a plug-in archive. This archive has the same name as the plug-in module and is stored at the same place. The menu entry “`Create Plugin Archive (Snapshot)`” is used to create a snapshot release of the plug-in. This command modifies the version entry of the plug-in module (see below) by appending a snapshot indicator consisting of “-snapshot-” followed by the date like “20141224”.

In order to debug a plug-in under development, eric6 has the command line switch “`--plugin=<plugin module filename>`”. That switch is used internally, if the project is of type “`Eric6 Plugin`”.

5 Anatomy of a plug-in

This chapter describes the anatomy of a plug-in in order to be compatible with eric6.

5.1 Plug-in structure

An eric6 plug-in consists of the plug-in module file and optionally of one plug-in package directory. The plug-in module file must have a filename, that starts with `Plugin` and ends with `.py`, e.g. `PluginRefactoringBRM.py`. The plug-in package directory may have an arbitrary name, but must be unique upon installation. Therefore it is recommended to give it the name of the module without the `Plugin` prefix. This package directory name must be assigned to the packageName module attribute (see the chapter describing the plug-in module header).

5.2 Plug-in header

The plug-in module must contain a plug-in header, which defines various module attributes. An example is given in the listing below.

```
# Start-Of-Header
name = "Assistant Eric Plugin"
author = "Detlev Offenbach <detlev@die-offenbachs.de>"
autoactivate = True
deactivateable = True
version = "1.2.3"
className = "AssistantEricPlugin"
packageName = "AssistantEric"
shortDescription = "Alternative autocompletion and calltips provider."
longDescription = """This plugin implements an alternative autocompletion"""
    """ and calltips provider."""
needsRestart = True
pyqtApi = 2
python2Compatible = True
# End-Of-Header

error = ""
```

Listing 2: Plug-in header

The various attributes to be defined in the header are as follows.

- `name`
  This attribute should contain a short descriptive name of the plug-in.
  Type: string
- `author`
  This attribute should be given the name and the email address of the plug-in author.
  Type: string
- `autoactivate`
  This attribute determines, whether the plug-in may be activated automatically upon startup of eric6. If this attribute is False, the plug-in is activated depending on some configuration settings.

Type: bool

- `deactivateable`
  This attribute determines, whether the plug-in may be deactivated by the user.
  Type: bool
- `version`
  This attribute should contain the version number.
  Type: string
- `className`
  This attribute must contain the name of the class implementing the plug-in. This class must be contained in the plug-in module file.
  Type: string
- `packageName`
  This names the package directory, that contains the rest of the plug-in files. If the plug-in is of the simple type (i.e. all logic is contained in the plug-in module), the packageName attribute must be assigned the value “None” (the string None).
  Type: string
- `shortDescription`
  This attribute should contain a short description of the plug-in and is used in the plug-in info dialog.
  Type: string
- `longDescription`
  This attribute should contain a more verbose description of the plug-in. It is shown in the plug-in details dialog.
  Type: string
- `needsRestart`
  This attribute should make a statement, if eric6 needs to be restarted after plug-in installation or update.
  Type: boolean
- `pyqtApi`
  This attribute should indicate the PyQt QString and QVariant API version the plug-in is coded for. Eric6 plug-ins must support at least version 2.
  Type: integer
- `python2Compatible`
  This attribute is introduced in eric 5.5. Each plug-in has to signalize if it supports the execution when eric is started in a Python 2 interpreter.
  Type: boolean
- `error`
  This attribute should hold an error message, if there was a problem, or an empty string, if everything works fine.
  Type: string
- The '`# Start-Of-Header`' and '`# End-Of-Header`' comments mark the start and the end of the plug-in header.

If the `autoactivate` attribute is False, the header must contain two additional attributes.

```
pluginType = "viewmanager"
pluginTypename = "tabview"
```

Listing 3: Additional header for on-demand plug-ins

- `pluginType`
  This attribute must contain the plug-in type. Currently eric6 recognizes the values “viewmanager” and “version_control”.
  Type: string
- `pluginTypename`
  This attribute must containe the plug-in type name. This is used to differentiate the plug-in within the group of plug-ins of the same plug-in type.
  Type: string

Plug-in modules may define additional optional attributes. Optional attributes recognized by eric6 are as follows.

- `displayString`
  This attribute should contain the user visible string for this plug-in. It should be a translated string, e.g. `displayString = QApplication.translate('VcsCVSPlugin', 'CVS')`. This attribute may only be defined for on-demand plug-ins.
  Type: string

If either the version or the className attribute is missing, the plug-in will not be loaded. If the autoactivate attribute is missing or this attribute is False and the pluginType or the pluginTypename attributes are missing, the plug-in will be loaded but not activated. If the packageName attribute is missing, the plug-in installation will be refused by eric6.

5.3 Plug-in module functions

Plug-in modules may define the following module level functions recognized by the eric6 plug-in manager.

- moduleSetup()
- prepareUninstall()
- getConfigData()
- previewPix()
- exeDisplayData() alternative exeDisplayDataList()
- apiFiles(language)
- clearPrivateData()

These functions are described in more detail in the next few chapters.

5.3.1 moduleSetup()

This function may be defined for on-demand plug-ins (i.e. those with autoactivate being False). It may be used to perform some module level setup. E.g. the CVS plug-in uses this function, to instantiate an administrative object to provide the login and logout menu entries of the version control submenu.

```
def moduleSetup():
    """
    Public function to do some module level setup.
    """
    global __cvsAdminObject
    __cvsAdminObject = CVSAdminObject()
```

Listing 4: Example for the moduleSetup() function

5.3.2 prepareUninstall()

```
import Preferences

def prepareUninstall():
    """
    Module function to prepare for an uninstallation.
    """
    Preferences.Prefs.settings.remove("Refactoring")
    Preferences.Prefs.settings.remove("RefactoringBRM")
```

Listing 5: Example for the prepareUninstall() function

This function is called by the plug-in uninstaller just prior to uninstallation of the plug-in. That is the right place for cleanup code, which removes entries in the settings object or removes plug-in specific configuration files.

5.3.3 getConfigData()

This function may be used to provide data needed by the configuration dialog to show an entry in the list of configuration pages and the page itself. It is called for active autoactivate plug-ins. It must return a dictionary with globally unique keys (e.g. created using the plug-in name) and lists of five entries. These are as follows.

- display string
  The string shown in the selection area of the configuration page. This should be a localized string.
  Type: Qstring
- pixmap name
  The filename of the pixmap to be shown next to the display string.
  Type: string
- page creation function
  The plug-in module function to be called to create the configuration page. The page must be subclasses from `Preferences.ConfigurationPages.ConfigurationPageBase` and must implement a method called 'save' to save the settings. A parent entry will be created in the selection list, if this value is `None`.
  Type: function object or None
- parent key
  The dictionary key of the parent entry or None, if this defines a toplevel entry.
  Type: string or None

- reference to configuration page
  This will be used by the configuration dialog and **must** always be None.
  Type: None

```
def getConfigData():
    """
    Module function returning data as required by the configuration dialog.

    @return dictionary with key "refactoringBRMPage" containing the
                relevant data
    """
    return {
        "refactoringBRMPage" : \
            [QApplication.translate("RefactoringBRMPlugin",
                                    "Refactoring (BRM)"),
             os.path.join("RefactoringBRM", "ConfigurationPage",
                          "preferences-refactoring.png"),
             createConfigurationPage, None, None],
    }
```

Listing 6: Example for the getConfigData() function

5.3.4 previewPix()

This function may be used to provide a preview pixmap of the plug-in. This is just called for viewmanager plug-ins (i.e. pluginType == "viewmanager"). The returned object must be of type QPixmap.

```
def previewPix():
    """
    Module function to return a preview pixmap.

    @return preview pixmap (QPixmap)
    """
    fname = os.path.join(os.path.dirname(__file__),
                         "ViewManagers", "Tabview", "preview.png")
    return QPixmap(fname)
```

Listing 7: Example for the previewPix() function

5.3.5 exeDisplayData()

This function may be defined by modules, that depend on some external tools. It is used by the External Programs info dialog to get the data to be shown. This function must return a dictionary that contains the data for the determination of the data to be shown or a dictionary containing the data to be shown.

The required entries of the dictionary of type 1 are described below.

- `programEntry`
  An indicator for this dictionary form. It must always be True.
  Type: bool
- `header`
  The string to be diplayed as a header.
  Type: Qstring
- `exe`
  The pathname of the executable.
  Type: string
- `versionCommand`
  The version commandline parameter for the executable (e.g. `--version`).
  Type: string
- `versionStartsWith`
  The indicator for the output line containing the version information.
  Type: string
- `versionPosition`
  The number of the element containing the version. Elements are separated by a whitespace character.
  Type: integer
- `version`
  The version string to be used as the default value.
  Type: string
- `versionCleanup`
  A tuple of two integers giving string positions start and stop for the version string. It is used to clean the version from unwanted characters. If no cleanup is required, it must be None.
  Type: tuple of two integers or None

```
def exeDisplayData():
    """
    Public method to support the display of some executable info.

    @return dictionary containing the data to query the presence of
        the executable
    """
    exe = 'pylint'
    if sys.platform == "win32":
        exe = os.path.join(sys.exec_prefix, "Scripts", exe + '.bat')

    data = {
        "programEntry"      : True,
        "header"            : QApplication.translate("PyLintPlugin",
                                "Checkers - Pylint"),
        "exe"               : exe,
        "versionCommand"    : '--version',
        "versionStartsWith" : 'pylint',
        "versionPosition"   : -1,
        "version"           : "",
        "versionCleanup"    : (0, -1),
    }

    return data
```

Listing 8: Example for the exeDisplayData() function returning a dictionary of type 1

The required entries of the dictionary of type 2 are described below.

- `programEntry`
  An indicator for this dictionary form. It must always be False.
  Type: bool
- `header`
  The string to be diplayed as a header.
  Type: string
- `text`
  The entry text to be shown.
  Type: string
- `version`
  The version text to be shown.
  Type: string

```
def exeDisplayData():
    """
    Public method to support the display of some executable info.

    @return dictionary containing the data to be shown
    """
    try:
        import pysvn
        try:
            text = os.path.dirname(pysvn.__file__)
        except AttributeError:
            text = "PySvn"
        version =  ".".join([str(v) for v in pysvn.version])
    except ImportError:
        text = "PySvn"
        version = ""

    data = {
        "programEntry" : False,
        "header"       : QApplication.translate("VcsPySvnPlugin",
                             "Version Control - Subversion (pysvn)"),
        "text"         : text,
        "version"      : version,
    }

    return data
```

Listing 9: Example for the exeDisplayData() function returning a dictionary of type 2

5.3.6 exeDisplayDataList()

In case the plugin has to report more than one external tool, it can define the function exeDisplayDataList in its module. The returned list has to consist of exeDisplayData type 1 or type 2 dictionaries (see 5.3.5 exeDisplayData()).

```
def exeDisplayDataList():
    """
    Public method to support the display of some executable info.

    @return dictionary containing the data to query the presence of
        the executable
    """
    dataList = []

    # 1. eric6_doc
    exe = 'eric6_doc'
    if Utilities.isWindowsPlatform():
        exe = os.path.join(getConfig("bindir"), exe +'.bat')
    dataList.append({
        "programEntry"      : True,
        "header"            : QApplication.translate("EricdocPlugin",
                                "Eric6 Documentation Generator"),
        "exe"               : exe,
        "versionCommand"    : '--version',
        "versionStartsWith" : 'eric6_',
        "versionPosition"   : -3,
        "version"           : "",
        "versionCleanup"    : None,
    })

    # 2. Qt Help Generator
    exe = 'qhelpgenerator'
    if Utilities.isWindowsPlatform():
        exe += '.exe'
    dataList.append({
        "programEntry"      : True,
        "header"            : QApplication.translate("EricdocPlugin",
                                "Qt4 Help Tools"),
        "exe"               : exe,
        "versionCommand"    : '-v',
        "versionStartsWith" : 'Qt',
        "versionPosition"   : -1,
        "version"           : "",
        "versionCleanup"    : (0, -1),
    })

    # 3. Qt Collection Generator
    exe = 'qcollectiongenerator'
    if Utilities.isWindowsPlatform():
        exe += '.exe'
    dataList.append({
        "programEntry"      : True,
        "header"            : QApplication.translate("EricdocPlugin",
                                "Qt4 Help Tools"),
        "exe"               : exe,
        "versionCommand"    : '-v',
        "versionStartsWith" : 'Qt',
        "versionPosition"   : -1,
        "version"           : "",
        "versionCleanup"    : (0, -1),
    })

    return dataList
```

Listing 10: Example for the exeDisplayDataList() function returning a list of dictionaries of type 1

5.3.7 apiFiles(language)

This function may be provided by plug-ins providing API files for the autocompletion and calltips system of eric6. The function must accept the programming language as a string and return the filenames of the provided API files for that language as a list of string.

```
def apiFiles(language):
    """
    Module function to return the API files made available by this plugin.

    @return list of API filenames (list of string)
    """
    if language == "Python":
        apisDir = \
            os.path.join(os.path.dirname(__file__), "ProjectDjango", "APIs")
        apis = glob.glob(os.path.join(apisDir, '*.api'))
    else:
        apis = []
    return apis
```

Listing 11: Example for the apiFiles(language) function

5.3.8 clearPrivateData()

This function may be provided by plug-ins defining private data in order to clear them upon requested by the user.

```
def clearPrivateData():
    """
    Module function to clear the private data of the plug-in.
    """
    for key in ["RepositoryUrlHistory"]:
        VcsMercurialPlugin.setPreferences(key, [])
```

Listing 12: Example for the clearPrivateData() function

5.4 Plug-in object methods

The plug-in class as defined by the `classname` attribute must implement three mandatory methods.

- `__init__(self, ui)`
- `activate(self)`
- `deactivate(self)`

These functions are described in more detail in the next few chapters.

5.4.1 __init__(self, ui)

This method is the constructor of the plug-in object. It is passed a reference to the main window object, which is of type `UI.UserInterface`. The constructor should be used to perform all initialization steps, that are required before the activation of the plug-in object. E.g. this would be the right place to load a translation file for the plug-in (s. Listing 16) and to initialize default values for preferences values.

```
    def __init__(self, ui):
        """
        Constructor

        @param ui reference to the user interface object (UI.UserInterface)
        """
        QObject.__init__(self, ui)
        self.__ui = ui
        self.__initialize()

        self.__refactoringDefaults = {
            "Logging" : 1
        }

        self.__translator = None
        self.__loadTranslator()
```

Listing 13: Example for the __init__(self, ui) method

5.4.2 activate(self)

```
    def activate(self):
        """
        Public method to activate this plugin.

        @return tuple of None and activation status (boolean)
        """
        global refactoringBRMPluginObject
        refactoringBRMPluginObject = self
        self.__object = Refactoring(self, self.__ui)
        self.__object.initActions()
        e5App().registerPluginObject("RefactoringBRM", self.__object)

        self.__mainMenu = self.__object.initMenu()
        extrasAct = self.__ui.getMenuBarAction("extras")
        self.__mainAct = self.__ui.menuBar()\
                             .insertMenu(extrasAct, self.__mainMenu)
        self.__mainAct.setEnabled(\
            e5App().getObject("ViewManager").getOpenEditorsCount())

        self.__editorMenu = self.__initEditorMenu()
        self.__editorAct = self.__editorMenu.menuAction()

        self.connect(e5App().getObject("ViewManager"),
                     SIGNAL('lastEditorClosed'),
                     self.__lastEditorClosed)
        self.connect(e5App().getObject("ViewManager"),
                     SIGNAL("editorOpenedEd"),
                     self.__editorOpened)
        self.connect(e5App().getObject("ViewManager"),
                     SIGNAL("editorClosedEd"),
                     self.__editorClosed)

        self.connect(self.__ui, SIGNAL('preferencesChanged'),
                     self.__object.preferencesChanged)

        self.connect(e5App().getObject("Project"), SIGNAL('projectOpened'),
                     self.__object.projectOpened)
        self.connect(e5App().getObject("Project"), SIGNAL('projectClosed'),
                     self.__object.projectClosed)
        self.connect(e5App().getObject("Project"), SIGNAL('newProject'),
                     self.__object.projectOpened)

        for editor in e5App().getObject("ViewManager").getOpenEditors():
            self.__editorOpened(editor)

        return None, True
```

Listing 14: Example for the activate(self) method

This method is called by the plug-in manager to activate the plug-in object. It must return a tuple giving a reference to the object implementing the plug-in logic (for on-demand plug-ins) or None and a flag indicating the activation status. This method should contain all the logic, that is needed to get the plug-in fully operational (e.g. connect to some signals provided by eric6). If the plug-in wants to provide an action to be added to a toolbar, this

action should be registered with the toolbar manager instead of being added to a toolbar directly.

5.4.3 deactivate(self)

This method is called by the plug-in manager to deactivate the plug-in object. It is called for modules, that have the `deactivateable` module attribute set to True. This method should disconnect all connections made in the activate method and remove all menu entries added in the activate method or somewhere else. If the cleanup operations are not done carefully, it might lead to crashes at runtime, e.g. when the user invokes an action, that is no longer available. If the plug-in registered an action with the toolbar manager, this action must be unregistered.

```
    def deactivate(self):
        """
        Public method to deactivate this plugin.
        """
        e5App().unregisterPluginObject("RefactoringBRM")

        self.disconnect(e5App().getObject("ViewManager"),
                        SIGNAL('lastEditorClosed'),
                        self.__lastEditorClosed)
        self.disconnect(e5App().getObject("ViewManager"),
                        SIGNAL("editorOpenedEd"),
                        self.__editorOpened)
        self.disconnect(e5App().getObject("ViewManager"),
                        SIGNAL("editorClosedEd"),
                        self.__editorClosed)

        self.disconnect(self.__ui, SIGNAL('preferencesChanged'),
                        self.__object.preferencesChanged)

        self.disconnect(e5App().getObject("Project"), SIGNAL('projectOpened'),
                        self.__object.projectOpened)
        self.disconnect(e5App().getObject("Project"), SIGNAL('projectClosed'),
                        self.__object.projectClosed)
        self.disconnect(e5App().getObject("Project"), SIGNAL('newProject'),
                        self.__object.projectOpened)

        self.__ui.menuBar().removeAction(self.__mainAct)

        for editor in self.__editors:
            self.disconnect(editor, SIGNAL("showMenu"), self.__editorShowMenu)
            menu = editor.getMenu("Main")
            if menu is not None:
                menu.removeAction(self.__editorMenu.menuAction())

        self.__initialize()
```

Listing 15: Example for the deactivate(self) method

5.4.4 __loadTranslator(self)

The constructor example shown in Listing 13 loads a plug-in specific translation using this method. The way, how to do this correctly, is shown in the following listing. It is important to keep a reference to the loaded QTranslator object. Otherwise, the Python garbage collector will remove this object, when the method is finished.

```
    def __loadTranslator(self):
        """
        Private method to load the translation file.
        """
        loc = self.__ui.getLocale()
        if loc and loc != "C":
            locale_dir = os.path.join(os.path.dirname(__file__),
                                      "RefactoringBRM", "i18n")
            translation = "brm_%s" % loc
            translator = QTranslator(None)
            loaded = translator.load(translation, locale_dir)
            if loaded:
                self.__translator = translator
                e5App().installTranslator(self.__translator)
            else:
                print "Warning: translation file '%s' could not be loaded." \
                    % translation
                print "Using default."
```

Listing 16: Example for the __loadTranslator(self) method

5.4.5 initToolbar(self, ui, toolbarManager)

This method must be implemented, if the plug-in supports a toolbar for its actions. Such toolbar will be removed, when the plug-in is unloaded. An example is shown in Listing 17.

```
    def initToolbar(self, ui, toolbarManager):
        """
        Public slot to initialize the VCS toolbar.

        @param ui reference to the main window (UserInterface)
        @param toolbarManager reference to a toolbar manager object
            (E5ToolBarManager)
        """
        if self.__projectHelperObject:
            self.__projectHelperObject.initToolbar(ui, toolbarManager)
```

Listing 17: Example for the initToolbar(self, ui, toolbarManager) method

5.4.6 prepareUnload(self)

This method must be implemented to prepare the plug-in to be unloaded. It should revert everything done when the plug-in was instantiated and remove plug-in toolbars generated with `initToolbar()`. Listing 18 shows an example.

```
def prepareUnload(self):
    """
    Public method to prepare for an unload.
    """
    if self.__projectHelperObject:
        self.__projectHelperObject.removeToolbar(
            self.__ui, e5App().getObject("ToolbarManager"))
    e5App().unregisterPluginObject(pluginTypename)
```

Listing 18: Example for the prepareUnload(self) method

6 Eric6 hooks

This chapter describes the various hooks provided by eric6 objects. These hooks may be used by plug-ins to provide specific functionality instead of the standard one.

6.1 Hooks of the project browser objects

Most project browser objects (i.e. the different tabs of the project viewer) support hooks. They provide methods to add and remove hooks.

- `addHookMethod(key, method)`
  This method is used to add a hook method to the individual project browser. “key” denotes the hook and “method” is the reference to the hook method. The supported keys and the method signatures are described in the following chapters.
- `addHookMethodAndMenuEntry(key, method, menuEntry)`
  This method is used to add a hook method to the individual project browser. “key” denotes the hook, “method” is the reference to the hook method and “menuEntry” is the string to be shown in the context menu. The supported keys and the method signatures are described in the following chapters.
- `removeHookMethod(key)`
  This method is used to remove a hook previously added. “key” denotes the hook. Supported keys are described in the followings chapters.

6.1.1 Hooks of the ProjectFormsBrowser object

The `ProjectFormsBrowser` object supports hooks with these keys.

- `compileForm`
  This hook is called to compile a form. The method must take the filename of the form file as its parameter.
- `compileAllForms`
  This hook is called to compile all forms contained in the project. The method must take a list of filenames as its parameter.
- `compileChangedForms`
  This hook is called to compile all changed forms. The method must take a list of

filenames as its parameter.

- `compileSelectedForms`
  This hook is called to compile all forms selected in the project forms viewer. The method must take a list of filenames as its parameter.
- `generateDialogCode`
  This hook is called to generate dialog source code for a dialog. The method must take the filename of the form file as its parameter.
- `newForm`
  This hook is called to generate a new (empty) form. The method must take the filename of the form file as its parameter.
- `open`
  This hook is called to open the selected forms in a forms designer tool. The method must take the filename of the form file as its parameter.

6.1.2 Hooks of the ProjectResourcesBrowser object

The `ProjectResourcesBrowser` object supports hooks with these keys.

- `compileResource`
  This hook is called to compile a resource. The method must take the filename of the resource file as its parameter.
- `compileAllResources`
  This hook is called to compile all resources contained in the project. The method must take a list of filenames as its parameter.
- `compileChangedResources`
  This hook is called to compile all changed resources. The method must take a list of filenames as its parameter.
- `compileSelectedResources`
  This hook is called to compile all resources selected in the project resources viewer. The method must take a list of filenames as its parameter.
- `newResource`
  This hook is called to generate a new (empty) resource. The method must take the filename of the resource file as its parameter.

6.1.3 Hooks of the ProjectTranslationsBrowser object

The `ProjectTranslationsBrowser` object supports hooks with these keys.

- `extractMessages`
  This hook is called to extract all translatable strings out of the application files. The method must not have any parameters. This hook should be used, if the translation system is working with a translation template file (e.g. *.pot) from which the real translation files are generated with the `generate...` methods below.
- `generateAll`
  This hook is called to generate translation files for all languages of the project. The method must take a list of filenames as its parameter.

- `generateAllWithObsolete`
  This hook is called to generate translation files for all languages of the project keeping obsolete strings. The method must take a list of filenames as its parameter.
- `generateSelected`
  This hook is called to generate translation files for languages selected in the project translations viewer. The method must take a list of filenames as its parameter.
- `generateSelectedWithObsolete`
  This hook is called to generate translation files for languages selected in the project translations viewer keeping obsolete strings. The method must take a list of filenames as its parameter.
- `releaseAll`
  This hook is called to release (compile to binary) all languages of the project. The method must take a list of filenames as its parameter.
- `releaseSelected`
  This hook is called to release (compile to binary) all languages selected in the project translations viewer. The method must take a list of filenames as its parameter.
- `open`
  This hook is called to open the selected languages in a translation tool. The method must take the filename of the translations file as its parameter.

6.2 Hooks of the Editor object

The `Editor` object provides hooks for auto-completion and call-tips. These are the methods provided to register, remove and get these hooks and to return completion results.

- `addCompletionListHook(key, func, async=False)`
  This method is used to add a completions provider. The given key must be unique within the set of registered providers. If that is not the case, a KeyError exception is raised. The function or method passed in the call must take a reference to the editor and a flag indicating to complete a context. If the completions provider works asynchronously, the async flag must be set and the function or method must accept a third parameter with the text to be completed. This third parameter must be sent back unaltered with the `completionsListReady()` method below. A synchronous completions provider must return a list of strings giving the possible completions, an asynchronous one must return nothing.
- `removeCompletionListHook(key)`
  This method removes a previously set completions provider.
- `getCompletionListHook(key)`
  This method returns a reference to a previously registered completions provider.
- `completionsListReady(completions, acText)`
  This method must be called by asynchronous completions providers to return the list of possible completions. The first parameter passed to this method is the list of completions and the second one is the text to be completed as given to the registered completions provider method.

- `addCallTipHook(key, func)`
  This method is used to add a call-tips provider. The given key must be unique within the set of registered providers. If that is not the case, a KeyError exception is raised. The function or method passed in the call must take a reference to the editor, a position into the text and the amount of commas to the left of the cursor. It should return the possible calltips as a list of strings.
- `removeCallTipHook(key)`
  This method removes a previously registered call-tips provider.
- `getCallTipHook(key)`
  This method returns a reference to a previously registered call-tips provider.

6.3 Hooks of the CodeDocumentationViewer object

The CodeDocumentationViewer object provides hooks for documentation providers. These are the methods provided to register and unregister a provider and to return the requested documentation.

- `registerProvider(providerName, providerDisplay, provider, supported)`
  This method is used to register a documentation provider. The given provider name must be unique within the set of registered providers. If that is not the case, a KeyError exception is raised. The second parameter must give a string used to show the provider in various places of the documentation viewer. The third parameter must give a function or method used to request documentation. This function must accept a reference to the editor. It is called when the user enters a '(' character or places the cursor somewhere within the text of interest. The fourth parameter passed in must be a function or method used to determine, if a specific programming language is supported by the provider. This function is called with the name of the programming language.
- `unregisterProvider(self, providerName)`
  This method unregisters a previously unregistered documentation provider.
- `documentationReady(self, documentationInfo, isWarning=False, isDocWarning=False)`
  This method is used to return the requested documentation. The first parameter must contain the documentation. This must be either some text in case of a warning or documentation warning or a dictionary with the relevant data. This dictionary should contain text information for these keys.
  - name
    This should contain the name of the inspected object.
  - argspec
    This should contain the argument specification (i.e. a string containing the call parameters).
  - typ
    This should contain the type information of the inspected object (e.g. method).
  - note
    This should contain a note if desired. This could for example be a hint of where the documentation was found. The text could be formatted as HTML text, if the

rich text display is activated. This can be tested with `Preferences.getDocuViewer("ShowInfoAsRichText")`.

- docstring
  This should contain the documentation string. If the rich text display is activated, any line break is converted to an HTML `<br/>` tag (i.e. line breaks are maintained).

All these keys are optional.

7 Eric6 functions available for plug-in development

This chapter describes some functionality, that is provided by eric6 and may be of some value for plug-in development. For a complete eric6 API description please see the documentation, that is delivered as part of eric6.

7.1 The eric6 object registry

Eric6 contains an object registry, that can be used to get references to some of eric6's building blocks. Objects available through the registry are

- `BackgroundService`
  This object gives access to non blocking remote procedure calls to execute functions on different Python versions. Refer to chapter 9 “The BackgroudService“.
- `Cooperation`
  This is the object responsible for chatting between eric6 instantiations and for shared editing.
- `DebugServer`
  This is the interface to the debugger backend.
- `DebugUI`
  This is the object, that is responsible for all debugger related user interface elements.
- `DocuViewer`
  This is the code documentation viewer object. The reference may also be get by using the `documentationViewer()` method of the `UserInterface` object.
- `IRC`
  This object is a simplified Internet Relay Chat client.
- `MultiProject`
  This is the object responsible for the management of a set of projects
- `Numbers`
  This object handles the number conversion.
- `PluginManager`
  This is the object responsible for managing all plug-ins.
- `Project`
  This is the object responsible for managing the project data and all project related user interfaces.

- `ProjectBrowser`
  This is the object, that manages the various project browsers. It offers (next to others) the method `getProjectBrowser()` to get a reference to a specific project browser (s. the chapter below)
- `Shell`
  This is the object, that implements the interactive shell (Python or Ruby).
- `Symbols`
  This object implements the symbol selection lists.
- `SyntaxCheckService`
  This object implements the online syntax check service interface for Python 2 and 3. Other languages can register to this service and getting checked as well. It's described in chapter 9.2 “The SyntaxCheckService“.
- `TaskViewer`
  This is the object responsible for managing the tasks and the tasks related user interface.
- `TemplateViewer`
  This is the object responsible for managing the template objects and the template related user interface.
- `Terminal`
  This is the object, that implements the simple terminal window.
- `ToolbarManager`
  This is the object responsible for managing the toolbars. Toolbars and actions created by a plug-in should be registered and unregistered with the toolbar manager.
- `UserInterface`
  This is eric6 main window object.
- `ViewManager`
  This is the object, that is responsible for managing all editor windows as well as all editing related actions, menus and toolbars.

Eric6's object registry is used as shown in this example.

```
from E5Gui.E5Application import e5App

e5App().getObject("Project")
```

Listing 19: Example for the usage of the object registry

The object registry provides these methods.

- `getObject(name)`
  This method returns a reference to the named object. If no object of the given name is registered, it raises a `KeyError` exception.
- `registerPluginObject(name, object)`
  This method may be used to register a plug-in object with the object registry. “name” must be a unique name for the object and “object” must contain a reference to the object to be registered. If an object with the given name has been registered

already, a `KeyError` exception is raised.

- `unregisterPluginObject(name)`
  This method may be used to unregister a plug-in object. If the named object has not been registered, nothing happens.
- `getPluginObject(name)`
  This method returns a reference to the named plug-in object. If no object of the given name is registered, it raises a `KeyError` exception.
- `getPluginObjects()`
  This method returns a list of references to all registered plug-in objects. Each list element is a tuple giving the name of the plug-in object and the reference.

7.2 The action registries

Actions of type `E5Action` may be registered and unregistered with the `Project` or the `UserInterface` object. In order for this, these objects provide the methods

- `Project.addE5Actions(actions)`
  This method registers the given list of E5Action with the `Project` actions.
- `Project.removeE5Actions(actions)`
  This method unregisters the given list of E5Action from the `Project` actions.
- `UserInterface.addE5Actions(actions, type)`
  This method registers the given list of E5Actions with the `UserInterface` actions of the given type. The `type` parameter may be “ui” or “wizards”
- `UserInterface.removeE5Actions(actions, type)`
  This method unregisters the given list of E5Actions from the `UserInterface` actions of the given type. The `type` parameter may be “ui” or “wizards”

7.3 The getMenu() methods

In order to add actions to menus, the main eric6 objects `Project`, `Editor` and `UserInterface` provide the method `getMenu(menuName)`. This method returns a reference to the requested menu or None, if no such menu is available. `menuName` is the name of the menu as a Python string. Valid menu names are:

- Project
  - `Main`
    This is the project menu
  - `Recent`
    This is the submenu containing the names of recently opened projects.
  - `VCS`
    This is the generic version control submenu.
  - `Checks`
    This is the “Check” submenu.
  - `Show`
    This is the “Show” submenu.

    - `Graphics`
      This is the “Diagrams” submenu.
    - `Session`
      This is the “Session” submenu.
    - `Apidoc`
      This is the “Source Documentation” submenu.
    - `Debugger`
      This is the “Debugger” submenu.
    - `Packagers`
      This is the “Packagers” submenu.
- Editor
    - `Main`
      This is the editor context menu (i.e. the menu appearing, when the right mouse button is clicked)
    - `Resources`
      This is the “Resources” submenu. It is only available, if the file of the editor is a Qt resources file.
    - `Checks`
      This is the “Check” submenu. It is not available, if the file of the editor is a Qt resources file.
    - `Tools`
      This is the “Tools” submenu. It is deactivated, if it has not been populated by some plug-ins.
    - `Show`
      This is the “Show” submenu. It is not available, if the file of the editor is a Qt resources file.
    - `Graphics`
      This is the “Diagrams” submenu. It is not available, if the file of the editor is a Qt resources file.
    - `Autocompletion`
      This is the “Autocomplete” submenu. It is not available, if the file of the editor is a Qt resources file.
    - `Exporters`
      This is the “Exporters” submenu.
    - `Languages`
      This is the submenu for selecting the programming language.
    - `Eol`
      This is the submenu for selecting the end-of-line style.
    - `Encodings`
      This is the submenu for selecting the character encoding.
- UserInterface

- `file`
  This is the “File” menu.
- `edit`
  This is the “Edit” menu.
- `view`
  This is the “View” menu.
- `start`
  This is the “Start” menu.
- `debug`
  This is the “Debug” menu.
- `unittest`
  This is the “Unittest” menu.
- `project`
  This is the “Project” menu.
- `extras`
  This is the “Extras” menu.
- `wizards`
  This is the “Wizards” submenu of the “Extras” menu.
- `macros`
  This is the “Macros” submenu of the “Extras” menu.
- `tools`
  This is the “Tools” submenu of the “Extras” menu.
- `settings`
  This is the “Settings” menu.
- `window`
  This is the “Window” menu.
- `subwindow`
  This is the “Windows” submenu of the “Window” menu
- `toolbars`
  This is the “Toolbars” submenu of the “Window” menu.
- `bookmarks`
  This is the “Bookmarks” menu.
- `plugins`
  This is the “Plugins” menu.
- `help`
  This is the “Help” menu.

7.4 Methods of the PluginManager object

The `PluginManager` object provides some methods, that might be interesting for plug-in development.

- `isPluginLoaded(pluginName)`
  This method may be used to check, if the plug-in manager has loaded a plug-in with the given plug-in name. It returns a boolean flag.

7.5 Methods of the UserInterface object

The `UserInterface` object provides some methods, that might be interesting for plug-in development.

- `getMenuAction(menuName, actionName)`
  This method returns a reference to the requested action of the given menu. `menuName` is the name of the menu to search in (see above for valid names) and `actionName` is the object name of the action.
- `getMenuBarAction(menuName)`
  This method returns a reference to the action of the menu bar associated with the given menu. `menuName` is the name of the menu to search for.
- `registerToolbar(name, text, toolbar)`
  This method is used to register a toolbar. `name` is the name of the toolbar as a Python string, `text` is the user visible text of the toolbar as a string and `toolbar` is a reference to the toolbar to be registered. If a toolbar of the given name was already registered, a KeyError exception is raised.
- `unregisterToolbar(name)`
  This method is used to unregister a toolbar. `name` is the name of the toolbar as a Python string.
- `getToolbar(name)`
  This method is used to get a reference to a registered toolbar. If no toolbar with the given name has been registered, None is returned instead. `name` is the name of the toolbar as a Python string.
- `addSideWidget(side, widget, icon, label)`
  This method is used to add a widget to one of the valid sides. Valid values for the `side` parameter are `UserInterface.LeftSide` and `UserInterface.BottomSide`.
- `removeSideWidget(widget)`
  This method is used to remove a widget that was added using the previously described method. All valid sides will be searched for the widget.
- `getLocale()`
  This method is used to retrieve the application locale as a Python string.
- `versionIsNewer(required, snapshot = None)`
  This method is used to check, if the eric6 version is newer than the one given in the call. If a specific snapshot version should be checked, this should be given as well. “snapshot” should be a string of the form “yyyymmdd”, e.g. “20080719”. If no snapshot is passed and a snapshot version of eric6 is discovered, this method will return True assuming, that the snapshot is new enough. The method returns True, if the eric6 version is newer than the given values.
- `documentationViewer()`

This method is used to get a reference to the documentation viewer object (e.g. to register hook functions).

7.6 Methods of the E5ToolBarManager object

The `E5ToolBarManager` object provides methods to add and remove actions and toolbars. These actions and toolbars are used to build up the toolbars shown to the user. The user may configure the toolbars using a dialog. The list of available actions are those, managed by the toolbar manager.

- `addAction(action, category)`
  This method is used to add an action to the list of actions managed by the toolbar manager. `action` is a reference to a QAction (or derived class); `category` is a string used to categorize the actions.
- `removeAction(action)`
  This method is used to remove an action from the list of actions managed by the toolbar manager. `action` is a reference to a QAction (or derived class).
- `addToolBar(toolBar, category)`
  This method is used to add a toolbar to the list of toolbars managed by the toolbar manager. `toolBar` is a reference to a QToolBar (or derived class); `category` is a string used to categorize the actions of the toolbar.
- `removeToolBar(toolBar)`
  This method is used to remove a toolbar from the list of toolbars managed by the toolbar manager. `toolBar` is a reference to a QToolBar (or derived class).

7.7 Methods of the Project object

The `Project` object provides methods to store and retrieve data to and from the project data store. This data store is saved in the project file.

- `getData(category, key)`
  This method is used to get data out of the project data store. `category` is the category of the data to get and must be one of
  - CHECKERSPARMS
    Used by checker plug-ins.
  - PACKAGERSPARMS
    Used by packager plug-ins.
  - DOCUMENTATIONPARMS
    Used by documentation plug-ins.
  - OTHERTOOLSPARMS
    Used by plug-ins not fitting the other categories.

  The `key` parameter gives the key of the data entry to get and is determined by the plug-in. A copy of the requested data is returned.
- `setData(category, key, data)`
  This method is used to store data in the project data store. `category` is the category of the data to store and must be one of

- CHECKERSPARMS
  Used by checker plug-ins.
- PACKAGERSPARMS
  Used by packager plug-ins.
- DOCUMENTATIONPARMS
  Used by documentation plug-ins.
- OTHERTOOLSPARMS
  Used by plug-ins not fitting the other categories.

The `key` parameter gives the key of the data entry to get and is determined by the plug-in. `data` is the data to store. The data is copied to the data store by using the Python function `copy.deepcopy()`.

In addition to this the `Project` object contains methods to register and unregister additional project types.

- `registerProjectType(type_, description, fileTypeCallback = None, binaryTranslationsCallback = None, lexerAssociationCallback = None)`
  This method registers a new project type provided by the plugin. The parameters to be passed are
  - `type_`
    This is the new project type as a Python string.
  - `description`
    This is the string shown by the user interface. It should be a translatable string of the project type as a `string`.
  - `fileTypeCallback`
    This is a reference to a function or method returning a dictionary associating a filename pattern with a file type (e.g. *.html -> FORMS). The file type must be one of
    - `FORMS`
    - `INTERFACES`
    - `RESOURCES`
    - `SOURCES`
    - `TRANSLATIONS`
  - `binaryTranslationsCallback`
    This is a reference to a function or method returning the name of the binary translation file given the name of the raw translation file.
  - `lexerAssociationCallback`
    This is a reference to a function or method returning the lexer name to be used for syntax highlighting given the name of a file (e.g. *.html -> Django)
- `unregisterProjectType(self, type_)`
  This method unregisters a project type previously registered with the a.m. method. `type_` must be a known project type.

7.8 Methods of the ProjectBrowser object

The `ProjectBrowser` object provides some methods, that might be interesting for plug-in development.

- `getProjectBrowser(name)`
  This method is used to get a reference to the named project browser. `name` is the name of the project browser as a Python string. Valid names are
  - `sources`
  - `forms`
  - `resources`
  - `translations`
  - `interfaces`
  - `protocols`
  - `others`
- `getProjectBrowsers()`
  This method is used to get references to all project browsers. They are returned as a Python list in the order
  - project sources browser
  - project forms browser
  - project resources browser
  - project translations browser
  - project interfaces browser
  - project protocols browser
  - project others browser
- `getProjectBrowserNames()`
  This method is used to get the names of all browsers. They are returned in the same order as above. These names may be used in a call to the `getProjectBrowser`() method.

7.9 Methods of QScintilla.Lexer

The `QScintilla.Lexer` package provides methods to register and unregister lexers (syntax highlighters) provided by a plugin.

- `registerLexer(name, displayString, filenameSample, getLexerFunc, openFilters = [], saveFilters = [], defaultAssocs = [])`
  This method is used to register a new custom lexer. The parameters are as follows.
  - `name`
    This parameter is the name of the new lexer as a Python string.
  - `displayString`

This parameter is the string to be shown in the user interface as a `string`.

- `filenameSample`
  This parameter should give an example filename used to determine the default lexer of a file based on its name (e.g. dummy.django). This parameter should be given as a Python string.
- `getLexerFunc`
  This is a reference to a function instantiating the specific lexer. This function must take a reference to the parent as its only argument and return the reference to the instantiated lexer object.
- `openFilters`
  This is a list of open file filters to be used in the user interface as a `list of strings.`.
- `saveFilters`
  This is a list of save file filters to be used in the user interface as a `list of strings`.
- `defaultAssocs`
  This gives the default lexer associations as a list of strings of filename wildcard patterns to be associated with the lexer

- `unregisterLexer(name)`
  This method is used to unregister a lexer previously registered with the a.m. method. name must be a registered lexer.

7.10 Signals

This chapter lists some Python type signals emitted by various eric6 objects, that may be interesting for plug-in development.

- `showMenu`
  This signal is emitted with the menu name as a Python string and a reference to the menu object, when a menu is about to be shown. It is emitted by these objects.
  - `Project`
    It is emitted for the menus
    - `Main`
      the Project menu
    - `VCS`
      the Version Control submenu
    - `Checks`
      the Checks submenu
    - `Packagers`
      the Packagers submenu
    - `ApiDoc`
      the Source Documentation submenu
    - `Show`
      the Show submenu

        - `Graphics`
          the Diagrams submenu
- `ProjectSourcesBrowser`
  It is emitted for the menus
    - `Main`
      the context menu for single selected files
    - `MainMulti`
      the context menu for multiple selected files
    - `MainDir`
      the context menu for single selected directories
    - `MainDirMulti`
      the context menu for multiple selected directories
    - `MainBack`
      the background context menu
    - `Show`
      the Show context submenu
    - `Checks`
      the Checks context submenu
    - `Graphics`
      the Diagrams context submenu
- `ProjectFormsBrowser`
  It is emitted for the menus
    - `Main`
      the context menu for single selected files
    - `MainMulti`
      the context menu for multiple selected files
    - `MainDir`
      the context menu for single selected directories
    - `MainDirMulti`
      the context menu for multiple selected directories
    - `MainBack`
      the background context menu
- `ProjectResourcesBrowser`
  It is emitted for the menus
    - `Main`
      the context menu for single selected files
    - `MainMulti`
      the context menu for multiple selected files
    - `MainDir`
      the context menu for single selected directories

        - `MainDirMulti`
          the context menu for multiple selected directories
        - `MainBack`
          the background context menu
    - `ProjectTranslationsBrowser`
      It is emitted for the menus
        - `Main`
          the context menu for single selected files
        - `MainMulti`
          the context menu for multiple selected files
        - `MainDir`
          the context menu for single selected directories
        - `MainBack`
          the background context menu
    - `ProjectInterfacesBrowser`
      It is emitted for the menus
        - `Main`
          the context menu for single selected files
        - `MainMulti`
          the context menu for multiple selected files
        - `MainDir`
          the context menu for single selected directories
        - `MainDirMulti`
          the context menu for multiple selected directories
        - `MainBack`
          the background context menu
    - `ProjectOthersBrowser`
      It is emitted for the menus
        - `Main`
          the context menu for single selected files
        - `MainMulti`
          the context menu for multiple selected files
        - `MainBack`
          the background context menu
    - `Editor`
      It is emitted for the menus
        - `Main`
          the context menu
        - `Languages`
          the Languages context submenu

        - `Encodings`
          the Encodings context submenu
        - `Eol`
          the End-of-Line Type context submenu
        - `Autocompletion`
          the Autocomplete context submenu
        - `Show`
          the Show context submenu
        - `Graphics`
          the Diagrams context submenu
        - `Margin`
          the margin context menu
        - `Checks`
          the Checks context submenu
        - `Tools`
          the Tools context submenu
        - `Resources`
          the Resources context submenu
    - `UserInterface`
      It is emitted for the menus
        - `File`
          the File menu
        - `Extras`
          the Extras menu
        - `Wizards`
          the Wizards submenu of the Extras menu
        - `Tools`
          the Tools submenu of the Extras menu
        - `Help`
          the Help menu
        - `Windows`
          the Windows menu
        - `Subwindows`
          the Windows submenu of the Windows menu
- `editorOpenedEd`
  This signal is emitted by the `ViewManager` object with the reference to the editor object, when a new editor is opened.
- `editorClosedEd`
  This signal is emitted by the `ViewManager` object with the reference to the editor object, when an editor is closed.
- `lastEditorClosed`

This signal is emitted by the ViewManager object, when the last editor is closed.

- projectOpenedHooks()
  This signal is emitted by the Project object after a project file was read but before the projectOpened() signal is sent.
- projectClosedHooks()
  This signal is emitted by the Project object after a project file was clsoed but before the projectClosed() signal is sent.
- newProjectHooks()
  This signal is emitted by the Project object after a new project was generated but before the newProject() signal is sent.
- projectOpened
  This signal is emitted by the Project object, when a project is opened.
- projectClosed
  This signal is emitted by the Project object, when a project is closed.
- newProject
  This signal is emitted by the Project object, when a new project has been created.
- preferencesChanged
  This signal is emitted by the UserInterface object, when some preferences have been changed.
- EditorAboutToBeSaved
  This signal is emitted by the each Editor object, when the editor contents is about to be saved. The filename is passed as a parameter.
- EditorSaved
  This signal is emitted by the each Editor object, when the editor contents has been saved. The filename is passed as a parameter.
- EditorRenamed
  This signal is emitted by the each Editor object, when the editor has received a new filename.

8 Special plug-in types

This chapter describes some plug-ins, that have special requirements.

8.1 VCS plug-ins

VCS plug-ins are loaded on-demand depending on the selected VCS system for the current project. VCS plug-ins must define their type by defining the module attribute pluginType like

```
pluginType = "version_control"
```

VCS plug-ins must implement the getVcsSystemIndicator() module function. This function must return a dictionary with the indicator as the key as a Python string and a

tuple of the VCS name (Python string) and the VCS display string (string) as the value. An example is shown below.

```
def getVcsSystemIndicator():
    """
    Public function to get the indicators for this version control system.

    @return dictionary with indicator as key and a tuple with the vcs name
        (string) and vcs display string (string)
    """
    global displayString, pluginTypename
    data = {}
    data[".svn"] = (pluginTypename, displayString)
    data["_svn"] = (pluginTypename, displayString)
    return data
```

Listing 20: Example of the getVcsSystemIndicator() function

8.2 ViewManager plug-ins

ViewManager plug-ins are loaded on-demand depending on the selected view manager. The view manager type to be used may be configured by the user through the configuration dialog. ViewManager plug-ins must define their type by defining the module attribute `pluginType` like

```
pluginType = "viewmanager"
```

The plug-in module must implement the `previewPix()` method as described above.

9 The BackgroudService

Introduced with Eric 5.5, the background service becomes part of the core system. It's a kind of remote procedure call, but other than, e.g. XMLRPC or CORBA, it's non blocking. Mainly developed to simplify the problems with some core modules, where the execution depends on the different Python versions, it could be used by other plug-ins as well. Even other languages than Python could be attached to the server side of the background service, to enhance Eric 5.

On the start of Eric, typically the Python 2 and 3 interpreters are started with Eric and some core plug-ins use them. Which interpreter is started, depends on the interpreter given in Settings → Debugger.

Based on the BackgroudService there are some core plug-ins which use it already to do their tasks.

9.1 How to access the background service

The interface, the background service supports, is quite simple. First of all, a plug-in has to get access to it through the object registry (refer to 7.1 “The eric6 object registry“).

Now it has access to the background service interface (the server side) and can announce its functions. Therefore the method `serviceConnect` must be called. To keep the

background service universal, a plug-in has to specify, e.g. the callback function which itself can emit a self defined signal.

```
self.backgroundService = e5App().getObject("BackgroundService")

self.backgroundService.serviceConnect(
    'style', lang, path, 'CodeStyleChecker',
    self.__translateStyleCheck,
    lambda fx, fn, ver, msg: self.styleChecked.emit(
        fn, {}, 0, [[0, 0, '---- ' + msg, False, False]]))
```

Listing 21: Example of a serviceConnect

The signature is

```
serviceConnect(fx, lang, modulepath, module, callback,
    onErrorCallback=None)
```

with

- `fx`
  Function name with which the service should be named.
- `lang`
  Language for which this call should be implemented.
- `modulepath`
  Full path to the module.
- `module`
  Module name without extension.
- `callback`
  Function which should be called after successful execution.
- `onErrorCallback`
  Function which should be called if anything unexpected happened.

Each plug-in which is based on Python has to support a special function `initService`. The `initService` function has to return the main service function pointer and has to initialize everything that is needed by the plug-in main function, e.g. create the object if it's a class method instead of a simple function.

After a successful `serviceConnect`, the plug-in can request a remote procedure call through the `enqueueRequest` method provided by the background service. The plug-in therefore has to use the registered service name. Furthermore it has to provide the language to use and an identifier. This identifier can be used to match the `enqueueRequest` call with the corresponding callback. Typically the file name is used, but other basic data types like integer or float could be used. The last parameter contains a list of function arguments, which are transferred to the remote procedure. Any basic data type can be used as arguments, but tuples are converted to lists by the underlying JSON module.

```
self.backgroundService.enqueueRequest('syntax', lang, filename, data)
```

Listing 22: Example of enqueueing a request

The signature is

`enqueueRequest(fx, lang, fn, data)`

with

- `fx`
  Function name with which the service should be named.
- `lang`
  Language for which this call should be implemented.
- `fn`
  Identifier to determine the callback to the service request.
- `data`
  List of any basic datatypes. These are the former arguments of the method call.

As the method name implies, the call of `enqueueRequest` only enqueues the request. If other requests are pending, the processing waits until it's his turn. In the current implementation this is also true if the language to use isn't busy. Future plug-ins should therefore be cooperative and wait for the response instead of enqueueing all their tasks. To avoid an overflow, only the arguments of a pending task are updated. This is the case if the service name, the language and the identifier are all the same on a new `enqueueRequest` call. But the position in the queue isn't changed.

On unload of a plug-in, it can remove the connection to the background service by calling `serviceDisconnect`.

```
self.backgroundService.serviceDisconnect('syntax', lang)
```

Listing 23: Example of disconnecting from a service

The signature is

`serviceDisconnect(fx, lang)`

with

- `fx`
  Function name which should be disconnected.
- `lang`
  Language for which the function name `fx` should be disconnected.

9.2 The SyntaxCheckService

Based on the background service, another general service was introduced. With the syntax check service, other languages than Python can implement a syntax check and reuse the dialogs and recurring checks for open files. Therefore a special interface is created to include the new language to the existing checking mechanism.

Like the background service, the SyntaxCheckService is also added to the Eric 5 object registry (see SyntaxCheckService).

A new language has to register itself to the syntax checker by calling `addLanguage`. Additionally the new language has to implement the client side of the syntax checker. One

way is to use the existing client side implemented in Python to call the checker. But this is very slow because of the overhead which comes from starting the syntax checker over and over again. It's better, to implement a new client side in the programming language the checker finally is. A good starting point for this is to look in Utilities/BackgroundClient.py.

`addLanguage` takes some parameters to handle the new programming language. The example shows the call from PluginSyntaxChecker.py

```
self.syntaxCheckService.addLanguage(
        'Python2', 'Python2', path, 'SyntaxCheck',
        self.__getPythonOptions,
        lambda: Preferences.getPython("PythonExtensions"),
        self.__translateSyntaxCheck,
        lambda fx, lng, fn, msg: \
            self.syntaxCheckService.syntaxChecked.emit(
                fn, True, fn, 0, 0, '', msg, []))
```

Listing 24: Example of registering a language

The signature is

`addLanguage(lang, env, path, module, getArgs, getExt, callback, onError)`

with

- `lang`
  The language which is to be registered. The name of the language is used in subsequent calls from the checker dialog.
- `env`
  The environment in which the checker is implemented.
- `path`
  The full path to the module which has to be imported.
- `module`
  The name of the module.
- `getArgs`
  Function pointer: Options and parameters which could be set by the user, e.g. through the preferences menu, are returned. It's called before every check of a file.
- `getExt`
  Function pointer: Returns a list of extensions which are supported by the language checker plugin.
- `callback`
  Function pointer: When the syntax check request has finished, this method is called by the background service.
- `onError`
  Function pointer: If an error happens, the callback function won't be called by the background service. To report that error and continue with the next request, the `onError` function is called. It should generate the same signature like the `callback` function.

path and module are the same as in the background service `serviceConnect` method.

Depending on the import mechanisms of the language and the client implementation it may be not necessary to provide path and / or module. In this case just empty strings should be enough.

The problems reported back to the callback method are stored in a dictionary which can hold two keys: error and warnings. The values of those keys are similar: the error key holds only the first five arguments and is a one dimensional list. The warnings key holds a two dimensional list (list of lists) and uses all arguments. The arguments and their sequence in the list are as follows:

- `filename`
  The file name where the problem was found. This should always be the same like the checked file name.
- `line`
  The line number starting from 1 where the problem was found.
- `column`
  The column where the problem was found or 0 when it wasn't possible to determine the exact position.
- `code`
  In case of a syntax error the source code line otherwise an empty string.
- `message`
  The message of the problem. The translation is done on the server side because the client has not to know how to translate the messages to the installed user language.
- `arguments`
  A list of arguments which has to be inserted into the translated message text. Does not exist in the error key.

It's also possible to deactivate a language by calling `removeLanguage` with the name of the language.

To query which languages are already registered, a plug-in can call `getLanguages` to get a list with the names of the registered languages.

To filter out the unsupported files, a plug-in can check for a correct file extension by retrieving the registered extensions with a call of `getExtensions`. It returns a list of supported file extensions.

At last a plug-in could start a check for a file by itself, by calling `syntaxCheck`. The signature is

```
syntaxCheck(lang, filename, source="")
```

with

- `lang`
  The language which is to be used or None to determine the language based on its extension.
- `filename`
  The file name or unique identifier like in enqueueRequest (see identifier page 51).
- `source`

The source code to check.