The eric4 plugin system

Table of contents

List of figures

List of listings

1 Introduction

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

2 Description of the plugin system

The eric4 plugin system is the extensible part of the eric4 IDE. There are two kinds of plugins. The first kind of plugins are automatically activated at startup, the other kind are activated on demand. The activation of the on-demand plugins is controlled by configuration options. Internally, all plugins are managed by the PluginManager object. Deactivated autoactivate plugins are remembered and will not be activated automatically on the next start of eric4.

Eric4 comes with quite a number of core plugins. These are part of the eric4 installation. In addition to this, there are additional plugins available via the internet. Those plugins may be installed and uninstalled using the provided menu or toolbar entries. Installable plugins live in one of two areas. One is the global plugin area, the other is the user plugin area. The later one overrides the global area.

3 The plugin system from a user perspective

The eric4 plugin system provides the user with a Plugins 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 plugins and to install or uninstall plugins.

3.1 The Plugins menu and toolbar

The plugins menu is located under the “Plugins” label in the main menu bar of the eric4 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.

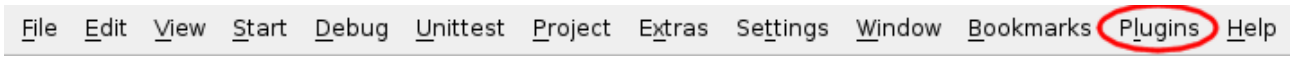

Figure 1: eric4 main menu

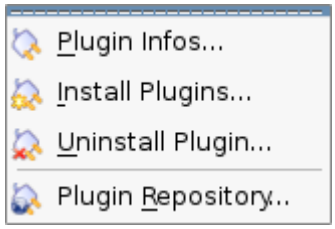

Figure 2: The Plugins menu

Figure 3: The Plugins toolbar

The “Plugin Infos...” action is used to show a dialog, that lists all the loaded plugins and there status. The entry labeled “Install Plugins...” opens a wizards like dialog to install new plugins from plugin archives. The entry, “Uninstall Plugin...”, presents a dialog to uninstall a plugin. If a plugin to be uninstalled is loaded, it is unloaded first. The last entry called “Plugin Repository...” shows a dialog, that displays the official plugins available in the eric4 plugin repository.

3.2 The Plugin Infos dialog

The “Plugin Infos” dialog show information about all loaded plugins. Plugins, 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.

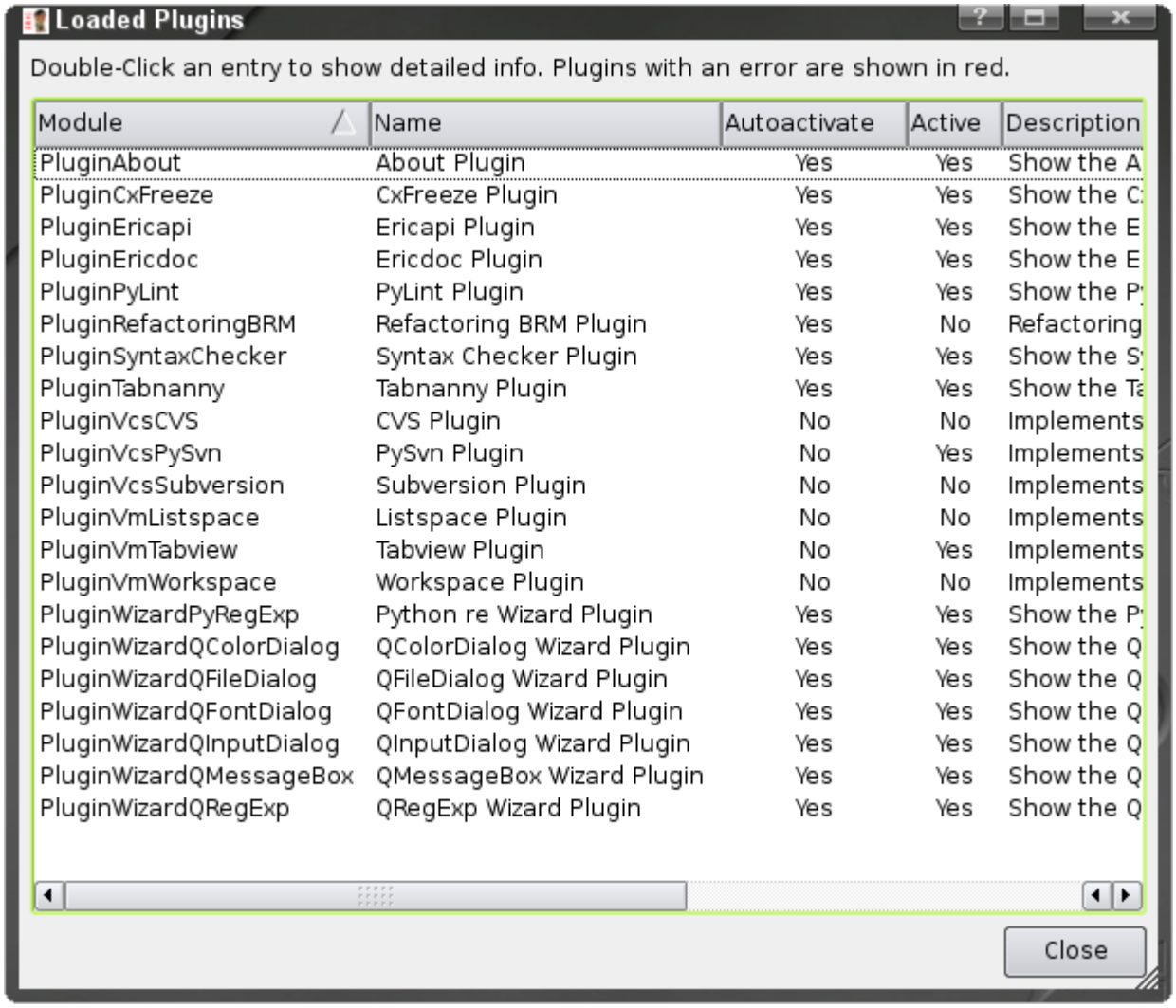

Figure 4: Plugins Info dialog

The columns show information as follows.

- Module
  This shows the Python module name of the plugin. It is usually the name of the

plugin file without the file extension. The module name must be unique.

- Name
  This is the name of the plugin as given by the plugin author.
- Autoactivate
  This indicates, if the plugin should be activated at startup of the eric4 IDE. The actual activation of a plugin is controlled by the state it had at the last shutdown of eric4.
- Active
  This gives an indication, if the plugin is active.
- Description
  This column show a descriptive text as given by the plugin author.

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

Figure 5: Plugins Info dialog context menu

Deactivated plugins are remembered and will not be activated automatically at the next startup of eric4. 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 plugin. An example is shown in the following figure.

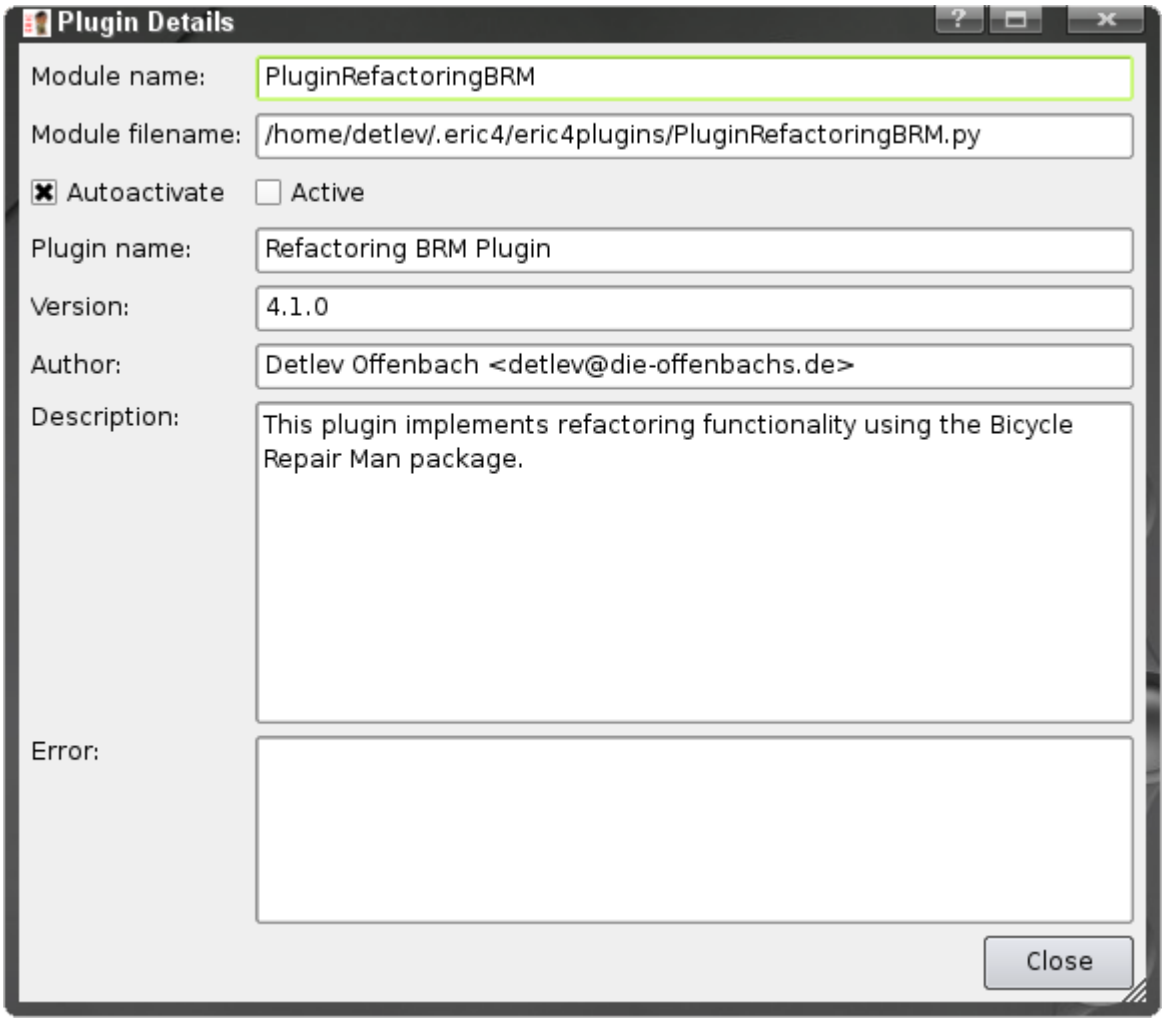

Figure 6: Plugin Details dialog

The entries of the dialog are as follows.

- Module name:
  This shows the Python module name of the plugin. It is usually the name of the plugin file without the file extension. The module name must be unique.
- Module filename:
  This shows the complete path to the installed plugin Python file.
- Autoactivate
  This indicates, if the plugin should be activated at startup of the eric4 IDE. The actual activation of a plugin is controlled by the state it had at the last shutdown of eric4.
- Active
  This gives an indication, if the plugin is active.
- Plugin name:
  This is the name of the plugin as given by the plugin author.
- Version:
  This shows the version number of the installed plugin. This number should be

passed to the plugin author when reporting a problem.

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

3.3 Installing Plugins

New plugins are installed from within eric4 using the Plugin 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 `eric4-plugininstall.py` script or via the eric4 tray menu. The user is guided through the installation process by a wizard like dialog. On the first page, the plugin archives are selected. eric4 plugins 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.

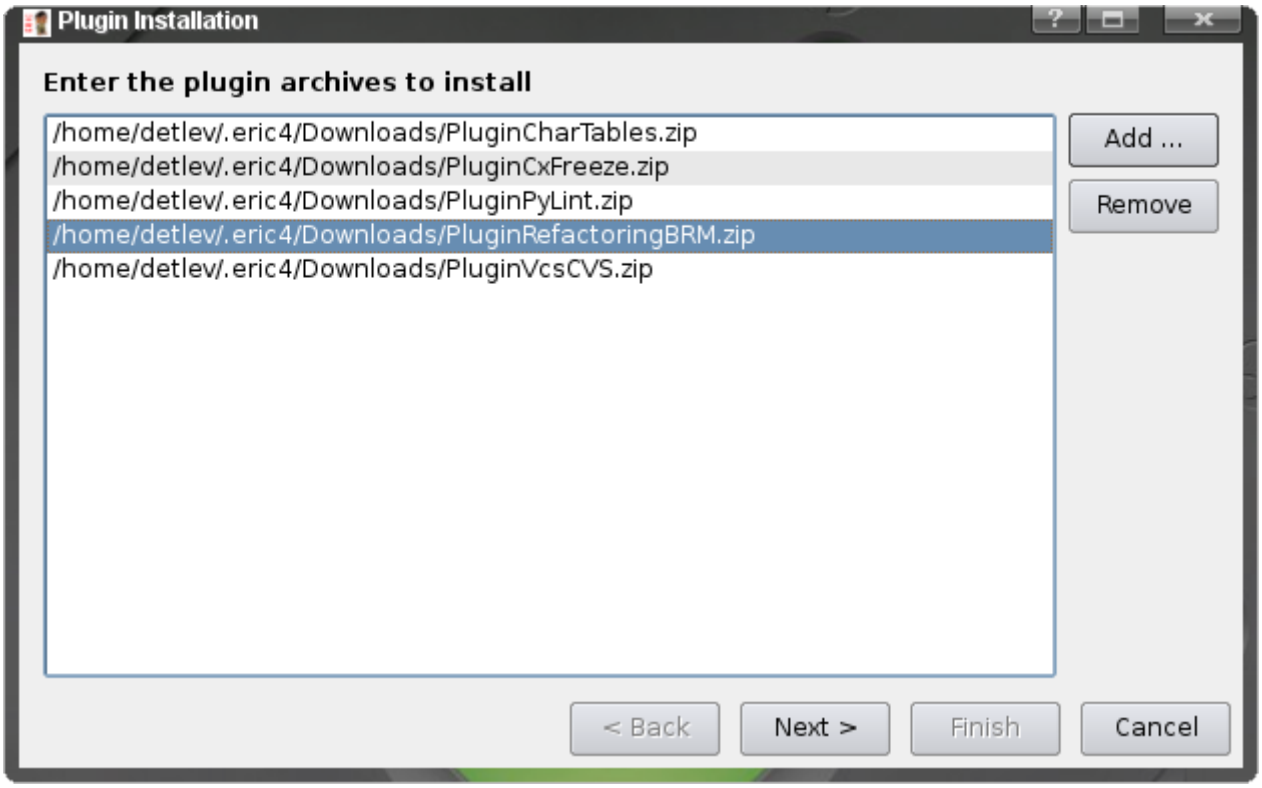

Figure 7: Plugins Installation dialog, step 1

The second display of the dialog is used to select the directory, the plugin should be installed into. If the user has write access to the global eric4 plugins directory, both the

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

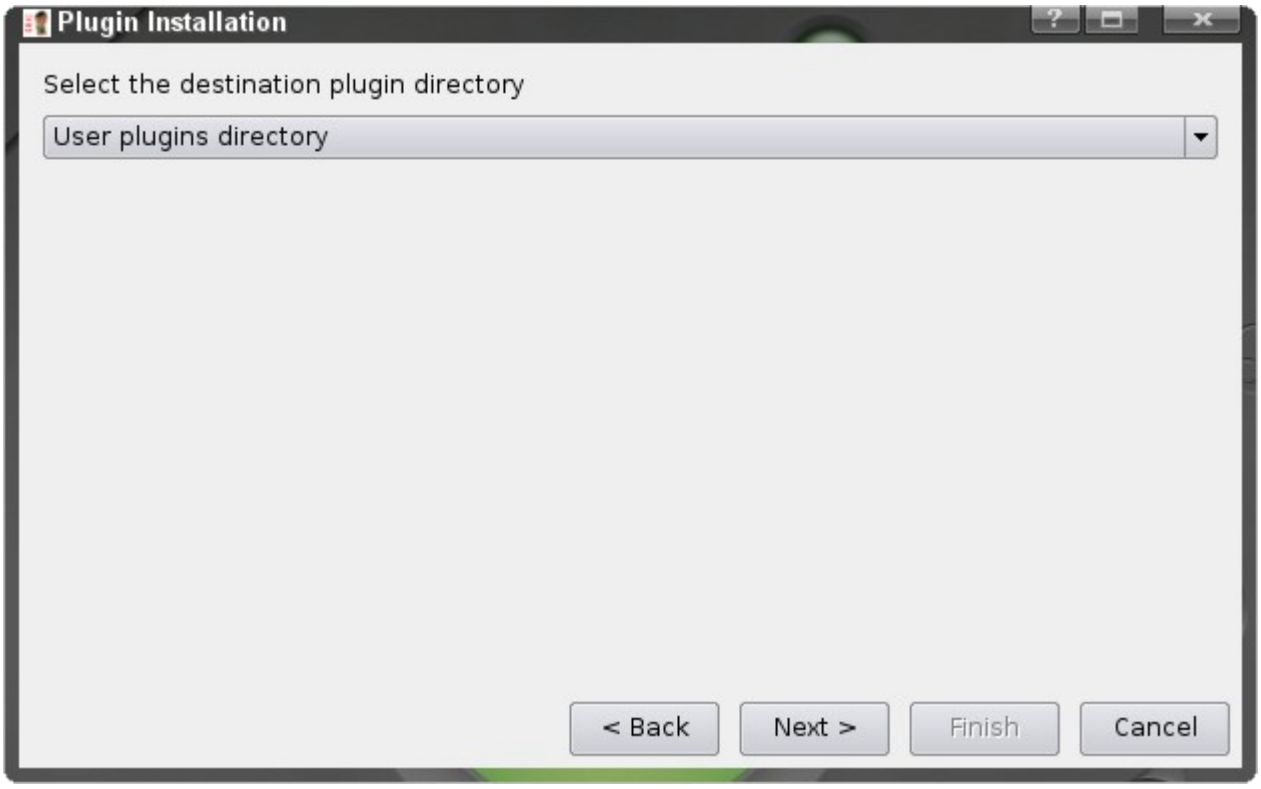

Figure 8: Plugins Installation dialog, step 2

The final display of the plugin 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.

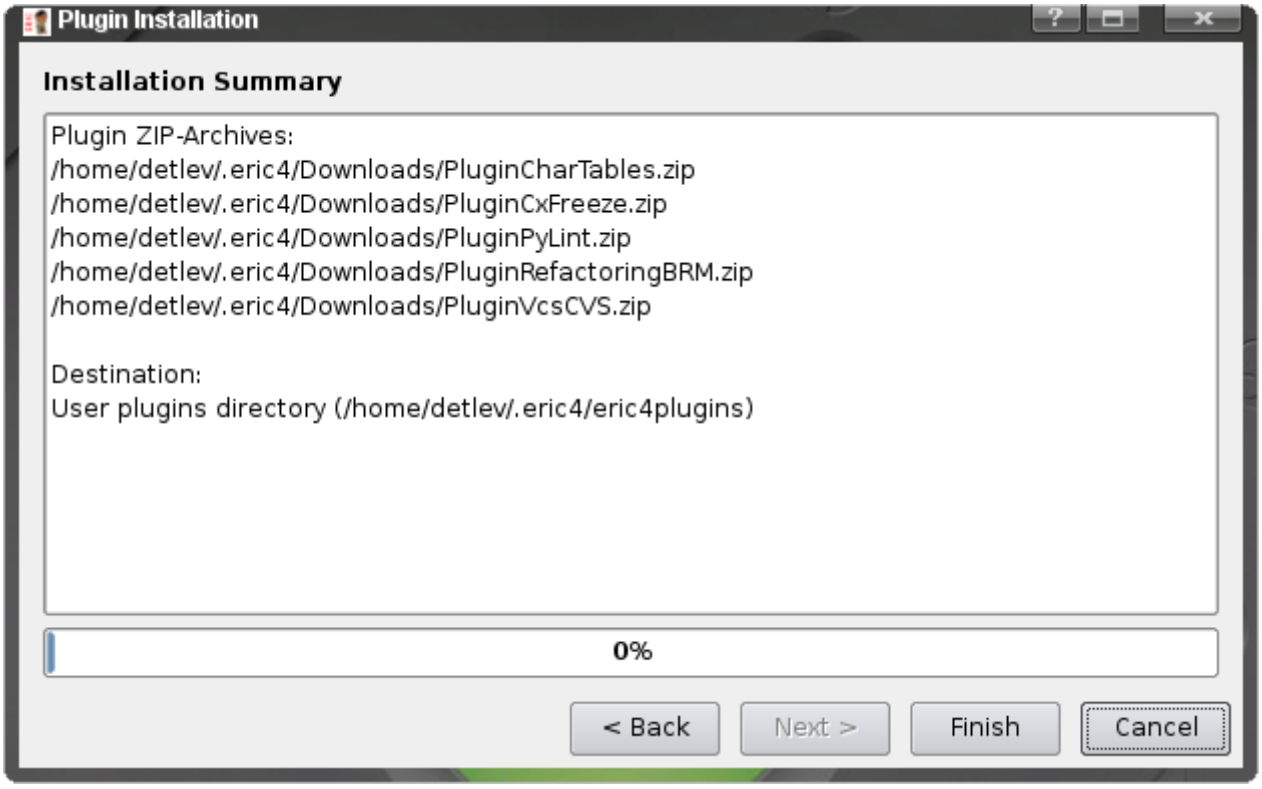

Figure 9: Plugins Installation dialog, step 3

The installation progress is show on the very same page. During installation the plugin archives is checked for various conditions. If the installer recognizes a problem, a message is shown and the installation for this plugin 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 plugin archive is shown by the progress bar.

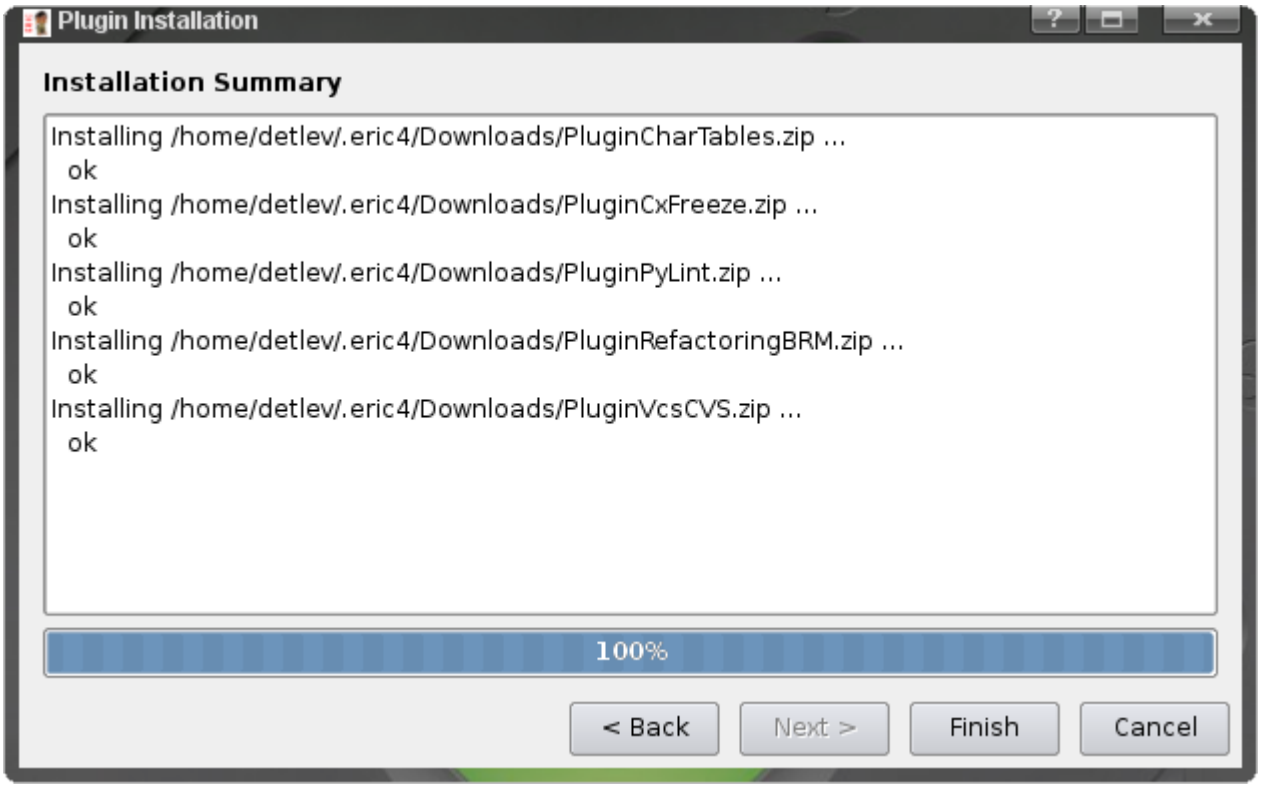

Figure 10: Plugins Installation dialog, step 4

Once the installation succeeds, a success message is shown.

Figure 11: Plugins Installation dialog, step 5

If plugins are installed from within eric4 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 Plugins

Plugins may be uninstalled from within eric4 using the “Uninstall Plugin...” menu, via the `eric4-pluginuninstall.py` script or via the eric4 tray menu. This displays the “Plugin Uninstallation” dialog, which contains two selection list. The top list is used to select the plugin directory. If the user has write access in the global plugins directory, the global and user plugins directory are presented. If not, only the user plugins directory may be selected. The second list shows the plugins installed in the selected plugins directory. Pressing the “OK” button starts the uninstallation process.

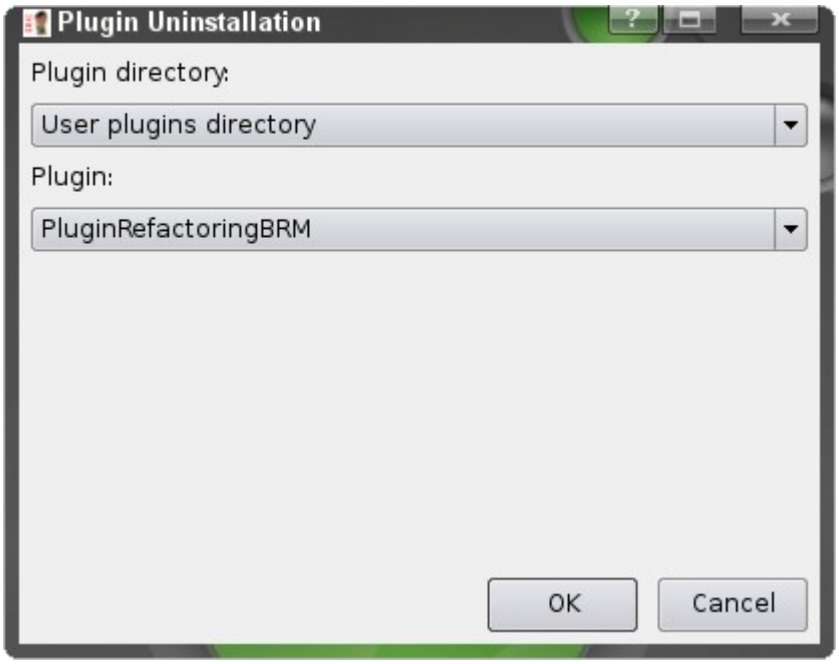

Figure 12: Plugin Uninstallation dialog, step 1

Figure 13: Plugin Uninstallation dialog, step 2

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

3.5 The Plugins repository

Eric4 has a repository, that contains all official plugins. The plugin repository dialog may be used to show this list and download selected plugins.

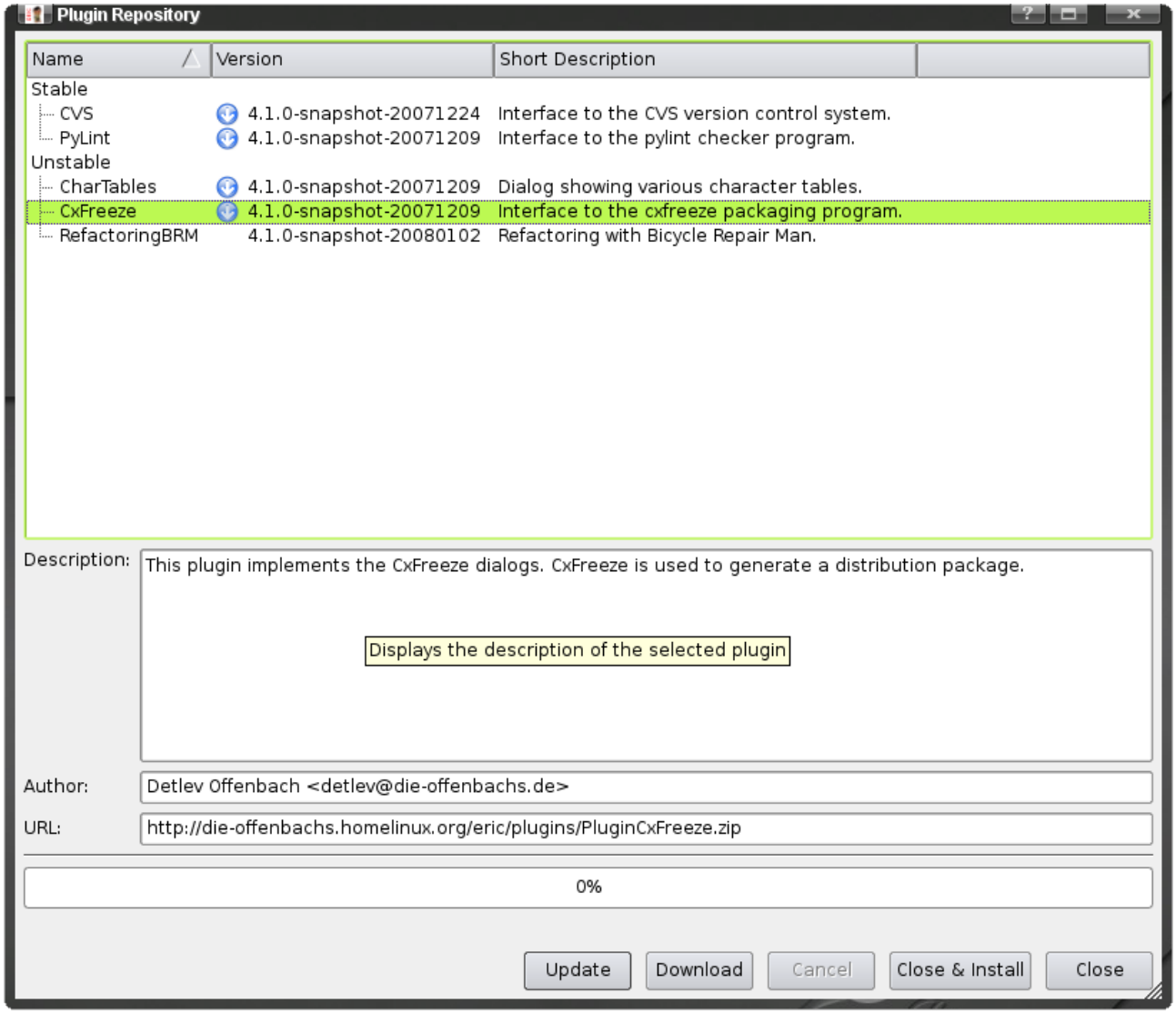

Figure 14: Plugin Repository dialog

The upper part of the dialog shows a list of available plugins. This info is read from a file stored in the eric4 user space. Using the Update button, this file can be updated from the Internet. The plugins are grouped by their development status. An icon next to the version entry indicates, whether this plugin 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 plugin, some detailed description and the author of the plugin. Pressing the Download button gets the selected plugins from the presented URL and stores them in the users plugin download area, which may be configured on the Plugins 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 plugin installation dialog (s. chapter 3.3).

4 Eric4 for plugin developers

This chapter contains a description of functions, that support plugin development with eric4. Eric4 plugin projects must have the project type “Eric4 Plugin”. The project's main

script must be the plugin main module. These project entries activate the built-in plugin development support. These are functions for the creation of plugin archives and special debugging support. An example of the project properties is shown in the following figure.

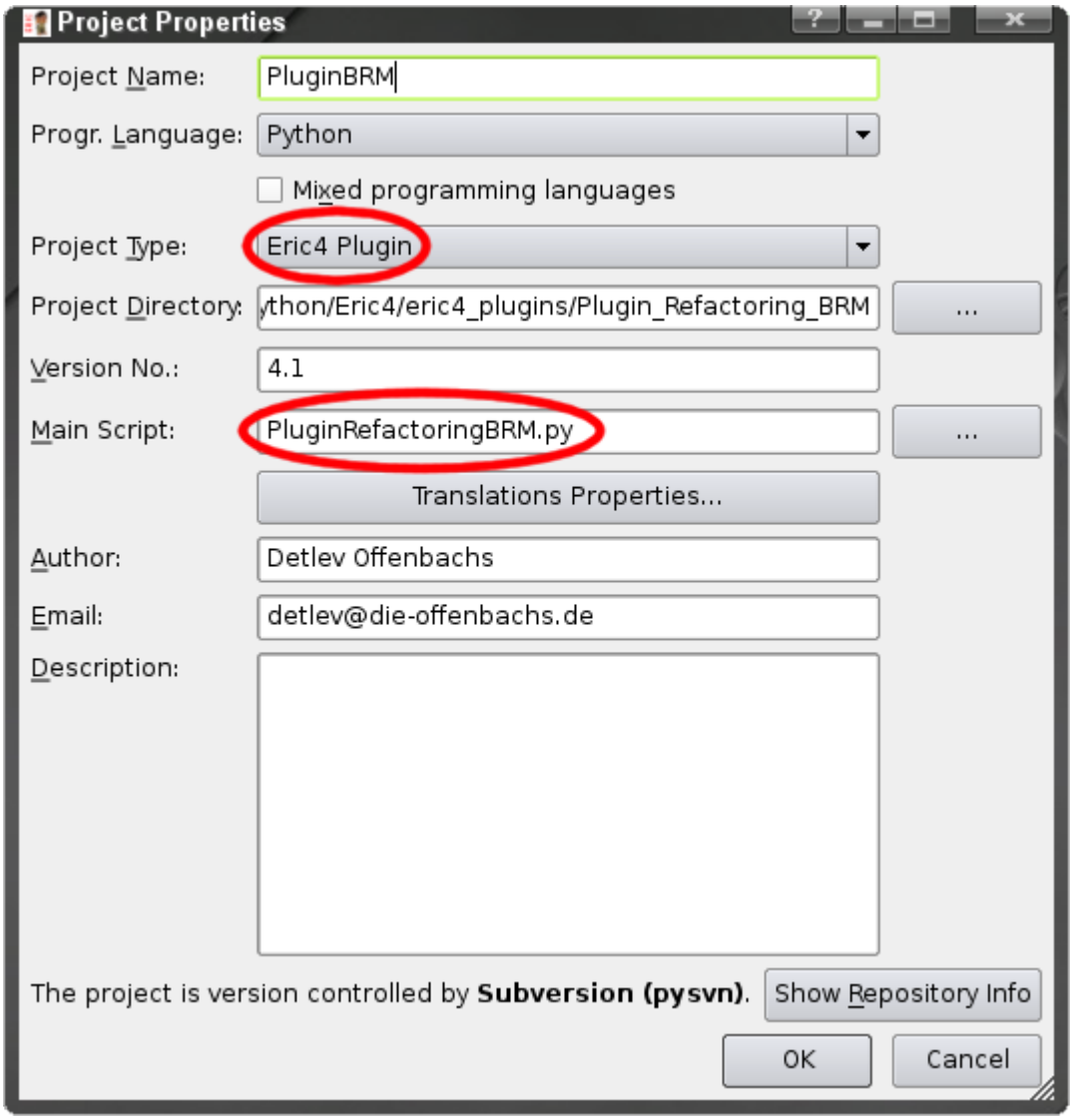

Figure 15: Plugin specific project properties

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

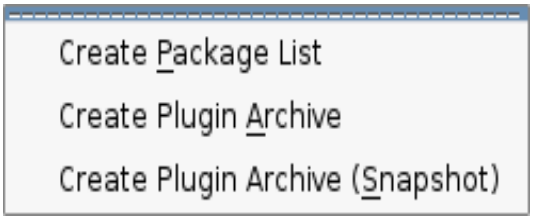

Figure 16: Packagers submenu

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 plugin archive. After the `PKGLIST` file has been created, it is automatically loaded into a new editor. The plugin author should modify this list and shorten it to just include the files required by the plugin at runtime. The following listing gives an example.

```
PluginRefactoringBRM.py
RefactoringBRM/ConfigurationPage/RefactoringBRMPage.py
RefactoringBRM/ConfigurationPage/Ui_RefactoringBRMPage.py
RefactoringBRM/ConfigurationPage/__init__.py
RefactoringBRM/ConfigurationPage/preferences-refactoring.png
RefactoringBRM/MatchesDialog.py
RefactoringBRM/Refactoring.py
RefactoringBRM/Ui_MatchesDialog.py
RefactoringBRM/__init__.py
RefactoringBRM/brm/__init__.py
RefactoringBRM/brm/bike/__init__.py
RefactoringBRM/brm/bike/bikefacade.py
RefactoringBRM/brm/bike/globals.py
RefactoringBRM/brm/bike/log.py
RefactoringBRM/brm/bike/logging.py
RefactoringBRM/brm/bike/parsing/__init__.py
RefactoringBRM/brm/bike/parsing/constants.py
RefactoringBRM/brm/bike/parsing/fastparser.py
RefactoringBRM/brm/bike/parsing/fastparserast.py
RefactoringBRM/brm/bike/parsing/load.py
RefactoringBRM/brm/bike/parsing/newstuff.py
RefactoringBRM/brm/bike/parsing/parserutils.py
RefactoringBRM/brm/bike/parsing/pathutils.py
RefactoringBRM/brm/bike/parsing/utils.py
RefactoringBRM/brm/bike/parsing/visitor.py
RefactoringBRM/brm/bike/query/__init__.py
RefactoringBRM/brm/bike/query/common.py
RefactoringBRM/brm/bike/query/findDefinition.py
RefactoringBRM/brm/bike/query/findReferences.py
RefactoringBRM/brm/bike/query/getAllRelatedClasses.py
RefactoringBRM/brm/bike/query/getReferencesToModule.py
RefactoringBRM/brm/bike/query/getTypeOf.py
RefactoringBRM/brm/bike/query/relationships.py
RefactoringBRM/brm/bike/refactor/__init__.py
RefactoringBRM/brm/bike/refactor/extractMethod.py
RefactoringBRM/brm/bike/refactor/extractVariable.py
RefactoringBRM/brm/bike/refactor/inlineVariable.py
RefactoringBRM/brm/bike/refactor/moveToModule.py
RefactoringBRM/brm/bike/refactor/rename.py
RefactoringBRM/brm/bike/refactor/utils.py
RefactoringBRM/brm/bike/transformer/WordRewriter.py
RefactoringBRM/brm/bike/transformer/__init__.py
RefactoringBRM/brm/bike/transformer/save.py
RefactoringBRM/brm/bike/transformer/undo.py
RefactoringBRM/i18n/brm_cs_CZ.qm
RefactoringBRM/i18n/brm_de.qm
RefactoringBRM/i18n/brm_fr.qm
RefactoringBRM/i18n/brm_ru.qm
```

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 plugin archive. This archive has the same name as the plugin module and is stored at the same place. The menu entry " Create Plugin Archive (Snapshot)" is used to create a snapshot release of the plugin. This command modifies the version entry of the plugin module (see below) by appending a snapshot indicator consisting of "-snapshot-" followed by the date like "20071206".

In order to debug a plugin under development, eric4 has the command line switch "--plugin=<plugin module filename>". That switch is used internally, if the project is of type "Eric4 Plugin".

5 Anatomy of a plugin

This chapter describes the anatomy of a plugin in order to be compatible with eric4.

5.1 Plugin structure

An eric4 plugin consists of the plugin module file and optionally of one plugin package directory. The plugin module file must have a filename, that starts with Plugin and ends with .py, e.g. PluginRefactoringBRM.py. The plugin 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 plugin module header).

5.2 Plugin header

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

```
name = "Refactoring BRM Plugin"
author = "Detlev Offenbach <detlev@die-offenbachs.de>"
autoactivate = True
deactivateable = True
version = "4.1.0"
className = "RefactoringBRMPlugin"
packageName = "RefactoringBRM"
shortDescription = "Refactoring with Bicycle Repair Man."
longDescription = """This plugin implements refactoring functionality"""\
    """ using the Bicycle Repair Man package."""

error = QString("")
```

Listing 2: Plugin header

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

- name
  This attribute should contain a short descriptive name of the plugin.
  Type: string

- `author`
  This attribute should be given the name and the email address of the plugin author.
  Type: string
- `autoactivate`
  This attribute determines, whether the plugin may be activated automatically upon startup of eric4. If this attribute is False, the plugin is activated depending on some configuration settings.
  Type: bool
- `deactivateable`
  This attribute determines, whether the plugin 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 plugin. This class must be contained in the plugin module file.
  Type: string
- `packageName`
  This names the package directory, that contains the rest of the plugin files. If the plugin is of the simple type (i.e. all logic is contained in the plugin 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 plugin and is used in the plugin info dialog.
  Type: string
- `longDescription`
  This attribute should contain a more verbose description of the plugin. It is shown in the plugin details dialog.
  Type: string
- `error`
  This attribute should hold an error message, if there was a problem, or an empty string, if everything works fine.
  Type: QString

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

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

Listing 3: Additional header for on-demand plugins

- `pluginType`
  This attribute must contain the plugin type. Currently eric4 recognizes the values “viewmanager” and “version_control”.
  Type: string

- `pluginTypename`
  This attribute must containe the plugin type name. This is used to differentiate the plugin within the group of plugins of the same plugin type.
  Type: string

Plugin modules may define additional optional attributes. Optional attributes recognized by eric4 are as follows.

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

If either the version or the className attribute is missing, the plugin 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 plugin will be loaded but not activated. If the packageName attribute is missing, the plugin installation will be refused by eric4.

5.3 Plugin module functions

Plugin modules may define the following module level functions recognized by the eric4 plugin manager.

- `moduleSetup()`
- `prepareUninstall()`
- `getConfigData()`
- `previewPix()`
- `exeDisplayData()`

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

5.3.1 moduleSetup()

This function may be defined for on-demand plugins (i.e. those with autoactivate being False). It may be used to perform some module level setup. E.g. the CVS plugin 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()

This function is called by the plugin uninstaller just prior to uninstallation of the plugin. That

is the right place for cleanup code, which removes entries in the settings object or removes plugin specific configuration files.

```
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

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 plugins. It must return a dictionary with globally unique keys (e.g. created using the plugin 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 plugin 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 plugin. This is just called for viewmanager plugins (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: Qstring
- `text`
  The entry text to be shown.
  Type: string or Qstring
- `version`
  The version text to be shown.
  Type: string or QString

```
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

5.4 Plugin object methods

The plugin 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 plugin 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 plugin object. E.g. this would be the right place to load a translation file for the plugin (s. Listing 13) 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 10: Example for the __init__(self, ui) method

5.4.2 activate(self)

This method is called by the plugin manager to activate the plugin object. It must return a tuple giving a reference to the object implementing the plugin logic (for on-demand plugins) or None and a flag indicating the activation status. This method should contain all the logic, that is needed to get the plugin fully operational (e.g. connect to some signals provided by eric4).

```
    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()
        e4App().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(\
            e4App().getObject("ViewManager").getOpenEditorsCount())

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

        self.connect(e4App().getObject("ViewManager"),
                     SIGNAL('lastEditorClosed'),
                     self.__lastEditorClosed)
        self.connect(e4App().getObject("ViewManager"),
                     SIGNAL("editorOpenedEd"),
                     self.__editorOpened)
        self.connect(e4App().getObject("ViewManager"),
                     SIGNAL("editorClosedEd"),
                     self.__editorClosed)

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

        self.connect(e4App().getObject("Project"), SIGNAL('projectOpened'),
                     self.__object.projectOpened)
        self.connect(e4App().getObject("Project"), SIGNAL('projectClosed'),
                     self.__object.projectClosed)
        self.connect(e4App().getObject("Project"), SIGNAL('newProject'),
                     self.__object.projectOpened)

        for editor in e4App().getObject("ViewManager").getOpenEditors():
            self.__editorOpened(editor)

        return None, True
```

Listing 11: Example for the activate(self) method

5.4.3 deactivate(self)

This method is called by the plugin manager to deactivate the plugin 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.

```
    def deactivate(self):
        """
        Public method to deactivate this plugin.
        """
        e4App().unregisterPluginObject("RefactoringBRM")

        self.disconnect(e4App().getObject("ViewManager"),
                        SIGNAL('lastEditorClosed'),
                        self.__lastEditorClosed)
        self.disconnect(e4App().getObject("ViewManager"),
                        SIGNAL("editorOpenedEd"),
                        self.__editorOpened)
        self.disconnect(e4App().getObject("ViewManager"),
                        SIGNAL("editorClosedEd"),
                        self.__editorClosed)

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

        self.disconnect(e4App().getObject("Project"), SIGNAL('projectOpened'),
                        self.__object.projectOpened)
        self.disconnect(e4App().getObject("Project"), SIGNAL('projectClosed'),
                        self.__object.projectClosed)
        self.disconnect(e4App().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 12: Example for the deactivate(self) method

5.4.4 __loadTranslator(self)

The constructor example shown in Listing 10 loads a plugin 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
            e4App().installTranslator(self.__translator)
        else:
            print "Warning: translation file '%s' could not be loaded." \
                % translation
            print "Using default."
```

Listing 13: Example for the __loadTranslator(self) method

6 Eric4 functions available for plugin development

This chapter describes some functionality, that is provided by eric4 and may be of some value for plugin development. For a complete eric4 API description please see the documentation, that is delivered as part of eric4.

6.1 The eric4 object registry

Eric4 contains an object registry, that can be used to get references to some of eric4's building blocks. Objects available through the registry are

- `UserInterface`
  This is eric4 main window object.
- `DebugUI`
  This is the object, that is responsible for all debugger related user interface elements.
- `DebugServer`
  This is the interface to the debugger backend.
- `ViewManager`
  This is the object, that is responsible for managing all editor windows as well as all editing related actions, menus and toolbars.
- `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)

- `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.
- `Shell`
  This is the object, that implements the interactive shell (Python or Ruby).
- `PluginManager`
  This is the object responsible for managing all plugins.

Eric4's object registry is used as shown in this example.

```
from KdeQt.KQApplication import e4App

e4App().getObject("Project")
```

Listing 14: 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 plugin 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 plugin object. If the named object has not been registered, nothing happens.
- `getPluginObject(name)`
  This method returns a reference to the named plugin 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 plugin objects. Each list element is a tuple giving the name of the plugin object and the reference.

6.2 The action registries

Actions of type `E4Action` may be registered with the `Project` or the `UserInterface` object. In order for this, these objects provide the methods

- `Project.addE4Actions(actions)`
  This method registers the given list of E4Action with the `Project` actions.
- `UserInterface.addE4Actions(actions, type)`
  This method registers the given list of E4Actions with the `UserInterface` actions

of the given type. The `type` parameter may be “ui” or “wizards”

6.3 The getMenu() methods

In order to add actions to menus, the main eric4 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.
  - `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.
- 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.
    - `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.

6.4 Methods of the PluginManager object

The `PluginManager` object provides some methods, that might be interesting for plugin development.

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

6.5 Methods of the UserInterface object

The `UserInterface` object provides some methods, that might be interesting for plugin 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(self, 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 QString 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(self, name)`
  This method is used to unregister a toolbar. `name` is the name of the toolbar as a Python string.
- `getToolbar(self, 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.
- `getLocale(self)`
  This method is used to retrieve the application locale as a Python string.

6.6 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 plugins.
  - PACKAGERSPARMS
    Used by packager plugins.

- DOCUMENTATIONPARMS
  Used by documentation plugins.
- OTHERTOOLSPARMS
  Used by plugins not fitting the other categories.

The `key` parameter gives the key of the data entry to get and is determined by the plugin. 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 plugins.
  - PACKAGERSPARMS
    Used by packager plugins.
  - DOCUMENTATIONPARMS
    Used by documentation plugins.
  - OTHERTOOLSPARMS
    Used by plugins not fitting the other categories.

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

6.7 Methods of the ProjectBrowser object

The `ProjectBrowser` object provides some methods, that might be interesting for plugin 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`
  - `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 others browser

6.8 Signals

This chapter lists some Python type signals emitted by various eric4 objects, that may be interesting for plugin 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
        - `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
        - `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
- 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.
- 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.

7 Special plugin types

This chapter describes some plugins, that have special requirements.

7.1 VCS plugins

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

```
pluginType = "version_control"
```

VCS plugins 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 (QString) 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 (QString)
    """
    global displayString, pluginTypename
    data = {}
    data[".svn"] = (pluginTypename, displayString)
    data["_svn"] = (pluginTypename, displayString)
    return data
```

Listing 15: Example of the getVcsSystemIndicator() function

7.2 ViewManager plugins

ViewManager plugins 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 plugins must define their type by defining the module attribute `pluginType` like

```
pluginType = "viewmanager"
```

The plugin module must implement the `previewPix()` method as described above.