Plugin development

Plugin specification

In order to qualify as a valid plugin, plugin implementations must comply with a few basic rules. The plugin must be provided in the form of a Python *module* or *package*. The module must be dropped in the plugins directory. If a plugin is implemented as a Python module (e.g. applicable for very simple plugins), it must following this naming convention:

- “*plugin.py”
- “plugin*.py”

On the other hand, if a plugin is implemented as Python package, the package directory must follow this naming convention:

- “*plugin”
- “plugin*”

Note: these conventions are case insensitive.

The module or the *_init_*.py module in the case of a package, must implement the method getPluginInstance, which returns an instance of the plugin.

Python doesn’t support natively the concept of *interface*, like Java or C#. Instead, each plugin must provide a class that implements the following methods:

- **register:** defines which *profile*, *command* and *target* it registers for. A helper class is available to the plugin for implementing this easily. This is called once during the installation of the plugin so that the plugin manager is aware of which profile the plugin corresponds to, in which target to execute it and make sure that all the files are present.
- **contribute:** defines which plugin it contributes to. A helper class is available to the plugin for implementing this easily.
- **init:** initialise the plugin and its environment
- **execute:** execute the plugin
- **stopExecute:** stop the execution of the plugin
- **publish:** harvest and publish to a known location the results of the execution
- **finalise:** executed once at the end of the processing, gives a chance to the plugin to clean its resources and perform any final actions, if any.

The first two methods are called as part of the registration process, in two distinct iterations, which guarantees that all plugins have being registered before contributions can be made.

During each target execution (e.g. *init*, *checkstyle*, *compile*, etc in the case of the *BuildCommand*), the following methods are called for the plugins matching the current processing state: init, execute, stopExecute and publish. Each plugin is expected to call its contributors in each method. A helper class is provided by the plugin framework to easily execute the contributors.

The plugins are also provided with an instance of the PropertyManager class (providing full access to the property space of the current module being executed) and the BuildStatus class (providing access to the build and test status XML document).

The plugin also needs a setup.py script that is used to copy files to a local directory when the plugin is being installed.

Plugin structure:

Properties used by plugins:

General properties:

notest	disables all plugins registered at TEST target
no<plugin name>	disable this plugin.
src.location	sets the location of source files (any language).

Language specific properties:

java.src.location	sets the location of java source files.
java.class.location	sets the location of java class files.
java.jar.location	sets the location of java jar files.
java.lib.location	sets the location of java lib files.

Plugin specific properties:

Plugin specific properties can be used to set a threshold or allow for any other customization that may be necessary for the plugin. These properties should start with <plugin.name>. For example findbugs.class.location or findbugs.failure.threshold.

Methods called by plugin framework:

Register

```
def register(self,pluginManager):
```

Is called once to register the plugin.

It sets where the plugin should be executed, normally during 'pretest'.

Here the plugin also verifies that the plugin is correctly installed, i.e. it has the files it needs. If it does not find any missing files it calls

```
self.pluginManager.register(self.__module__, definition)
```

Execute

```
{def
execute(self,profile=None,command=None,target=None,targetString=None,propertyManager=None,buildStatus=Nor
```

Is called for each component.

Runs the metric application, for example with an apache-ant command.

Publish

```
def
publish(self,profile=None,command=None,target=None,targetString=None,propertyManager=None,buildStatus=Nor
```

Is called for each component.

Publishes metric data for each component and links to the detailed html/xml pages created by the metrics application.

Creates the module metrics:

```
metrics = buildStatus.CreateMetrics('Plugin name')
metrics.detailshtmllink = #path to component specific .html page (created by metrics application, not
the plugin).
metrics.detailsxmllink = #path to component specific .xml page (created by metrics application, not
the plugin)
metrics.values = #array with all the values
metrics.value = #overall value
metrics.type = #integer, float, percent, etc.
metrics.unit = #bugs, violations, etc.

buildStatus.setModuleMetrics(propertyManager['moduleName'],propertyManager['name'],propertyManager['proje
```

Finalise

```
def
finalise(self,profile=None,command=None,target=None,targetString=None,propertyManager=None,buildStatus=No
```

Is called once to finalise the plugin.

First it creates the plugin specific report page from a template file. It replaces lines in the template .html file with html created by the plugin.

Then it creates the summary page located in the /reports/index.html page in the report where it shows the metric data as an average of each component's data.

```
metrics = buildStatus.CreateMetrics('Plugin name')
metrics.detailshtmllink = 'plugin name/index.html' #path to plugin specific .html page.
metrics.values = #array with all the values
metrics.value = #overall value
metrics.type = #integer, float, percent, etc.
metrics.unit = #bugs, violations, etc.
buildStatus.setOverallMetrics(metrics)
```

Other resources:

ETICS: Plugin framework (look also at the subsections describing four of the plugins).