Plugins

This section provides an overview of the client-side plugin framework users can use to extend their build and test procedures.

Updated information about plugins can be found in the twiki page:
https://twiki.cern.ch/twiki/bin/view/EMI/EticsPluginsFramework

Overview

The plugin framework provides the ability for common actions to be performed during build and test execution, without putting the burden on the user for selecting which build actions should take place and when. In other words, the framework provides a clean separation between specific build and test execution, analysis and metrics collection. This section provides a high-level description of the plugin framework architecture and guiding principals.

The reason for exposing the framework to users is that the framework also provides the ability for the ETICS client to be extended dynamically, without having to modify the core client functionality. Further, ETICS users can leverage the extensibility, such that they can better reuse investments in home grown and/or commercial tools and techniques valuable to them during build and test.

In order to even better leverage the work performed by existing plugins, the plugin framework also allow plugins to *contribute* to other plugins, without having the plugin being contributed to needing intimate knowledge from the contributing plugins. For example, a plugin calculating code coverage during execution can contribute to a testing plugin, without the testing plugin requiring intimate knowledge about how code coverage is calculated.

The goal of the plugin framework can be summarised as follows:

- Provide an extendible framework for common actions to be performed during build and test execution
- Provide natural separation between specific build and test execution, analysis and metrics collection
- Provide open interface for users to register their own plugins to be executed during build and test
- Provide raw data for build and test and repository services

To guide the definition of the framework, here are examples of actions that could map to plugins, to be executed by the framework:

- **Static analysis**
  - Single line of code count (SLOC)
  - Cyclomatic complexity
  - Depth of inheritance
  - IPv6 compliance
  - Check style
- **Dynamic analysis**
  - Code coverage
  - Memory leaks
- **Test execution**
  - Java: JUnit
  - C++: CPPUNIT (CPPTEST)
  - Python: PyUnit
  - Script/Executable

The guiding principal in the examples above is that none of these actions related to the specifics of the modules on which these actions are performed.

The following describes the high-level framework architecture.

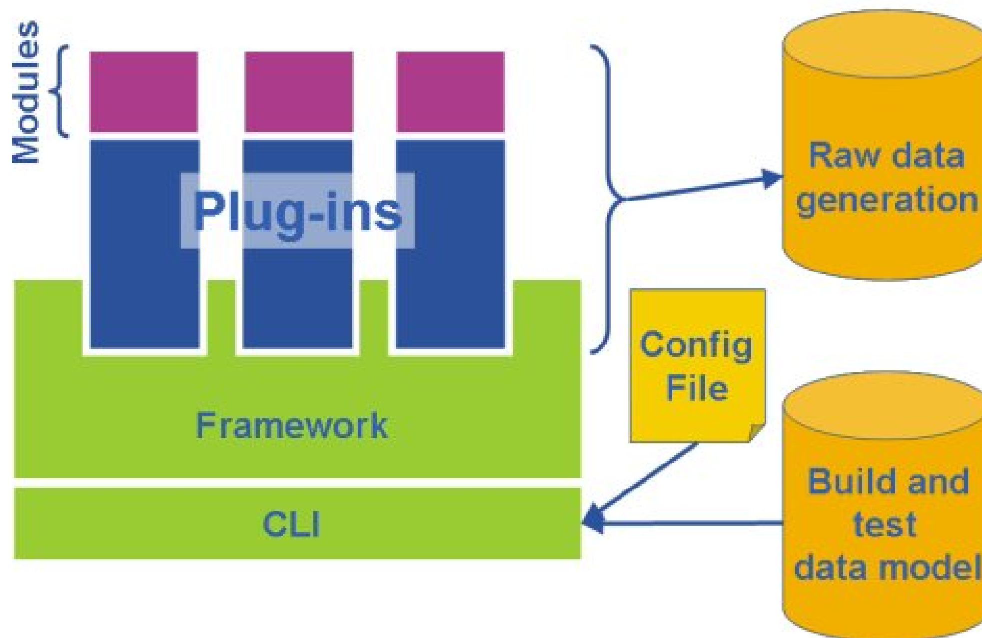

Figure: Plugin Framework High-level architecture

The framework is implemented as an extension to the ETICS command-line client. In this respect the framework is totally transparent to the user. The framework also leverages the property processing already performed by the client. Raw data can be generated and should be placed in the *reports* directory in order for these generated outputs being available after the remote execution.

When a command is executed, the plugin framework first loads and registers all plugins. The registration process requires for each plugin to provide information regarding the *profile* for which they should be activated, as well as which ETICS target of which command they apply to (i.e. *VCS Command*, *Build Command* and/or *Test Command*). In other words, the following combination dictates the execution of each plugin: profile+command+target.

To add an extra level of flexibility, wildcards can be used by plugins during the registration. For example, the generic *SystemCallPlugin* responsible for executing as a system call the string value of each command might register for: profiles="**", commands="**" and target="*".

The next section provides an overview of the specification of plugins.

Plugin installation:

1. The ETICS client determines which plugins to install based on the plugin-list.xml. By deafult it looks in the **registered** repository, but a volatile repository can be specified while specifing the plugin-list xml file. More on specifing the plugin-list xml in the ?Plugin testing section.
2. Downloads all the plugin source packages.
3. Extracts the pacakges.
4. Installs the files using the include setup.py with the install commands from the plugin-list.xml.
5. Registeres the plugin with a profile and a target.
6. The plugin will now execute in the given target if the profile is triggered by one of the three options mentioned in the "How to run a plugin" section under ?Plugin testing.

Other resources:

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