Software Engineering Group Project
Group2
Earth Plugin Framework
Semester 2 2008

❖ Version History

Version number	Date	Author(s)	Changes
1.0	10/9/2008	Mohammad	First draft

Objective

On this document, we will describe the plugin architecture used in the Earth project. Actually, we split earth plugins into two categories: daemon plugins and GUI plugins. On this paper, we will focus on the first category. The same structure could be used for the GUI part but it needs further analysis to make sure that this design will work on the controllers and views.

Overall Structure

Visualizing the design will make it easier to explain. So, we will start by a figure which summarizes the overall design.

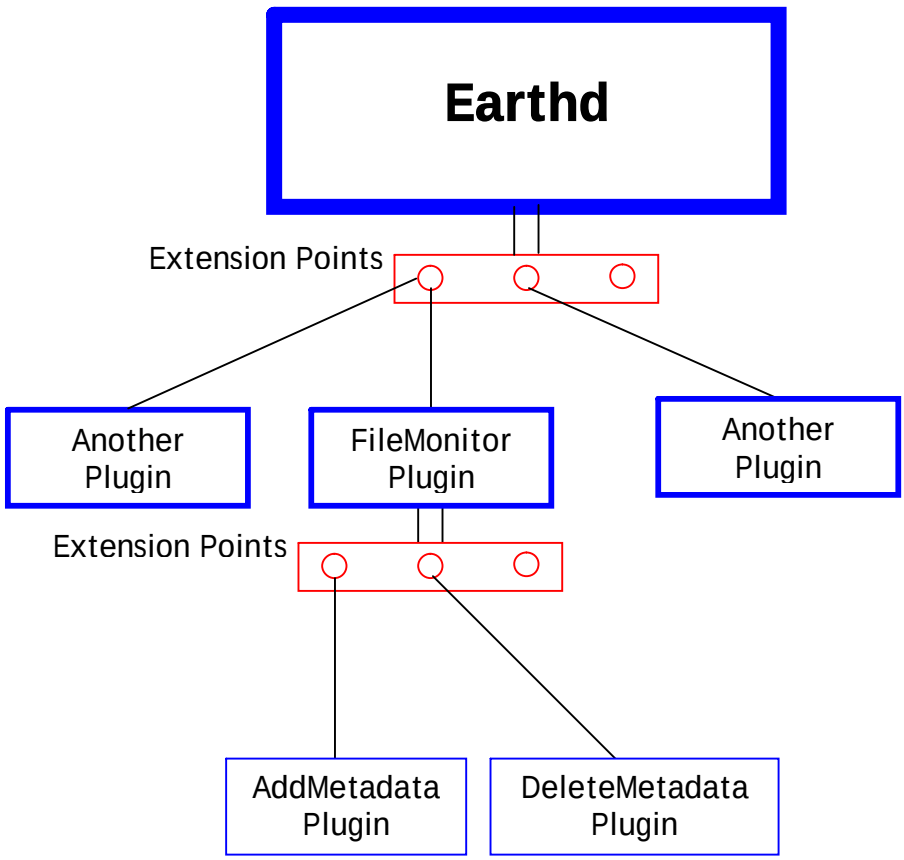

Figure 1 Earth daemon plugin framework

We can think of *Earthd* (Earth Daemon) as being something like USB hub. There is a standard plugin specification which defines how the hub operates. Using these standards you can plug any devise you want into the hub. And like USB hub, adding/removing plugins is dynamic; it happens in the run-time.

As we can see in figure 1, *Earthd* has multiple *extension points* (USB ports). These extension points are used to extend the core functionality of *Earthd*. Each plugin will be linked to one specific extension point to extend *Earthd*. Each extension point is unique and could be used by many plugins (more about extension points in the next section).

In addition, notice that plugins themselves can be extended. If you want your plugin to be extended, create extension points inside it and register these extension points in the database.

Extension Points

Extension points are analogues to USB ports in our example about USB hub. The only difference is that each extension point is unique. Unlike USB ports, extension points provide different information to their plugins. so, when you create a plugin, it must be specified to a specific extension point.

All extension points must have records in the database table (extension_points).

Each extension point will have the following characteristics:

1. It is unique. It should have: *name* (string) which reflects its position in the host file (plugin), *description* which describes what it extends and what arguments does it provide, *host-plugin* which is the name of Class which hosts this extension point. When this extension point is inserted it will have a unique *ID*.
2. Each extension point could provide some information (arguments) which can be used by the plugins. These arguments are like messages (parameters) between host-plugin and extended plugin.

This is an example of an extension point inside Earthd:

```
extension_point("main_loop", "Earthd", :logger => @logger, …)
```

- "main_loop" === is the extension point *name*. It should be something meaningful. You can understand immediately that this extension point is in Earthd main loop.
- "Earthd" === is the host-plugin class name. (Since each plugin should have a method "plugin_name", it is better to use it)

- :logger => @logger === after you give name and host-plugin name, you can send any number of arguments to plugins. They should be in a Hash form: (:argument_name => argument_value). In the plugin side, you can retrieve parameters by calling method get_param(:argument_name). This method is in the EarthPlugin class so it is available for all plugins which should inherit from the EarthPlugin class.

Here is another example of an extension point inside FileMonitor:

```
extension_point("file_added", self.plugin_name, :file => new_file)
```

- "file_added" === name of the extension point. You can understand that this point extend the core functionality of FileMonitor which is Adding Files to the database. So, we can create plugins that save metadata for that file, save file type or do anything with that file.
- self.plugin_name === host-plugin name
- :file => new_file === parameters which can be used by the plugins which are linked to this extension point.

Where do I put my extension points (in the host plugin)?

This is a design issue. For example, inside FileMonitor, you need to look for the locations where we can extend the core functionality. Some of the core functionalities of FileMonitor: adding files to the database, removing files, adding directories, removing directories. So, we look where does FileMonitor add new files to the database and create an extension point there so we can do other stuff with that file.

Since FileMonitor has been created before the idea of extension points, we might find it hard to find the right spots for our extension points. But from know on, when you create your a plugin, you should consider that it could be extended. It is NOT necessary to have extension points inside your plugin; it depends on your design.

Earth API

Extension points gives us a framework to plug new stuff to our earth existing system dynamically (at run-time). What about the plugin implementation? The plugins will add more functionality to the existing system. They need some how to interact with Earth models.

It is not a good practice to let the plugins interact directly with Earth modules. This will make any changes in Earth implementation affect all the plugins. Also, any one wants to create new plugin has to know about Earth implementation. This is not practical.

For that, we should provide APIs to be used by plugins. APIs will be the bridge between Earth and the plugins. Plugins' developers will not need to know anything about the internal implementation of earth except extension points information which should be available in the database.

Currently, there is no APIs. Since Earth is not documented well, we can not decide the needed APIs. Our plan is to create the APIs in parallel with creating the plugins. As long as we implement more plugins, we will have better idea about the common functionality needed to be provided in the API. Figure 2 demonstrates the use of the APIs.

How are the plugins loaded?

When a plugin is installed, a new record in the *plugins_descriptors_table* will be added. The new records will contain:

- *plugin name*: name of the installed plugin class
- *plugin code*: all the plugin code because we will load the plugins from the database.
- *signature*: for verification purposes
- *method*: name of the (method) which needs to be called to add the needed functionality. We can think of it as the main method in the plugin class. If the method column is null, then the plugin do its job inside the initialization method.
- extension_point_id: foreign key to the *extension_points* table

Each plugin will be linked to one specific extension point. That's why there is a foreign key column named: *extension_point_id*.

Having the plugin installed in the database, when an extension point encountered, the framework will look for all the plugins which are associated with this point. It will use the plugin_manager to load each plugin. Then, if there is a method in the method column, it will be called.

Simple Plugin Example

In this section I will give you very simple example of a plugin which extends the "main_loop" extension point in the Earthd. To remind you, this the extension point inside Earthd:

```
extension_point("main_loop", "Earthd", :logger => @logger, …)
```

Now, let's see the example plugin. I will make it self explanatory by putting enough comments.

```
class ExamplePlugin < EarthPlugin
      # defines your plugin name
      slef.plugin_name
      #the name should start with Earth followed by the class name
            "EarthExamplePlugin"
      end
      #define plugin version
      self.plugin_version
            1
      end
      #when ever your plugin is loaded, this method will be called
      #so, you make your plugin do its job when it is loaded
immediately without calling any other methods
      def initialize
            #bring the parameters you want to use
            @my_logger = get_param(:logger)
            #Here, what this plugin does. It just adds a debug message
            to the logs but it used Earthd logger.
            @my_logger.debug("Hello, I am an example plugin!!!")
      end
end
```

Figure 2 Example plugin

On the previous example, the plugin does its job when instantiated. So, in the plugin_descriptor table, we do not need to have anything in the (method) column because there are no methods to be called.

Figure 3 shows a possible template to be used for any plugin. You need to change the code in blue.

```
class "plugin_name" < EarthPlugin
     # defines your plugin name
     slef.plugin_name
     #the name should start with Earth followed by the class name
          "plugin_name"
     end
     #define plugin version
     self.plugin_version
          "integer number"
     end
     #when ever your plugin is loaded, this method will be called
     def initialize
          #bring the parameters you want to use (optional)
          local_variable = get_param(:argument name) #(optional)

          #Here, you can do what ever you want the plugin to do
          ...................
          ...................
     end

     # Local methods
end
```

About the (method) in the plugins_descriptors table

If you use a specific (method) to be called when your plugin is loaded, you need to consider one important thing. Your method parameters MUST have default values. Otherwise, it will not work. Why is that?

When you call your method, you expect the parameters from the host plugin. Unfortunately, exchanging parameters (arguments) between the host plugin and the extended plugin can not happen like any two classes. Remember, we are reading plugins code from database!! So, you need to use method *get_param()* to bring the parameters you need. Figure 4 shows two possible ways to create a plugin method which is registered in the *plugin_descriptors* table.

```
.................

def plugin_method(param1 = nil, param2 = [],,,)
      param1 = get_param(:param1_name)
      param2 = get_param(:param2_name)
      ....
end

.................
```

OR

```
.................
def plugin_method(p1 = get_param(:1_name), p2 =get_param(:p2_name),,,)

      ....
end

.................
```

Steps to create your first plugin

In this section, I will take you step by step to create your first plugin.

1- Find a suitable extension point for your plugin: Look in the *extension_points* table and understand the description of each extension point. You could look at the host plugin code to understand it more. That is why it is very important to write meaningful description about each extension point.
2- Create your plugin: you could use the template from this document. remember:
    a. Make your plugin self contained. Focus on one job.
    b. Make your plugin as small as possible.
    c. your plugin must inherit from *EarthPlugin*
    d. there must be two static methods: *self.plugin_name* and *self.plugin_version*
    e. Take care of parameters exchanges. Use the method get_param() to bring parameters from the host plugin
3- Sign your plugin: use the script *sign_plugin* to sign your plugin.
4- Install your plugin: use the script *install_plugin* to sign your plugin. Refer to "Plugin_System_Notes.pdf" document in: segp2/Subgroups/Group_3/Documents/Design/ if you face any problems signing or installing your plugin. (TODO this must be included in this document)
5- Temporary step: manually go to the plugin_descriptors table and update the extension_point_id column to the one you want. Also, you could update the (method) column if you need a specific method to be called. This should happen with the installation script. It is under development.
6- Test your plugin: after installing your plugin, it will be activated immediately. Check the logs. You may start testing your plugin by running it from the file system to make sure it is working fine.

GUI Plugins

???????