Plugin System Notes 2

Where does the API fit in the current plugin architecture?

As mentioned in Plugin_System_Notes.pdf, the current plugin architecture is only available for the daemon to load extra functions and features when it runs. The file_monitor.rb is the only "plugin" the daemon is loading for now.

So, the API comes in as a gateway between Earth and plugins. During semester 1, we directly changed the codes in Earth to incorporate our fixes. Now, we will have to introduce one extra step so in order to add a new feature, we will need to use the API layer instead.

Right, what's the deal with decomposing the method available in file_monitor.rb?

The issue with the `api_*.rb` files contained in the `lib/earth_plugins` directory are specific to file monitor. So, perhaps this is not the API that we want. The last thing we do not want is to have a set of APIs that are specific to respective tickets. What we want are API that allows developers directly use, inherit or instantiate for the purpose of their plugins. It is really up to what the developers on how they use the APIs.

So, if we use the current "APIs", we cannot even create a plugin, let alone use them. Yes, they are being used by file monitor, but what if I want to create plugin that improves the garbage clean up of file monitor, which the clean up method is already used in the file monitor API? Now, I cannot use it unless instantiate file monitor, which will create 2 copies of file monitor running concurrently.

Perhaps is there a trick to decomposition?

Well, there is no smoking gun for such an activity. One of the entry points is to try to generalize some supporting methods and make them as atomic as possible. Then, look at what file monitor can offer by applying these atomic methods.

An example will be generalizing the `Earth::*` models to allow file monitor not to use the same syntax everywhere. If that is successful, we are now on track as a lot of daemon tickets use that, and we can start converting simple tickets utilizing the first set of APIs.

Where should we store the API files and how are we going to categorize them?

For the time being, we can store them in separate `module...end` files. We will further brainstorm a way to store them in the database to be loaded remotely.

The following are the possible files and their descriptions:

- `lib/earth_api/eapi_utilities.rb` Stores all the commonly used methods and classes.
- `lib/earth_api/eapi_earth.rb` Stores all the Earth model accessors, classes and methods.
- `lib/earth_api/eapi_conventions.rb` Stores all the naming conventions stuffs.
- `lib/earth_api/eapi_tags.rb` Stores all the tagging stuffs.
- `lib/earth_api/eapi_xfile.rb` Stores all the file metadata accessor stuffs that is not generic to be stored `eapi_earth.rb`.
- `lib/earth_api/eapi_xdirectory.rb` Same like `xfile`, but for directories.
- `lib/earth_api/eapi_daemon.rb` Stores all the daemon accessor stuffs for plugins that is low-level enough to intercept the daemon's activities.

- `lib/earth_api/eapi_rsp_library.rb` Stores RSP specific APIs, which they would like to license them as open-source software.

Well, this is just a list of suggestions. We do not need to create them all, but I hope it will give you an idea what we are up against.

So how do we use them APIs?

I have submitted my "bogus" plugin into `pamalite/earth` repo, and my little experimental plugins into the Google Groups. You can have a look at how I use the `ETAPrinter` class, which is from the `eapi_utilities.rb` class. (Yes, a module is a class in Ruby. Don't ask me why.)

All these plugin stuffs is making the daemon nervous. What do I do?

Yup, the daemon can be considered out-of-date due to the hap-hazard-ness of RSP to get the plugin working. This is an obvious evidence that the plugin concept is not mature in Earth.

So, as Fil had pointed out, we will have to:

1. Get the current plugin system working. That means, load file monitor from the database.
2. Figure out a way to automatically run the plugins as it is loaded.

After the 2 tasks, we will have to:

1. Figure out how to handle dependencies. This is important as shown in my experiments, that a developer may want to take the short-cut to built a plugin base on another existing plugin. (Well, that's what I will do if I want to expand an existing plugin.)
2. Figure out store and load the APIs so that we can deploy them only once on a single machine. We can use what Mohammad said by letting the `earth_plugin` class do all the API management.

These are just some thoughts. Probably we can use it as a lead for future milestones.

So, what about the GUI plugin you showed?

My little experiment is also in the repo. You can have a look. The files highlighted were documented in the first notes. Look for the keyword "usages" and you will find what I did.

You probably might say that there are so many things to consider! Yup, that is how Rails work. (Who said Ruby on Rails is happiness for developers?)

Since your GUI plugin is still in its infancy, what can possibly be done to make it production worthy?

Well, my experiment is not meant as a production worthy product. It is meant for group inspiration and to get the ball rolling.

I was hoping to create a set of API dubbed the "Earth Toolkit" to let the developers to "draw" stuffs on the front-end. The toolkit interacts tightly with the GUI engine. I borrowed this idea from Mac OS X's Cocoa framework. (You can even picture it as the Microsoft Foundation Class (MFC).)

I know it is not easy to create the GUI engine, let alone the toolkit. There is always the notion of exploration and creating an experimental version that can keep us "busy" with RSP. `:-)`