PlugMan

A simple but powerful Ruby plugin framework.

Table of Contents

Overview

PlugMan is a simple and effective plugin architecture for building Ruby applications. It allows the developer to create programmes from highly reusable components, and allows programmes to become highly extensible by opening them up to external development opportunities.

PlugMan manages a repository of plugins.Plugins can be loaded at any time and can also be stopped and started at will.

There are two main mechanisms that are used in PlugMan's architecture:

1) Plugins as dependencies; and
2) Plugins as extensions

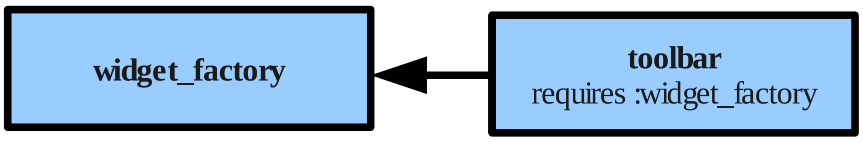
These are explained in the next section.

Plugin Types

Plugins as Dependencies

Suppose *plugin_a* requires some service of *plugin_b*. When plugin_a is defined, it has an explicit dependency requirement on *plugin_b* and won't be able to function without *plugin_b* being present. *plugin_b* does not require (or even know about) *plugin_a*.

An example of this may be that a *toolbar* plugin may require the services of a *widget_factory* plugin. The *widget_factory* plugin does not know anything about the toolbar plugin but the *toolbar* plugin will not function without *widget* factory.



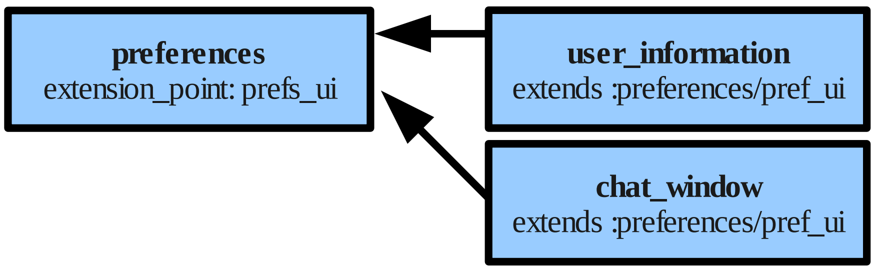
At runtime, *toolbar* will get a reference to *widget_factory* through PlugMan, and it will then execute the method on *widget_factory* that it requires.

This type of plugin use has the child plugin executing the parent plugin.

Plugins as Extensions

Suppose *plugin_x* defines an extension point *ext_x*. *plugin_y* and *plugin_z* are defined as extending *plugin_x*/*ext_x* and at runtime, plugin_x discovers *plugin_y* and *plugin_z*. plugin_x does not explicitly know about the extending plugins until they are discovered, and even then it only knows the interface method to call.

To illustrate, suppose we have a preferences screen in our application.Some parts of our application will need to present the user with configuration items. To do this, they extend an extension point defined by our *preferences* plugin. The following diagram shows the *preferences* plugin and a couple of extensions than need configuring: *user_information* and *chat_window* (perhaps this is for a chat client):


preferences
extension_point: prefs_ui
user_information
extends :preferences/pref_ui
chat_window
extends :preferences/pref_ui

At runtime, *preferences* will ask PlugMan for a list of plugins that are defined as extending its *prefs_ui* extension point. For each of the returned plugins, *preferences* can execute a contract method

It is possible for plugins to have multiple extension points, they can also extend multiple different plugins and extension points.

This kind of plugin use has the parent plugin executing methods on the children.

Doing it in Code

Now the theory is done with, let's put it all together.

Main Application

Your main application is usually just a bootstrap for your plugins. Application startup will typically consist of the following three steps (as far as PlugMan is concerned):

1) Load all the plugins, usually from a subdirectory;
2) Start all the plugins; and
3) Run the method of your main/base/core plugin.

Of course all this is advisory only. You can load plugins at any time you like, start/stop them at will and run whatever method you feel like on any plugins you desire. But the above steps will help get you started in you application.

Other things your application may do, but are not listed above, are check command -line parameters or set up other application specific infrastructure. The following code snippets are take from the PlugMan demo called Texter (./src/demos/Texter.rb). Texter is a trivial example application demonstrating PlugMan's abilities. Texter takes a string and transforms it in many different ways using plugins.

Load All Plugins

PlugMan defines a method to do this for us. It recursively scans a directory tree for all .rb files and loads them. Typically all the plugins for an application will live in their own distinct directory tree.

```
# load all the plugins in the plugin dir
PlugMan.load_plugins "./demos/text_demo_plugins"
```

Start the Plugins

Once again, PlugMan has methods to perform this. Plugins can be loaded individually using

PlugMan.start_plugin(plug_name) or all registered plugins can be loaded using *PlugMan.start_all_plugins.*

```
  # Start all the plugins
  PlugMan.start_all_plugins
```

PlugMan.start_plugin(plug_name) is smart enough to know that a plugin can't be started without starting all it's parent (required) plugins first. In this case, all dependent plugins are also started your behalf.

The same goes for stopping plugins. There are two methods *PlugMan.stop_plugin(plug_name) and PlugMan.stop_all_plugins.* The former will stop any plugins that depend on the plugin being stopped.

Run the Main/Base/Core Plugin

Typically you will define a main/base/core plugin that is the parent of all your application's plugins. This should typically add the PlugMan root plugin as *required*. In the *Texter* application there is the :*main* plugin in the *core* subdirectory. The application code runs the *:main* plugin after all the plugins are loaded and started:

```
    # execute the transformations using Hello, World!
    PlugMan.registered_plugins[:main].do_xforms("Hello, World!")
```

PlugMan.registered_plugins is a Hash of all the registered plugins in the system (both started and stopped.) The Hash is indexed by the plugin's name, so to get a reference to the *:main* plugin, we use the code:

```
  PlugMan.registered_plugins[:main]
```

And the application knows that the *:main* plugin has the method *do_xforms(str)*...so it ust runs it.

There are no hard or enforced interface contracts in PlugMan, just ol' duck typing. When a plugin author creates a plugin or extension point, they should document the interface so other plugin authors can easily interact with their plugins.

Plugin Goodness

Plugins are of the class *Plugin*, but you don't actually create any subclasses of *Plugin.* Instead, your plugin files (which are .rb files) will contain a code block that defines the plugin using the *PlugMan.define* method. *PlugMan.define* will create a Plugin instance on your behalf and add it to the registry of plugins. It will also do some metadata manipulation behind the scenes.

An example of a plugin definition is (the entire contents of CaseSwapRevers.rb):

```
PlugMan.define :case_swap_reverse do
  author "Aaron"
  version "1.0.0"
  extends({ :main => [:transform] })
  requires [:case_swap, :reverse]
  extension_points []
  params({ :description => "Swaps case and reverses the input 
text, using other plugins." })
```

```
  def xform(str)
    ret = PlugMan.registered_plugins[:case_swap].xform(str)
    PlugMan.registered_plugins[:reverse].xform(ret)
  end
end
```

The first line calls *PlugMan.define* telling PlugMan the name of the new plugin. The lines following describe the plugin metadata. These fields are described as:

author	The plugin author.
version	The plugin version. Only latest version of a plugin can be active.
extends	Extension points this plugin extends { :parent_plug => [:extpt1, :extpt2], parent_plug2 => [:extpt3] }
requires	The plugins that are required by this plugin (not needed for extensions though).
extension_points	Extension points defined by this plugin [:ext_a:, :ext_b] that other plugins can extend.
params	Parameters to pass to the parent plugin { :param1 => "abc", :param2 => 123 }
source_file	The file that the plugin was loaded from, populated by PlugMan.

The above plugin does not define any extension points but does extend the *:main* plugin and requires the use of two other plugins, *case_swap* and *:reverse*.

The *xform(str)* method is the method required by *:main* plugin's *:transform* extension point. During the processing of the *:main* plugin,*xform(str)* will be executed.

Running Extensions

Extensions are useful for openly extending your application. The beauty of extensions is that the parent plugin doesn't need to know anything about the extending plugin. As long as the extension implements the require interface method, it can be successfully used by the parent plugin.

PlugMan know which plugins have extension points and which plugins extend those extension points. A plugin needs only to ask PlugMan for a list of plugins that extend an extension point.

Here is an example of a plugin that defines an extension point (a slightly modified version of Texter's *:main* plugin):

```
PlugMan.define :main do
  author "Aaron"
  version "1.0.0"
  extends(:root => [:root])
  requires []
  extension_points [:transform]
  params()
```

```
# Uses plugins to perform string transformations
def do_xforms(input)

  puts "Input text is #{input.inspect}\n"

  # run the text processor plugins
  PlugMan.extensions(:main, :transform).each do |plugin|
    out = plugin.xform(input)
    puts out
  end
end
```

It can be seen that this plugin defines an extension point called *:transform*. In **bold** is the line of code that gets a list of plugins from PlugMan that extend the *:transform* extension point. Each of the plugins that extend *:transform* must implement the *xform(str)* method.

The method *PlugMan.extension(plug_name, ext_name)* asks PlugMan for a list of plugins that extend the plugin *plug_name*'s extension point *:transform*. Theoretically it is possible for a plugin to request a list of plugins that extend another plugins extension point but this is probably not a wise thing to do unless you really know what you are doing.

It is also possible to have multiple extension points in a plugin.

Plugin Lifecycle

The Plugin class defines plugin lifecycle methods *start* and *stop*. These are executed when *PlugMan.start_plugin(plug_name)* and *PlugMan.stop_plugin(plug_name)* are run. By default these methods just return *true* but if there is any specific processing required in the plugin start/stop you can override these methods. Returning Boolean true indicates the method was successful, false indicates some kind of error occurred.

Cleaning Up

It is advisable to stop all extensions before terminating a programme.Each plugin can have its own shutdown sequence that may persist data or cleanup resources. This can be done using *PlugMan.stop_all_plugins*.