# PluginManager

Contents

## New Plugin Manager

Revised: 31 August, 2005 LarryStone

This Plugin Manager has been substantially redesigned to be more useful throughout the platform, and on a larger scale, than the first proposal. Its feature set has been reduced to just what is needed at the present time.

## Synopsis

The `PluginManager` is a very simple component container.
It creates and organizes components (plugins), and helps select a plugin in the cases where there are many possible choices. It also gives some limited control over the lifecycle of a plugin.
Please keep in mind, it is *not* an "inversion of control" (IoC) framework like Spring or Picocontainer.
DSpace's `PluginManager` is what IoC folks call an "active dependency manager". It could be used as the engine *behind* a simple IoC framework, if that is desired someday.
See ModularityMechanism for more details about coding techniques and standards to facilitate moving to a plug-in architecture.
This proposal is only concerned with the machinery of choosing a plugin from among various implementations.
Also see EvolutionToServiceLocatorProposal for a picture of a DSpace platform where all modules are plugins. It can be built on the "singleton" plugin mechanism from this proposal.

## Concepts

Be familiar with the following terms before reading the rest of this document:

- Plugin Interface
  A Java *interface*, the defining characteristic of a plugin. The consumer of a plugin asks for its plugin *by interface*.
- Plugin
  a.k.a. *Component*, this is an instance of a class that implements a certain *interface*. It is interchangeable with other implementations, so that any of them may be "plugged in", hence the name. A Plugin is an instance of any class that implements the *plugin interface*.
- Implementation class
  The actual class of a plugin. It may implement several plugin interfaces, but *must* implement at least one.
- Name
  Plugin implementations *can* be distinguished from each other by *name*, a short `String` meant to symbolically represent the implementation class. They are called *"named plugins"*. Plugins only need to be named when the caller has to make an active choice between them.
- SelfNamedPlugin class
  Plugins that extend the `SelfNamedPlugin` class can take advantage of additional features of the Plugin Manager. Any class can be managed as a plugin, so it is not *necessary*, just possible.
- Reusable
  *Reusable* plugins are only instantiated once, and the Plugin Manager returns the same (cached) instance whenever that same plugin is requested again. This behavior can be turned off if desired.

## Using the Plugin Manager

### Types of Plugin

The Plugin Manager supports three different patterns of usage:

#### 1. Singleton Plugins

There is only one implementation class for the plugin. It is
indicated in the configuration.
For examples, see
EvolutionToServiceLocatorProposal.
This type of plugin chooses an implementation of a service, for
the entire system, at configuration time.
Your application just fetches the
plugin for that interface and gets the configured-in choice.
See the #PluginManager Class getSinglePlugin() method.

#### 2. Sequence Plugins

You need a *sequence* or series of plugins, to implement a mechanism
like StackableAuthenticationMethods or a pipeline, where each plugin
is called in order to contribute its implementation of a process to the whole.
The Plugin Manager supports this by letting you configure a sequence
of plugins for a given interface. See the
#PluginManager Class getPluginSequence() method for details.

#### 3. Named Plugins

Use a *named plugin*when the application has to choose one plugin implementation out of
many available ones.
Each implementation
is bound to one or more *names* (symbolic identifiers) in the
configuration.

The *name* is just a `String` to be associated with the combination of
*implementation class* and *interface*. It may contain any characters
**except for comma (`,`) and equals (`=`)**. It may contain embedded spaces. Comma is
a special character used to separate names in the configuration entry.
Names must be unique within an interface: No plugin classes
implementing the same interface may have the same name.

Think of plugin names as a *controlled vocabulary* – for a given
plugin interface, there is a set of names for which plugins can be
found. The designer of a Named Plugin interface is responsible for
deciding what the name means and how to derive it; for example, names
of metadata crosswalk plugins may describe the target metadata format.
See the #PluginManager Class getNamedPlugin() method and the
#PluginManager Class getPluginNames() method
for more details.

##### Self-Named Plugins

Named plugins can get their names either from the
configuration or, for a variant called *self-named plugins*, from
within the plugin itself.

Self-named plugins are necessary because one plugin implementation can
be *configured* itself to take on many "personalities", each of which
deserves its own plugin name. It is already managing its own
configuration for each of these personalities, so it makes sense to
allow it to export them to the Plugin Manager rather than expecting
the plugin configuration to be kept in sync with it own configuration.
An example helps clarify the point: There is a named plugin that does
crosswalks, call it CrosswalkPlugin. It has several implementations
that crosswalk some kind of metadata. Now we add a new plugin
which uses XSL stylesheet transformation (XSLT) to crosswalk many
types of metadata – so the single plugin can act like many different
plugins, depending on which stylesheet it employs.

This XSLT-crosswalk plugin has its own configuration that maps
a Plugin Name to a stylesheet – it has to, since of course the Plugin
Manager doesn't know anything about stylesheets. It becomes a self-named
plugin, so that it reads its configuration data, gets the list of names
to which it can respond, and passes those on to the Plugin Manager.
When the Plugin Manager creates an instance of the XSLT-crosswalk,
it records the Plugin Name that was responsible for that instance.
The plugin can look at that Name later in order to configure itself
correctly for the Name that created it. This mechanism is all part
of the `SelfNamedPlugin` class which is part of any self-named plugin.

### Obtaining a Plugin Instance

The most common thing you will do with the Plugin Manager is obtain an
instance of a plugin. To request a plugin, you must *always* specify
the **plugin interface** you want. You will also supply a *name* when
asking for a *named plugin*.

A *sequence plugin* is returned as an array of `Objects` since it
is actually an ordered list of plugins.
See the methods
#PluginManager Class getSinglePlugin(), getPluginSequence(), getNamedPlugin()

### Lifecycle Management

When `PluginManager` fulfills a request for a plugin, it checks whether
the implementation class is *reusable*; if so, it creates one
instance of that class and returns it for every subsequent request for
that interface and name. If it is not reusable, a new instance is always
created.

For reasons that will become clear later, the manager actually caches
a separate instance of an implementation class for each *name* under
which it can be requested.

You can ask the `PluginManager` to forget about (*decache*) a
plugin instance, by *releasing* it. See the `PluginManager.releasePlugin()`
method. The manager will drop its reference to the plugin so the garbage
collector can reclaim it. The next time that plugin/name combination is
reuqested, it will create a new instance.

### Getting Meta-Information

The `PluginManager` can list all the names of the
Named Plugins which implement an interface.
You may need this, for example, to implement a menu in a user interface
that presents a choice among all possible plugins.
See the method
#PluginManager Class getPluginNames().
Note that it only returns the *plugin name*, so if you need a
more sophisticated or meaningful "label" (*i.e.* a key into the I18N
message catalog) then you should add a method to the plugin itself to
return that.

## Implementation

**Note:** The `PluginManager` refers to interfaces and classes internally
only by their names whenever possible, to avoid loading classes until
absolutely necessary (*i.e.* to create an instance). As you'll see
below, self-named classes still have to be loaded to query them for names,
but for the most part it can avoid loading classes. This saves a lot
of time at start-up and keeps the JVM memory footprint down, too. As
the Plugin Manager gets used for more classes, this will become a greater
concern.

The only downside of "on-demand" loading is that errors in the configuration
don't get discovered right away. The solution is to call the
`checkConfiguration()` method after making any changes to the configuration.

## PluginManager Class

The `PluginManager` class is your main interface to the Plugin Manager.
It behaves like a factory class that never gets instantiated, so
its public methods are `static`.
Here are the public methods, followed by explanations:

### getSinglePlugin()

```
static Object getSinglePlugin(Class intface)
throws PluginConfigurationError;
```

Returns an instance of the *singleton (single) plugin* implementing
the given interface. '''There must be exactly one single plugin
configured for this interface,''' otherwise the `PluginConfigurationError`
is thrown.

Note that this is the only "get plugin" method which throws an exception.
It is typically used at initialization time to set up a
permanent part of the system so any failure is fatal.
See the `plugin.single` configuration key for configuration details.

### getPluginSequence()

```
static Object getPluginSequence(Class intface);
```

Returns instances of all plugins that implement
the interface *intface*, in an `Array`. Returns an empty array
if no there are no matching plugins.
The order of the plugins in the array is the same as their class names
in the configuration's value field.
See the `plugin.sequence` configuration key for configuration details.

### getNamedPlugin()

```
static Object getNamedPlugin(Class intface, String name);
```

Returns an instance of a plugin that implements the interface *intface*and is bound to a name matching *name*. If there is no matching
plugin, it returns `null`. The names are matched by `String.equals()`.
See the `plugin.named` and
`plugin.selfnamed` configuration keys for configuration details.

### releasePlugin()

```
static void releasePlugin(Object plugin);
```

Tells the Plugin Manager to let go of any references to a reusable
plugin, to prevent it from being given out again and to allow the
object to be garbage-collected. Call this when a plugin instance
must be taken out of circulation.

### getAllPluginNames()

```
static String getAllPluginNames(Class intface);
```

Returns all of the names under which a named plugin implementing
the interface *intface* can be requested (with `getNamedPlugin()`).
The array is empty if there are no matches. Use this to populate a
menu of plugins for interactive selection, or to document what the
possible choices are.
"The names are NOT returned in any predictable order, so you may wish
to sort them first."
**Note:** Since a plugin may be bound to more than one name, the
list of names this returns does not represent the list of plugins.
To get the list of unique implementation classes corresponding to the
names, you might have to eliminate duplicates (*i.e.* create a `Set` of classes).

### checkConfiguration()

```
static void checkConfiguration();
```

Validates the keys in the DSpace `ConfigurationManager` pertaining
to the Plugin Manager and reports any errors by logging them.
This is intended to be used interactively by a DSpace administrator,
to check the configuration file after modifying it.
See the section about #Validating the Configuration for details.

## SelfNamedPlugin Class

A named plugin implementation must `extend` this class if it
wants to supply its own Plugin Name(s). See
Self-Named Plugins for why this is sometimes necessary.

```
abstract class SelfNamedPlugin
{
  // Your class must override this:
  // Return all names by which this plugin should be known.
  public static String[] getPluginNames();

  // Returns the name under which this instance was created.
  // This is implemented by SelfNamedPlugin and should NOT be overridden.
  public String getPluginInstanceName();
}
```

## Errors and Exceptions

```
public class PluginConfigurationError extends Error
{
  public PluginConfigurationError(String message);
}
```

An error of this type means the caller asked for a single plugin, but
either there was no single plugin configured matching that interface,
or there was more than one. Either case causes a fatal configuration error.

```
public class PluginInstantiationException extends untimeException
{
  public PluginInstantiationException(String msg, Throwable cause)
}
```

This exception indicates a fatal error when instantiating a plugin class.
It should only be thrown when something unexpected happens in the
course of instantiating a plugin, e.g. an access error, class not found,
etc. Simply not finding a class in the configuration is not an exception.
This is a `RuntimeException` so it doesn't have to be declared, and can
be passed all the way up to a generalized fatal exception handler.

## Configuring Plugins

All of the Plugin Manager's configuration comes from the DSpace
Configuration Manager, which is a Java `Properties` map. You can
configure these characteristics of each plugin:
#**Interface:** Classname of the Java interface which defines the plugin, including package name. *e.g.* `org.dspace.app.mediafilter.MediaFilter`
#**Implementation Class:** Classname of the implementation class, including package. *e.g.* `org.dspace.app.mediafilter.PDFFilter`
#**Names:** *(Named plugins only)* There are two ways to bind names to plugins: listing them in the value of a `plugin.named.`*interface* key, or configuring a class in `plugin.selfnamed.`*interface* which extends the `SelfNamedPlugin` class.
#**Reusable option:** *(Optional)* This is declared in a `plugin.reusable` configuration line. Plugins *are reusable* by default, so you only need to configure the non-reusable ones.

### Configuring Singleton (Single) Plugins

This entry configures a Single Plugin for use with `getSinglePlugin()`:

**`plugin.single.`***interface* **`=`** *classname*

For example, this configures the class `org.dspace.app.webui.SimpleAuthenticator`
as the plugin for interface `org.dspace.app.webui.SiteAuthenticator`:

```
plugin.single.org.dspace.app.webui.SiteAuthenticator = org.dspace.app.webui.SimpleAuthenticator
```

### Configuring Sequence of Plugins

This kind of configuration entry defines a Sequence Plugin,
which is bound to a
*sequence* of implementation classes. The key identifies the
*interface*, and the value is a comma-separated list of *classnames*:

**`plugin.sequence.`***interface* **`=`** *classname***`,`** *...*

The plugins are returned by `getPluginSequence()` in the same order as
their classes are listed in the configuration value.
For example, this entry configures the StackableAuthenticationMethods
with three implementation classes:

```
plugin.sequence.org.dspace.eperson.AuthenticationMethod = \
 org.dspace.eperson.X509Authentication, \
 org.dspace.eperson.PasswordAuthentication, \
 edu.mit.dspace.MITSpecialGroup
```

### Configuring Named Plugins

There are two ways of configuring named plugins:

#### 1. Plugins Named in the Configuration

A named plugin which gets its name(s) from the configuration is listed
in this kind of entry:

**`plugin.named.`***interface* **`=`** *classname* **`=`** *name* `` **`,`** *name..* ` ` *classname* **`=`** *name..* ``

The syntax of the configuration value is: *classname*, followed by
an equal-sign and then at least one plugin *name*. ind more names
to the same implementation class by by adding them here, separated by commas.
Names may include any character other than comma
(`**,**`)
and equal-sign
(`**=**`).
For example, this entry creates one plugin with the names `GIF`, `JPEG`,
and `image/png`, and another with the name `TeX`:

```
plugin.named.org.dspace.app.mediafilter.MediaFilter = \
 org.dspace.app.mediafilter.JPEGFilter = GIF, JPEG, image/png \
 org.dspace.app.mediafilter.TeXFilter = TeX
```

This example shows a plugin name with an
embedded whitespace character. Since comma (`,`) is the separator
character between plugin names, spaces are legal (between words of
a name; leading and trailing spaces are ignored).
This plugin is bound to the names "`Adobe PDF`", "`PDF`", and
"`Portable Document Format`".

```
plugin.named.org.dspace.app.mediafilter.MediaFilter = \
 org.dspace.app.mediafilter.TeXFilter = TeX \
 org.dspace.app.mediafilter.PDFFilter =  Adobe PDF, PDF, Portable Document Format
```

**NOTE:** Since there can only be one key with `plugin.named.` followed
by the interface name in the configuration, *all* of the plugin
implementations must be configured in that entry.

### 2. Self-Named Plugins

Since a self-named plugin supplies its own names through a static
method call, the configuration only has to include its interface and
classname:

`**plugin.selfnamed.**`*interface* `**=**` *classname* `` `**,**` *classname..* ``

The following example
first demonstrates how the plugin class,
`XsltDisseminationCrosswalk` is configured to implement
its own names "`MODS`" and "`DublinCore`". These come from
the keys starting with `crosswalk.dissemination.stylesheet.`. The
value is a stylesheet file.
The class is then configured as a self-named plugin:

```
crosswalk.dissemination.stylesheet.DublinCore = xwalk/TESTDIM-2-DC_copy.xsl
crosswalk.dissemination.stylesheet.MODS = xwalk/mods.xsl
plugin.selfnamed.crosswalk.org.dspace.content.metadata.DisseminationCrosswalk = \
 org.dspace.content.metadata.MODSDisseminationCrosswalk, \
 org.dspace.content.metadata.XsltDisseminationCrosswalk
```

**NOTE:** Since there can only be one key with `plugin.selfnamed.` followed
by the interface name in the configuration, *all* of the plugin
implementations must be configured in that entry. The
`MODSDisseminationCrosswalk` class is only shown to illustrate this point.

## Configuring the Reusable Status of a Plugin

Plugins are assumed to be *reusable* by default, so you only need
to configure the ones which you would prefer not to be reusable.
The format is as follows:

`**plugin.reusable.**`*classname* `**=**` `(` `**true**` `|` `**false**` `)`

For example, this marks the PDF plugin from the example above as non-reusable:

```
plugin.reusable.org.dspace.app.mediafilter.PDFFilter = false
```

## Validating the Configuration

Anchor(validate)The Plugin Manager is very sensitive to mistakes in
the DSpace configuration. Subtle errors can have unexpected consequnces
that are hard to detect: for example, if there are two
`"plugin.single"` entries for the same interface, one of them will be
silently ignored.

To validate the Plugin Manager configuration, call the
`PluginManager.checkConfiguration()` method. It looks for the following
mistakes:

- Any duplicate keys starting with `"plugin."`.
- Keys starting `plugin.single`, `plugin.sequence`, `plugin.named`, and `plugin.selfnamed` that don't include a valid interface.
- Classnames in the configuration values that don't exist, or don't implement the plugin interface in the key.
- Classes declared in `plugin.selfnamed` lines that don't extend the `SelfNamedPlugin` class.
- Any name collisions among named plugins for a given interface.
- Named plugin configuration entries without any names.
- Classnames mentioned in `plugin.reusable` keys must exist and have been configured as a plugin implementation class.

The `PluginManager` class also has a `main()` method which simply runs
`checkConfiguration()`, so you can invoke it from the command line to
test the validity of plugin configuration changes.
Eventually, someone should develop
a general configuration-file sanity checker for DSpace, which would
just call `PluginManager.checkConfiguration()`.

## Use Cases

Here are some usage examples to illustrate how the Plugin Manager works.

### Managing the MediaFilter plugins transparently

The existing DSpace 1.3 `MediaFilterManager` implementation can
be largely replaced by
the Plugin Manager. The `MediaFilter` classes
become plugins named in the configuration.
These lines are added to the DSpace configuration:

```
plugin.named.org.dspace.app.mediafilter.MediaFilter = \
 plugin.named.org.dspace.app.mediafilter.HTMLFilter; HTML, Text \
 plugin.named.org.dspace.app.mediafilter.JPEGFilter;  GIF, JPEG, image/png \
 plugin.named.org.dspace.app.mediafilter.PDFFilter; Adobe PDF, PDF \
 plugin.named.org.dspace.app.mediafilter.WordFilter; Microsoft Word, Word
```

Add this code to `MediaFilterManager` where it chooses a plugin. Note
how the plugin name comes from the `itstreamFormat`'s short name; this
is part of the design and contract of that particular plugin.

```
MediaFilter myFilter = (MediaFilter)PluginManager.getNamedPlugin(MediaFilter.class,
 myitstream.getFormat().getShortDescription());
```

### A Singleton Plugin

This shows how to configure and access a single anonymous plugin,
such as the `SiteAuthenticator` plugin in DSpace 1.3:
Configuration:

```
plugin.single.org.dspace.app.webui.SiteAuthenticator = edu.mit.dspace.MITAuthenticator
```

The following code fragment shows how `siteAuth`, the service object, is
initialized and used:

```
SiteAuthenticator siteAuth =
 (SiteAuthenticator)PluginManager.getSinglePlugin(SiteAuthenticator.class);
siteAuth.startAuthentication(context, request, response);
```

## Plugin that Names Itself

This crosswalk plugin acts like many different plugins since it is
configured with different XSL translation stylesheets. Since it already
gets each of its stylesheets out of the DSpace configuration, it makes sense
to have the plugin give `PluginManager` the names to which it answers
instead of forcing someone to configure those names in two places
(and try to keep them synchronized).
**NOTE:** Remember how `getPlugin()` caches a separate instance of
an implementation class for every *name* bound to it? This is why:
the instance can look at the name under which it was invoked and configure
itself specifically for that name. Since the instance for each name
might be different, the Plugin Manager has to cache a separate
instance for each name.

Here is the configuration file listing both the plugin's own configuration
and the `PluginManager` config line:

```
crosswalk.dissemination.stylesheet.DublinCore = xwalk/TESTDIM-2-DC_copy.xsl
crosswalk.dissemination.stylesheet.MODS = xwalk/mods.xsl
plugin.selfnamed.org.dspace.content.metadata.DisseminationCrosswalk = \
 org.dspace.content.metadata.XsltDisseminationCrosswalk
```

This look into the implementation shows how it finds configuration
entries to populate the array of plugin names returned by the
`getPluginNames()` method. Also note, in the `getStylesheet()` method,
how it uses the plugin name that created the current instance (returned
by `getPluginInstanceName()`) to find the correct stylesheet.

```
public class XsltDisseminationCrosswalk extends SelfNamedPlugin
{
  ....
  private final String prefix = "crosswalk.dissemination.stylesheet.";
  ....
  public static String getPluginNames()
  {
    List aliasList = new ArrayList();
    Enumeration pe = ConfigurationManager.propertyNames();
    while (pe.hasMoreElements())
    {
      String key = (String)pe.nextElement();
      if (key.startsWith(prefix))
        aliasList.add(key.substring(prefix.length()));
    }
    return (String)aliasList.toArray(new StringaliasList.size());
  }

  // get the crosswalk stylesheet for an instance of the plugin:
  private String getStylesheet()
  {
    return ConfigurationManager.getProperty(prefix + getPluginInstanceName());
  }
}
```

## Stackable Authentication

The Stackable Authentication mechanism needs
to know all of the plugins configured for the interface, in the
order of configuration,
since order is significant. It gets a Sequence Plugin from the
Plugin Manager.
Configuring stackable authentication would look like this:

```
plugin.sequence.org.dspace.eperson.AuthenticationMethod = \
 org.dspace.eperson.X509Authentication, \
 org.dspace.eperson.PasswordAuthentication, \
 edu.mit.dspace.MITSpecialGroup
```

Within the `AuthenticationManager`, the code to initialize the "stack"
of plugins looks like this:

```
AuthenticationMethod stack = getPluginSequence(AuthenticationMethod.class);
```

## Comments?

- There may be a need someday for "named sequence of plugins": For example, if an application can choose between different "pipelines" of classes, it would want the Plugin Manager to offer alternative sequences of plugins for the same interface. To put it another way, Named sequences are to sequence plugins as Named Plugins are to Single Plugins. This can be left for future implementation, once there is a demonstrated need.