OSI2 Design and Architecture

This document describes the design and architecture plans for OSI2.

There are a number of overarching goals:

- A plugin API architecture, so that individual APIs can be kept to a manageable size and implementation of a module in support of an API is a manageable task.
- Dynamic loading of solvers and other functional modules in support of specific APIs.
- An implementation that is primarily a lightweight and (when necessary) transparent plugin management layer.

This leads to a use model where clients create an OSI2 object and then request it to load modules (often, but not necessarily, solvers) that implement specific APIs. In response to a request, the OSI2 object loads the requested module, invokes the module's factory methods to produce an object supporting the requested API(s), and hands this object over to the client. Clients interact with these objects according to the definition of the API.

The OSI2 object acts as a broker, handling the mechanics of managing the plugins.

Plugin Architecture

The plugin architecture used in OSI2 is broadly adopted from an architecture originally described by G. Sayfan [1].

There is one (static) instance of the plugin manager, returned by a call to `getInstance`. The plugin manager provides certain services and information to plugins, defined by the `PlatformServices` class. In particular, a `PlatformServices` object specifies a registration method that allows a plugin to register with the plugin manager, and it provides a service method that the plugin can invoke to request other services from the manager (currently unimplemented).

An immediate deficiency in the original architecture is that the plugins used for OSI2 are in general not stateless, hence the architecture has been expanded to allow a reference to a plugin object to be returned to the plugin manager and passed back whenever plugin services are invoked.

The working sequence to load a plugin and return an object to the client goes like this

1. Load the shared library that implements the plugin.

   Currently, this is accomplished by a bare call to the static `load` method in the `DynamicLibrary` class. The plugin manager can be interrogated to return a default plugin directory. This should be better hidden once we've settled on how to communicate a list of directories to be searched for plugins.

2. Initialise and register the plugin.

   Given a `DynamicLibrary` object, we can use `getSymbol` to retrieve the `initPlugin` method. This method is then passed as a parameter to the plugin manager's `initializePlugin` method. `initializePlugin` in turn invokes the `initPlugin` method, passing a `PlatformServices` structure as a parameter.

   ¡¡ *So why not bury all of this within the plugin manager? Let the plugin manager keep track of loaded plugin libraries. Do we want to allow separate load and initialisation? Or should they be an atomic operation from the client's point of view?* ¿¿

The `initPlugin` method is responsible for getting the plugin ready to work and registering its capabilities (APIs). (For example, the `ClpShim` plugin is responsible for loading the clp solver library.) For each API supported by the plugin library, a `RegisterParams` object is created specifying an identifying string, create and destroy methods, descriptive information (version and language), and an associated plugin object to hold state. The `RegisterParams` object is handed back to the plugin manager as a parameter to the registration (`registerObject`) method supplied in the `PlatformServices` object.

The plugin manager's `registerObject` method adds the `RegisterParams` object to an appropriate map (specific or wildcard) according to the identifying string. It then returns to the `initPlugin`.

The `initPlugin` method returns an appropriate cleanup method (an `ExitFunc`) to the `initializePlugin` method, which stores it for use when the plugin library is unloaded. Initialisation is complete.

3. Ask the plugin manager for an object that implements an API.

   Using the plugin manager's `createObject` method, the user asks for an API by name. The `createObject` method searches the maps held by the plugin manager looking for a suitable match. An `ObjectParams` object is created, including the API(s) requested, a `PlatformServices` object, and the plugin object itself (the only way for the plugin to maintain state across calls). This is passed as a parameter to the create function that the plugin registered for this type of API. The returned object is handed back to the client.

It's useful to note that a given plugin library (loadable shared library) can supply multiple APIs, and asking for an object that implements some combination of APIs is possible. It's the business of the plugin object to keep track of what particular APIs or combination of APIs it supports, and what objects it's given out.

Which begs the question, "How will the plugin manage capabilities?" I guess in the end it's the plugin's problem. If it hands out a single object that supports some number of APIs, then there's no real issue. If it can do more complicated things, then it'll need a way to handle that.

On our side, perhaps some sort of `inquireAPI` method that'll allow the plugin to inspect an API and decide if it supports it.

And now I can see an interesting issue. The client asks for some set of APIs. It could well be that the client has loaded more than one plugin capable of supporting the API set. Some solvers are better at one task, some at another. Do we allow the user to choose? Seems like we have to.

And what about that plugin that hands out a single object that supports multiple APIs? How (should?) we allow the client to signal that it wants the same object, if possible. In the opposite case, the client might want an entirely clean object, not connected with the previous one even when that's possible.

API Architecture

APIs will derive from a common top-level virtual class `Osi2::API` as shown in Figure 1.

References

[1] G. Sayfan. Building your own plugin framework, parts 1 – 5. *Dr. Dobbs*, November 2007. Online at http://drdobbs.com/cpp/204202899 (first of five parts).

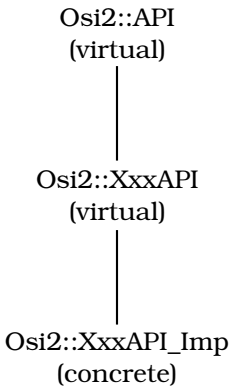

Figure 1: Class Hierarchy for Osi2 APIs