Les bases d’un système de plugins

Réaliser un système modulaire est un objectif très classique pour des informaticiens. Le terme de Plugin arrive fréquemment peu après la modularité : un Plugin étend les fonctionnalités d’un système de façon optionnel, dynamiquement à l’exécution.

# Le principe fondamental

Le Système doit travailler avec des objets qu’il ne « connait pas » directement. Pour pouvoir néanmoins interagir a minima avec le Plugin (ou permettre au Plugin d’interagir avec lui), il doit néanmoins en connaître « quelque chose ». Ce quelque chose est le plus souvent une classe de base (typiquement abstraite) ou une interface.

La première nécessité d’un Système de Plugin est un « contrat » minimal que le Plugin doit satisfaire.  
Le contrat s’exprime par une abstraction (classe de base ou interface dans le paradigme objet traditionnel).

Ci-dessous, en exemple de Contrat : tous les Plugins de CiviKey doivent implémenter cette interface.

#region LGPL License

/\*----------------------------------------------------------------------------

\* This file (CK.Plugin.Model\IPlugin.cs) is part of CiviKey.

\*

\* CiviKey is free software: you can redistribute it and/or modify

\* it under the terms of the GNU Lesser General Public License as published

\* by the Free Software Foundation, either version 3 of the License, or

\* (at your option) any later version.

\*

\* CiviKey is distributed in the hope that it will be useful,

\* but WITHOUT ANY WARRANTY; without even the implied warranty of

\* MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the

\* GNU Lesser General Public License for more details.

\* You should have received a copy of the GNU Lesser General Public License

\* along with CiviKey. If not, see <http://www.gnu.org/licenses/>.

\*

\* Copyright © 2007-2012,

\* Invenietis <http://www.invenietis.com>,

\* In’Tech INFO <http://www.intechinfo.fr>,

\* All rights reserved.

\*-----------------------------------------------------------------------------\*/

#endregion

namespace CK.Plugin

{

/// <summary>

/// This interface defines the minimal properties and behavior of a plugin.

/// </summary>

public interface IPlugin

{

/// <summary>

/// This method initializes the plugin: own resources must be acquired and running conditions should be tested.

/// No interaction with other plugins must occur (interactions must be in <see cref="Start"/>).

/// </summary>

/// <param name="info">Enables the implementation to give detailed information in case of error.</param>

/// <returns>True on success. When returning false, <see cref="IPluginSetupInfo"/> should be used to

/// return detailed explanations.

/// </returns>

bool Setup( IPluginSetupInfo info );

/// <summary>

/// This method must start the plugin: it is called only if <see cref="Setup"/> returned true.

/// Implementations can interact with other components (such as subscribing to their events).

/// </summary>

void Start();

/// <summary>

/// This method uninitializes the plugin (it is called after <see cref="Stop"/>).

/// Implementations MUST NOT interact with any other external components: only internal resources should be freed.

/// </summary>

void Teardown();

/// <summary>

/// This method is called by the host when the plugin must not be running anymore.

/// Implementations can interact with other components (such as unsubscribing to their events).

/// <see cref="Teardown"/> will be called to finalize the stop.

/// </summary>

void Stop();

}

}

Le cœur du Système ne connait que le Contrat. Il doit néanmoins travailler avec des objets concrets (qui implémentent/supportent ce contrat). Et une des premières choses qu’il va devoir faire, c’est d’instancier un ou plusieurs de ces objets concrets.

# Du nom d’un Type …à un objet de ce type

En .Net, en Java, et dans la majorité des langages et plateformes modernes, il existe toujours un moyen de désigner/nommer un type ou une classe concrète. En .Net, c’est le type « Type » qui représente le type d’un objet, en Java, c’est la classe « Class ».

Un Type qui représente un Type, une Classe qui représente une Classe ? Bienvenu dans le monde de la « Reflection » qui est la capacité d’un Système/Langage/Plateforme de manipuler ses propres objets et concepts.

Tout Système de Plugin repose sur un moyen de désigner/nommer le « Type » concret d’un Plugin. Les « Plugins à charger » sont souvent dans un fichier de configuration (typiquement en xml)[[1]](#footnote-1) : on démarre le plus souvent avec une chaîne de caractère qui décrit le Type concret.

Comment obtenir un « vrai objet » (une instance) avec lequel le Système va pouvoir travailler ? Il suffit de 3 lignes de code :

C#

string pluginTymeName = "MyCompany.Plugins.MyPluginThatWorksWell, MyCompany.Plugins";

Type pluginType = Type.GetType( pluginTymeName );

IPlugin thePlugin = (IPlugin)Activator.CreateInstance( pluginType );

Java

string pluginTymeName = "myCompany.plugins.MyPluginThatWorksWell";

Class pluginClass = Class.forName( pluginTymeName );

IPlugin thePlugin = (IPlugin)pluginClass.newInstance();

Bien sûr, ici, nous n’utilisons qu’une petite partie de ce que permettent ces deux plateformes, sans rentrer dans les subtilités (et les pièges) de chacune d’elles.

Un mot sur cette « chaîne de caractères » magique. C’est simplement le nom du Type ou de la Classe. Les deux plateformes diffèrent de façon assez radicale sur ce point.

En .Net, un Type est pleinement identifié par son AssemblyQualifiedName :

MyCompany.Plugins.MyPluginThatWorksWell, MyCompany.Plugins

Namespace

Espace de nommage

TypeName

Nom de la Classe

AssemblyName

(Nom de la dll)

En .Net, le namespace n’est pas suffisant pour identifier un Type, les espaces de nommages ne sont que des « espaces de rangements » d’objets, sans corrélation avec le « Package » (.dll) dans lequel ils existent.

En java, le « Class Name » qui est composé du namespace et du nom de la classe permet de localiser un objet directement car, en Java, la localisation de la classe est couplée au namespace.

# Un peu d’architecture

Une architecture simple pour Plugin est composée d’une application hôte (l’application principale) – Host en anglais, d’un répertoire pour les Plugins (généralement situé sous l’application et appelé /Plugins/ ou /Extensions/) ainsi que d’un Package commun au Host et aux Plugins qui contient tout ce qui est nécessaire à l’interaction entre les Plugins et le Host.

Dans notre cas, nous aurions :

* Un assembly Host.exe qui contient le code de l’application proprement dite.
* Un assembly Plugin.Model.dll qui contient le Contrat et tous les objets annexes nécessaires au bon fonctionnement du Système.
* Le Host et tous les Plugins connaissent (référencent) Plugin.Model.dll afin de partager le(s) Contrat(s), mais les Plugins n’ont pas à connaître Host.exe, et, symétriquement, le Host ne connait aucun Plugin[[2]](#footnote-2).

Host.exe

Plugin.Model.dll

Plugin1.dll

Plugin2.dll

Plugin3.dll

IPlugin

IPluginSetupInfo

Référence

**Noyau :** Développé et installé en 2010.

**Extensions :** Développées par d’autres développeurs et installées en 2038.

# Découverte dynamique

Les Plugins peuvent être découverts « automatiquement ». Il faut et il suffit de parcourir le répertoire censé les contenir et inspecter le contenu des Packages (.dll ou .jar) qui s’y trouvent.

1. Les « Plugins disponibles » peuvent être découverts automatiquement dans des « Packages » (des .dlls en .Net – des assemblies, des .jar en Java), mais c’est une autre histoire que l’on abordera bientôt. [↑](#footnote-ref-1)
2. En pratique, le Host connait souvent quelques Plugins « de base » livrés en standard avec l’application. Le répertoire /Plugins/ hébergeant les « autres », ceux que l’on ne connait pas au moment du développement de l’application. [↑](#footnote-ref-2)