Création de plug-in avec Qt

par Nicolas Arnaud Cosmos

Date de publication : 18/02/2007

Dernière mise à jour : 23/11/2010

Comment créer des plug-ins avec Qt et de manière multiplateforme ? Comment permettre à mes utilisateurs d'étendre eux-mêmes mon application ? Ce tutoriel va détailler la création d'un système de plug-ins avec la bibliothèque Qt et le chargement de ces plug-ins depuis notre application.

I - Plug-ins, vous avez dit plug-ins ?

I-A - Qu'est-ce que c'est ?

Je n'ai pas réussi à faire mieux que **Wikipedia** pour la définition :

En informatique, un plug-in, de l'anglais to plug in (brancher), parfois traduit en module externe, module enfichable, module d'extension, greffon ou plugiciel, est un logiciel tiers venant se greffer à un logiciel principal afin de lui apporter de nouvelles fonctionnalités. Le logiciel principal fixe un standard d'échange d'informations auquel ses modules se conforment. Le module n'est généralement pas conçu pour fonctionner seul mais avec un autre programme.

I-B - Comment ça marche ?

C'est relativement simple (en théorie) :

- le logiciel principal défini une API standard : un ensemble de classes et de fonctions pouvant être utilisées/héritées par les plug-ins ; c'est le lien, le langage commun entre le logiciel principal et les plug-ins ;
- les plug-ins se basent sur cette API, et créent ainsi de nouvelles fonctionnalités pour le logiciel principal ; ils se présentent généralement sous forme de bibliothèques dynamiques (*.dll sous Windows ou *.so sous Linux) ; pour être chargés par le programme principal, ils doivent être copiés dans un répertoire spécifique ;
- le logiciel principal, au démarrage, charge les plug-ins qu'il trouve, ajoutant ainsi de nouvelles fonctionnalités.

I-C - Avantages ?

Les avantages d'un système de plug-ins sont nombreux. En voici quelques-uns :

- l'utilisateur ou l'intervenant externe peut étendre les fonctionnalités d'une application sans avoir son code source ;
- le logiciel est beaucoup plus modulaire : possibilité de faire des versions réduites, versions complètes□ ;
- la mise en place d'un système de plug-in force à réfléchir un peu à l'architecture du logiciel, ce qui n'est pas un mal.

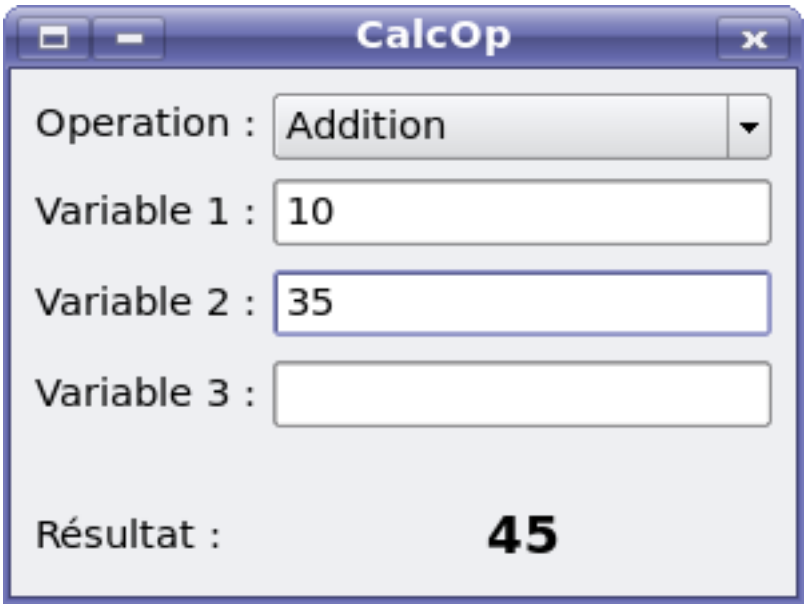
I-D - Les plug-ins avec Qt, pourquoi est-ce si pratique ?

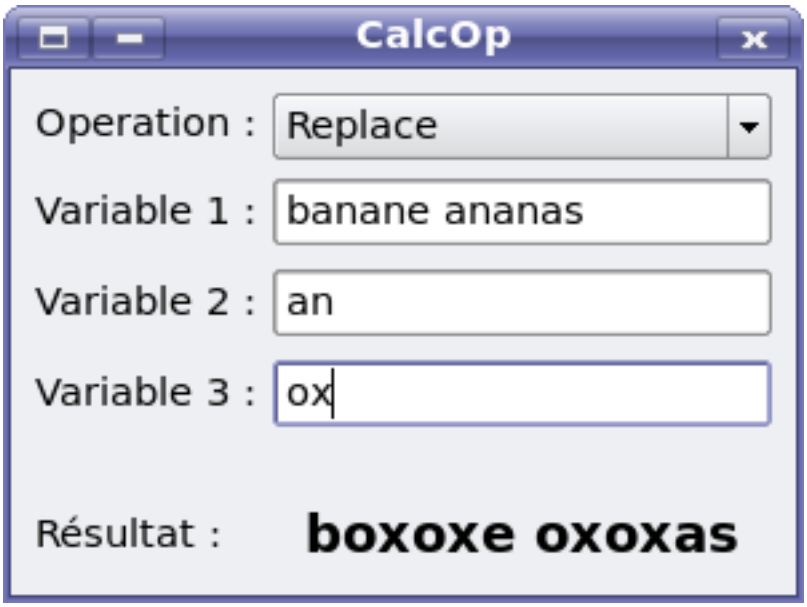
La mise en place d'un système de plug-in est parfois assez lourde et nécessite par exemple de vérifier dans le programme principal si un plug-in contient telle ou telle fonction. Je ne parle même pas d'un système multi-plateforme... heureusement, Qt est là.

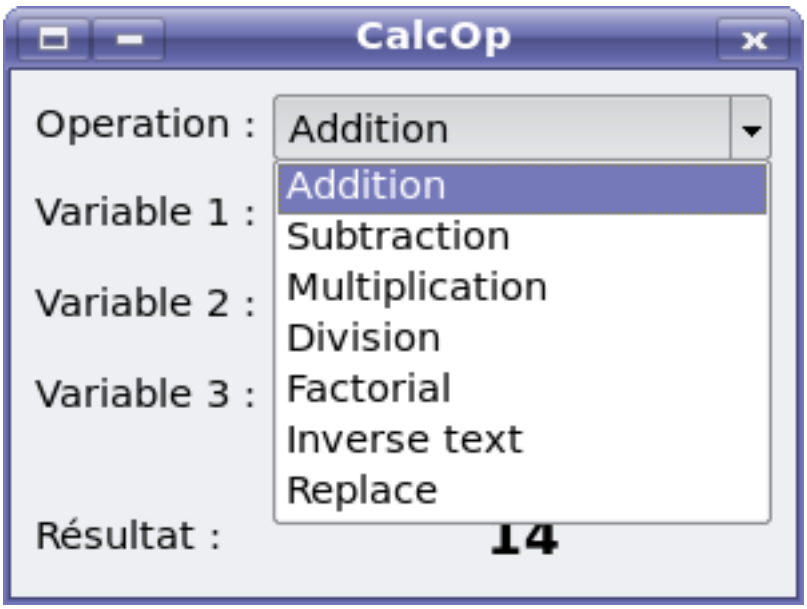
Grâce à Qt, la réalisation d'un système de plug-ins multi-plateforme est relativement simple : **la documentation Qt sur le système de plug-ins**.

II - CalcOp

Pour illustrer la création de plug-in, nous allons utiliser un exemple assez simple : un petit projet que j'ai nommé calcop et qui permet à partir de une, deux ou trois variables d'effectuer une opération au choix de l'utilisateur (par exemple des additions, multiplications, remplacement de caractères dans une chaîne...).







Ça ne sert à rien, mais le principal c'est que l'exemple est très simple, il permet donc de se concentrer sur le code lié à la gestion de plug-in. Le code source est livré avec deux plug-ins :

- un plug-in contenant des opérations mathématiques (addition, multiplication, division, soustraction, factorielle) ;
- un plug-in contenant des opérations sur les chaînes (inversion de sens et remplacement dans une chaîne).

Les sources du logiciel sont **téléchargeables**. Je vous conseille de les télécharger avant de vous lancer dans la lecture du tutoriel.

II-A - Le programme principal

Le programme principal est relativement simple. Je ne vais pas détailler tout le code, je me focaliserai sur les parties nécessaires à la mise en place de plug-ins.

La première chose à faire, c'est de définir un ensemble d'interfaces à nos plug-ins : ce sont des classes avec seulement des fonctions virtuelles pures utilisées pour dialoguer entre l'application et les plug-ins. Il est possible bien entendu de définir plusieurs interfaces pour différents types de plug-in mais, dans notre cas, nous n'en définissons qu'une seule : OperationInterface.

```
class OperationInterface
{
public:
 virtual ~OperationInterface() {}
 virtual QStringList operationList() = 0;
 virtual bool canCalculate( QString opName ) = 0;
 virtual int numVariable( QString opName ) = 0;
 virtual QString calculate( QString opName, QStringList variableList ) = 0;
};
Q_DECLARE_INTERFACE(OperationInterface, "org.nikikko.CalcOp.OperationInterface/1.0")
```

La macro **Q_DECLARE_INTERFACE** permet d'indiquer lors de la compilation (à l'outil moc, le précompilateur Qt) que cette classe est une interface.

Attention : le deuxième élément de la macro "org.nikikko.CalcOp.OperationInterface" doit obligatoirement se finir par le nom de la classe (je me suis fait avoir au début).

Le reste du code, notamment la classe OperationManager, sera détaillé par la suite.

II-B - Création d'un plug-in

La création du plug-in est relativement simple. Il suffit de créer une classe qui hérite de notre interface OperationInterface. Voici par exemple le code pour le plug-in mathématique :

```
#include <QObject>
#include <operationinterface.h>
class MathPlugin : public QObject, public OperationInterface
{
Q_OBJECT
Q_INTERFACES(OperationInterface)
public:
 QStringList operationList();
 bool canCalculate( QString opName );
 int numVariable( QString opName );
 QString calculate( QString opName, QStringList variableList );
};
```

Deux remarques importantes :

- la classe de base d'un plug-in (qui hérite d'une classe d'interface) doit toujours hériter de QObject ;
- toutes les fonctions de l'interface doivent être définies.

Note : il est possible dans un plug-in d'avoir une classe qui hérite de plusieurs interfaces, pour cela voir l'exemple plugandpaint de Qt.

Et dans le fichier source :

```
Q_EXPORT_PLUGIN2(calcop_math, MathPlugin)
```

La macro **Q_EXPORT_PLUGIN2** indique au compilateur qu'il faut exporter cette classe et que cette classe est le point d'entrée du plug-in. Pour faire simple, une classe exportée est une classe qui est visible à l'extérieur de la bibliothèque (donc par le programme principal).

Enfin, il faut indiquer dans le fichier .pro que l'on souhaite créer un plug-in. Voici le fichier .pro correspondant au plug-in calcop_math :

```
TEMPLATE    = lib
CONFIG     += release plugin
INCLUDEPATH += ../../src
HEADERS     = mathplugin.h
SOURCES     = mathplugin.cpp
TARGET     = calcop_math
DESTDIR     = ../../bin/plugins
```

Voici quelques explications :

- TEMPLATE = lib : création d'une bibliothèque ;
- CONFIG += plugin : cette bibliothèque est un plug-in ;
- TARGET = calcop_math : ce plug-in s'appelle calcop_math (même nom que dans la macro Q_EXPORT_PLUGIN2) ;
- DESTDIR = ../../bin/plugins : mettre ce plug-in dans le bon répertoire (répertoire des plug-ins de l'application).

Voilà, c'est relativement simple à faire et, en plus, c'est multiplateforme.

II-C - Chargement des plug-ins dans l'application

Dernière étape, c'est le chargement des plug-ins dans l'application. Tout ce qui concerne la gestion des plug-ins se fait à travers la classe OperationManager dans le programme principal. Cette classe charge les plug-ins et s'occupe d'effectuer les appels nécessaires aux fonctions des plug-ins lors du calcul.

Pour me faciliter la vie, j'ai décidé d'en faire un **singleton** : c'est pour ça que le constructeur est privé et que la classe comporte une fonction membre statique instance. Mais le code le plus intéressant se trouve dans le constructeur de cette classe :

```
QDir pluginsDir = QDir(qApp->applicationDirPath());
pluginsDir.cd("plugins");
foreach (QString fileName, pluginsDir.entryList(QDir::Files)) {
 QPluginLoader loader(pluginsDir.absoluteFilePath(fileName));
 QObject *plugin = loader.instance();
 if (plugin)
 {
  OperationInterface * op = qobject_cast<OperationInterface *>(plugin);
  if (op)
  {
   m_operationList << op;
  }
 }
}
```

Nous avons trois actions importantes dans ce code :

- déplacement dans le répertoire de plug-in : les deux premières lignes vont nous permettre d'aller dans le répertoire des plug-ins ;
- chargement du plug-in avec la classe **QPluginLoader** : on vérifie que le plug-in est bien compatible avec la version du logiciel et de Qt et crée une instance de l'objet racine (qui est une classe héritée de l'interface, dans notre cas ce sera une instance de MathPlugin) ;
- vérification de la compatibilité du plug-in : de quelle interface hérite-t-il (ici, OperationInterface) ?

Si tout se passe bien, on enregistre le plug-in dans la liste des opérations. Et voilà, c'est simple, clair et propre

III - Conclusion

La classe QPluginLoader est au c ur de ce code : c'est elle qui va se charger de créer une instance du plug-in. Au passage, l'instance créée, que l'on récupère à l'aide de la fonction instance(), est statique : si l'on utilise le même code dans une autre fonction, l'instance renvoyée sera toujours la même (c'est-à-dire le même pointeur).

Merci à **mahefasoa** pour sa relecture !