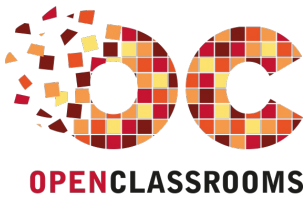
Qt - Création de plugins

Par EPonix



www.openclassrooms.com

Dernière mise à jour le 25/05/2011

Sommaire

Qt - Création de plugins

Par  EPonix

Mise à jour : 25/05/2011
Difficulté : Difficile

Ce tutoriel a pour but de vous faire découvrir les plugins avec Qt.

Il demande quelques connaissances en POO, dont l'abstraction de classe et montre un petit bout de code utilisant les templates. Si vous ne connaissez pas cela, ou si vous n'y êtes pas habitués, vous pouvez lire le tutoriel de Nanoc qui explique bien quelques principes avancés de la POO (en C++).

Dans un premier temps, j'introduirai le principe des plugins, puis dans un second temps, vous aurez droit à une grande partie théorique. Pour finir et pour s'entraîner un petit peu, vous disposerez un mini TP sur la création d'un programme utilisant des plugins.
Sommaire du tutoriel :

Introduction

Définition

En informatique, un plugin est un moyen de rajouter de nouvelles fonctionnalités à un logiciel de base. C'est pour cela qu'on appelle aussi les plugins « modules » ou « greffons ».
Ces modules sont en général utilisés pour qu'un programme puisse évoluer facilement. De plus, d'autres personnes voulant aider le logiciel peuvent à leur tour en créer pour ajouter de nouvelles fonctionnalités. L'un des cas les plus connus reste le célèbre navigateur Internet Mozilla Firefox.

Où et pourquoi ?

On peut rendre un programme totalement modulaire mais il ne faut pas perdre de vue que dans certains cas, cela peut se révéler complètement inutile. En effet, nous verrons par la suite qu'un plugin doit avoir une sorte de définition. Plus on veut rendre une partie de logiciel modulaire et plus la définition doit être floue, abstraite. Mais si elle l'était trop, cette partie n'aurait plus de but et ne servirait donc plus.

Il faut alors un but minimum pour qu'un système de plugins soit rentable. On peut pourtant en mettre dans beaucoup de programmes et votre seule limite est votre imagination. En effet, l'utilisation des plugins peut aller de la simple boîte de dialogue à un système complet utilisant lui aussi des plugins.

Théorie

Pour créer un plugin, il suffit de créer une définition puis de l'implémenter. Cependant, un plugin seul n'est pas très utile.
Il faut donc aussi modifier la base du programme en lui donnant un moyen de charger les plugins.

Créer une définition

Comme dit plus haut, un plugin doit avoir un but minimal et une sorte de définition.
La définition est la classe qui lie le projet mère à ses plugins. En effet, pour pouvoir concevoir un plugin, il y aura besoin de cette définition. Et pour que le projet utilise les plugins, il a besoin de savoir comment ils fonctionnent.
La définition est donc dans les deux projets (projet mère et projet de chaque plugin).

Cette définition est une simple interface, c'est-à-dire une classe abstraite contenant une liste de fonctions que tous les plugins du même type auront.

Par exemple la classe Animal qui permettra de créer des plugins de type Animal aura comme méthodes :

- `void manger(const Nourriture &repas) ;`
- `void bouger(const Position &destination) ;`
- `void attaquer(const Cible &cible) ;`
- ...

Plus le but est abstrait et moins la définition contient de méthodes, car plus de choses devraient « coller » avec elle.

Vous avez peut-être remarqué que cela ressemble à l'utilisation de la POO classique. En effet, pour concevoir plusieurs objets qui se ressemblent, on crée des classes de base abstraites, puis on les spécialise. Pour créer une interface, il faut que toutes les méthodes soient virtuelles pures.

Pour ceux qui ne savent pas comment les définir, voici un exemple :

Code : C++

```
class MaClass {
    public:
 virtual ~MaClass() {}

        virtual void maFonction() = 0; // On met = 0 pour dire
qu'elle est virtuelle pure.
        virtual void maFonctConst() const = 0; // On met le =0 après
le const
};
```

On utilise le mot-clé **`virtual`** pour dire que la fonction est virtuelle. Ensuite, pour dire qu'on ne peut pas la définir, c'est-à-dire qu'elle est pure, on met `= 0` à la fin de la déclaration. Cela rend notre classe abstraite car une méthode (ou plus) n'est pas définie.

Vous pouvez apercevoir que l'on ne rend pas le destructeur virtuel pur. En effet, il n'est pas possible pour la classe fille de choisir ce que fera le destructeur de la classe mère.

Cependant, comme vous l'avez remarqué, au tout début, j'ai utilisé le mot *interface* et non *classe* directement. En effet, Qt veut que nous déclarions explicitement une interface. Pour cela, on utilise la macro-définition **Q_DECLARE_INTERFACE**.

Elle prend en paramètre deux choses.
La première est le nom de la classe interface et la seconde est l'identifiant de l'interface sous forme de chaîne de caractères. Attention, celle-ci doit finir par le nom de la classe et être unique.
En général, cette macro-définition est située juste après la déclaration de la classe.

Si vous voulez créer une interface dans un namespace, il faut que la macro-définition soit appelée hors du namespace. Pour le nom de la classe, on utilise l'opérateur de portée « :: » (par exemple : « MonNamespace::MaClasse »).

Voici un exemple de deux interfaces :

Code : C++

```
#ifndef __INTER_H__
```

```
#define __INTER_H__
#include <QtPlugin>
#include "EtreVivant.h" // C'est une classe comme les autres (qui
sont abstraites).

 class Animal : public EtreVivant {
  public:
   virtual ~Inter1() { }

   virtual void mange(const EtreVivant &proie) = 0;
   virtual void attaquer(const EtreVivant &victime) = 0;

  // On peut mettre des variables.
  protected:
   int m_Faim;
   int m_Vie;
 };
 Q_DECLARE_INTERFACE(Animal, "Mon programme.Animal")

 // Voici un exemple avec un namespace.
 namespace Out { // Un namespace qui contient des classes de sorties
pour le programme. Ce n'est qu'à titre d'exemple, bien sûr.
  class Console { // Ici on a l'interface dans le namespace.
   public:
    virtual Console() {}

    virtual print(const std::string &str) = 0;
  };
 }
 Q_DECLARE_INTERFACE(Out::Console, "Mon programme.Console") // On
doit utiliser l'opérateur de portée :: pour accéder à notre
interface.

#endif
```

Il ne faut pas mettre de point-virgule après la macro-définition Q_DECLARE_INTERFACE. Il en est de même pour toutes les macro-définitions que vous rencontrerez dans ce cours.

Vous connaissez maintenant presque tout sur la création d'interfaces avec Qt. Il ne reste plus que la création du plugin en lui-même, qui est très simple, et le chargeur.

Création du plugin

Un plugin est avant tout un objet qui respecte une définition. Donc pour cela, il suffit de le faire hériter de l'interface.
Puisqu'un plugin est aussi un objet Qt, alors il doit hériter de QObject (directement ou indirectement).

Voici donc pour l'instant votre définition du plugin :

Code : C++

```
class MyPlugin : public QObject, public MyInter { };
```

Le fait de ne pas dériver votre plugin de QObject avant toute autre classe entraînera forcément une erreur. Veillez donc à bien faire hériter votre interface dans un premier temps de QObject, puis de toute autre classe.

Il faut ensuite dire à Qt que cette classe n'est pas comme les autres. Tout d'abord, c'est un QObjet, donc on peut mettre la macro-définition **Q_OBJECT**. Puis il faut signaler à Qt que l'on utilise une ou des interfaces. Pour cela, on utilise la macro-définition **Q_INTERFACES**. Elle attend la liste des interfaces séparées par des espaces.

Notre prototype de la classe est donc pour l'instant :

Code : C++

```
class Chien : public QObject, public Animal { // Un chien est un
animal, il dérive donc de l'interface Animal.
 Q_OBJECT
 Q_INTERFACES(Animal)

 // Si notre plugin dérivait de plusieurs interfaces, il aurait
fallu donner la liste à la macro-définition avec des espaces entre
chaque nom.
 // Comme cela :
 // Q_INTERFACES(MyInter1 MyInter2 ...)
};
```

Pour pouvoir utiliser un plugin, il faut qu'il soit concret. En effet, un plugin est avant tout une classe. Or si une classe est abstraite, elle ne peut pas être instanciée.
Puisque notre plugin hérite d'une interface abstraite, il faudra donc définir toutes les méthodes virtuelles pures.

Si notre plugin a besoin de plus de fonctions, on peut en rajouter. Cependant, le programme de base ne pourra toucher qu'aux méthodes définies dans l'interface. En effet, il ne connaît que l'interface, il ne sait donc pas si un plugin a la fonction machin().

Dans l'implémentation, tout se passe comme d'habitude, mais il faut rajouter la macro-définition **Q_EXPORT_PLUGIN2(nomPlugin, nomClass)**.
Le nom du plugin est celui que l'on définira dans le *.pro* du plugin. Le nom de la classe est celui de la classe d'entrée du plugin, donc celle qui hérite de l'interface. On peut donc mettre plusieurs classes dans un plugin.

Voici la macro-définition pour l'exemple :

Code : C++

```
Q_EXPORT_PLUGIN2(nom_plugin, MyPlugin) // Pas de point-virgule après
la macro-définition.
```

Il faut maintenant changer le *.pro* du projet. En effet, notre « projet » ne contient pas de main, donc le compilateur ne trouvera pas de point d'entrée s'il en cherche. Il faut donc le signaler en lui disant que l'on veut faire une bibliothèque. Pour cela, on utilise la variable TEMPLATE du *.pro* comme ceci :

Code : C++

```
TEMPLATE = lib
```

Il existe différents types de bibliothèques : les dynamiques, les statiques et aussi les plugins. On utilise alors CONFIG pour montrer à Qt que c'est un plugin :

Code : C++

```
CONFIG += plugin
```

Si un programme est en débogage, alors les plugins doivent aussi être en mode débogage, de même pour le release. On peut donc dire le type de compilation en rajoutant « release » ou « debug » à CONFIG.

Pour finir, on définit le nom de la cible (c'est-à-dire le plugin) avec TARGET. C'est aussi le nom du plugin qu'on avait mis dans la

macro-définition Q_EXPORT_PLUGIN2 :

Code : C++

```
TARGET = nom_plugin
```

On a donc maintenant un *.pro* qui permet de compiler notre plugin. Il ne reste plus que la partie sur la création d'un chargeur de plugins et la partie théorique est finie.

Pour qu'il n'y ait pas d'ambiguïtés, sachez qu'un plugin ne se compile pas avec le projet mère. Il utilise donc un fichier *.pro* différent. Mais il peut utiliser des fichiers du projet mère (l'interface, notamment). De même, un plugin a son propre fichier projet et ne le partage pas avec un autre plugin.

Faire un chargeur de plugins

Il est relativement simple de charger un plugin car Qt a une classe prête pour cela : **QPluginLoader**.
Pour en créer un, il suffit juste de donner le chemin vers le plugin.
Ensuite, on peut récupérer un pointeur vers le plugin avec la méthode **QPluginLoader::instance()**. Elle retourne un **QObject***, donc il faudra la réinterpréter en MyPlugin*, par exemple.
Pour la réinterpréter, soit on utilise le cast C++ `reinterprete_cast<T>(obj)` , soit on utilise un cast défini par Qt : `qobject_cast<T>(obj)` .
Puisque l'on utilise Qt et que **qobject_cast** utilise **reinterprete_cast** avec un plus, le mieux est d'utiliser **qobject_cast**. Ce cast demande simplement le type de transformation comme argument template et prend aussi l'objet à réinterpréter.

Cela se résume par ce bout de code :

Code : C++

```
QPluginLoader loader("./cheminVersMonPlugin"); // On charge le
plugin en lui donnant juste le chemin.
if(QObject *plugin = loader.instance()) { // On prend l'instance de
notre plugin sous forme de QObject. On vérifie en même temps s'il
n'y a pas d'erreur.
    MyPlugin* myPlugin = qobject_cast<MyPlugin *>(plugin); // On
réinterprète alors notre QObject en MyPlugin
}
```

Ce code fonctionne et permet de charger un plugin, mais il est assez basique. Un système de plugin doit permettre à une application d'évoluer facilement, elle doit donc savoir si un plugin a été ajouté. Si on met tous les plugins du même type dans le même dossier, alors on peut utiliser **QDir** pour lire ce dossier et donc connaître tous les plugins.

Déjà, il faut créer un **QDir** en lui donnant le dossier du programme pour être sûr de connaître le bon chemin vers le plugin. Justement, la fonction `qApp->applicationDirPath()` retourne le chemin vers l'application.

Ensuite, il faut déplacer le **QDir** dans le dossier du plugin avec la méthode « cd ». Si vous utilisez la console, vous verrez que l'utilisation de cette méthode est identique à la commande « cd ».

Voici pour l'instant le code :

Code : C++

```
QDir plugDir = QDir(qApp->applicationDirPath()); // On place le QDir
dans le dossier de notre exécutable.
plugDir.cd("./cheminSecretVersLeTresorDuTuto"); // Puis on le
déplace dans le dossier des plugins. Je ne pense pas que le chemin
de votre plugin soit celui-ci.
```

Il faut maintenant boucler avec la liste des fichiers se trouvant dans le répertoire. **QDir::entryList()** retourne la liste des fichiers avec comme arguments la define **QDir::Files** qui est un filtre : il ne prendra que les fichiers du dossier, les répertoires ne seront pas listés. Elle retourne un QStringList.
Puisqu'un QStringList est une List, il est alors possible d'utiliser un foreach. Un foreach est une structure de Qt qui permet de faire des actions **pour chaque** élément d'une liste. D'où son nom.
Cela revient au même qu'une boucle itérative, mais je trouve qu'elle prend moins de place.

Voici comment on utilise une classe foreach :

Code : C++

```
// T est un type défini.

T unElement;
QList<T> listElement;

foreach(unElement, listElement) { // On prend chaque élément de
listElement que l'on met l'un après l'autre dans la variable
unElement.
    // Les actions.
}
```

On se servira aussi de **QDir::absoluteFilePath(QString)** qui nous permet d'avoir le chemin absolu vers le fichier.

Voici donc notre chargeur de plugins :

Code : C++

```
QList<MyPlugin *> m_LsPlugin; // On crée une liste de MyPlugin* qui
contiendra nos plugins.
QDir plugDir = QDir(qApp->applicationDirPath()); // Comme avant, on
crée un QDir.
plugDir.cd("./Chemin"); // On se déplace encore.

foreach(QString file, plugDir.entryList(QDir::Files)) { // Puis on
utilise le foreach.
 QPluginLoader loader(plugDir.absoluteFilePath(file)); // On fait
ensuite la même chose que pour un seul plugin.
 if(QObject *plugin = loader.instance())
  MyPlugin* myPlugin = qobject_cast<MyPlugin *>(plugin);
  m_LsPlugin.push_back(myPlugin); // Vous pouvez maintenant les
stocker ou directement les utiliser.
 }
}
```

Vous avez enfin un chargeur de plugins. Vous pouvez soit utiliser une fonction soit une classe singleton (pas la peine d'avoir 50 instances) pour le chargeur.

Pratique : mini TP

Maintenant que vous savez comment faire pour réaliser des plugins avec Qt, je pense qu'il ne reste plus qu'à s'entraîner.
Cette dernière partie sera donc sous la forme d'un mini TP où le but sera de créer une application utilisant les plugins.

Pour ne pas rendre l'exemple trop compliqué, il sera ~~inutile~~ simple.
En effet, le sujet de ce cours n'est pas la création d'une grosse application. C'est pourquoi, vous aurez au maximum à créer une fenêtre.

J'aurais même pu vous faire faire un TP en console car Qt n'est pas qu'une bibliothèque de fenêtrage. Et vous le voyez justement

dans ce cours.

Le programme principal de ce TP sera donc une simple fenêtre avec un menu ne contenant rien par défaut.

Quand vous l'aurez fini, je pense que vous trouverez que la capacité de votre programme est très limitée.

Pour mettre un but à notre programme, il faudrait alors remplir ce menu. Mais cela reviendrait à changer complètement le programme. Si on utilise les plugins dans ce cas, il suffirait de changer ou de les ajouter dans un dossier sans recompiler le projet. Il faudra simplement un chargeur de plugin mais une fois installé, on n'a plus besoin de toucher à l'application de base. De plus, d'autres pourront vous aider à développer votre application.

Je vous l'accorde, ce n'est pas vraiment le choix le plus judicieux, mais c'est pour que cela reste un exemple simple puisque je ne veux pas rendre le TP trop difficile.

Dans un premier temps, il faut donc d'abord créer cette fenêtre.
Ensuite, puisque c'est vide pour l'instant, je vous propose de créer des plugins pour une action au hasard. Dans le TP ça sera un plugin simple qui ouvre une boîte de dialogue contenant le cri d'un animal (pour reprendre l'exemple de la partie théorique).
Ce n'est pas très utile mais au moins, c'est facile. Cependant, une fois que vous aurez bien compris comment utiliser les plugins, vous pourrez créer des plugins beaucoup plus complexes (comme la résolution d'équation dans les complexes).

Pour que vous ne vous égariez pas dans la jungle de plugins à créer, voici une petite liste des cris des animaux à faire (vous pouvez en faire d'autre si vous voulez) :

- aboiement du chien ;
- miaulement du chat ;
- hennissement du cheval ;
- hurlement d'une hystérique.

Analyse du problème

Cette partie vous permet, si vous n'y arrivez pas, à mieux voir comment on pourrait résoudre ce TP.

Il y a plusieurs solutions et la mienne n'est ni la meilleure ni la moins bonne (j'espère) ; donc si vous faites le programme d'une autre façon, ce n'est pas grave tant que ça fonctionne.

La création de la fenêtre principale ne demande pas énormément d'analyse car c'est une simple fenêtre qui contient un menu.
Pour remplir ce menu, je pense que l'on peut stocker les plugins dans un tableau puis le boucler et y mettre les actions.

Il faut maintenant choisir la définition de nos plugins. Un animal a déjà un nom que l'on pourra mettre dans le menu.
Il doit aussi avoir un cri qui apparaîtra quand on cliquera sur son nom.

Cela nous donne donc deux informations pour notre définition :

- `QString getName() const`
- `void brawl() const [slot]`

Maintenant vous devez avoir tous les éléments pour créer ce programme. Il est donc temps de travailler un peu.

Correction

Comme la création de l'application n'est pas très difficile, je vous donne la correction directement.

Header

Code : C++

```
#ifndef MYWIDGET_H
#define MYWIDGET_H
```

```
#include <QtGui/QMainWindow>

 class MyWidget : public QMainWindow { // On crée un QMainWindow.
  Q_OBJECT

  public:
   MyWidget(QWidget *parent = 0);
   ~MyWidget();
 };
#endif // MYWIDGET_H
```

L'implémentation

Code : C++

```
#include "mywidget.h"
#include "PluginInter.h" //Pour que notre système puisse savoir la
forme de nos plugins

#include <QApplication> //Pour l'utilisation de qApp plus tard
#include <QMenu>
#include <QMenuBar>

MyWidget::MyWidget(QWidget *parent) : QMainWindow(parent) {
 QMenu* mCri = menuBar()->addMenu("Cri"); // On ajoute le menu.
}

MyWidget::~MyWidget() { }
```

Je ne mets pas le *main.cpp* car il est trop simple et trop banal. Comme vous le voyez, je n'ai fait que le minimum (et ça se voit à l'exécution).

Le chargeur de plugin

Pour le chargeur de plugin, j'ai remarqué que l'on n'utilise que deux manières différentes pour charger les plugins : quand on veut un plugin précis ou quand on veut tous les plugins d'un dossier. J'ai donc décidé de créer ici des méthodes statiques.
Ici une seule méthode nous servira mais au moins, vous pourrez réutiliser ce code. 

Puisque ces méthodes devront être utilisées avec n'importe quel plugin, il faudra utiliser les templates.

Tout d'abord, voici la correction de la première méthode (avec un plugin précis).
Si l'utilisateur demande un plugin de type T, il faut lui renvoyer un pointeur vers le plugin, soit un T*.
On utilise donc la méthode dans la partie théorie mais pas de QDir, cette fois. En effet, l'utilisateur devra donner le chemin vers le plugin. Donc c'est à lui d'utiliser qApp->applicationPath() (je le vois comme ça). Vous pouvez demander à l'utilisateur un chemin relatif par rapport à l'exécutable si vous voulez.

Ensuite on écrit le même code que dans la partie théorique. En effet, on crée un QPluginLoader. Puis on prend l'instance. Cette instance est alors retournée sous forme réinterprétée.

Donc cela fait :

Code : C++

```
// C'est une méthode qui demande un argument template pour éviter
de créer un chargeur par type de plugin.
template<typename T> T* pluginByName(const QString& fileName) {
 QPluginLoader loader(fileName); // On charge le plugin.
 QObject *plugin = loader.instance(); // On prend l'instance.
 return qobject_cast<T*>(plugin); // Et on retourne le plugin casté
.
```

```
}
```

Comme vous le voyez, il y a une petite différence avec le code de la partie théorique : il n'y a pas de vérification. Si on utilise cette méthode, il faudra donc vérifier si le plugin est valide.
On aurait aussi pu faire la vérification dans la méthode et si le plugin n'est pas bon, cela aurait retourné une exception.

La seconde méthode, celle qui prend tous les plugins d'un dossier, doit prendre chaque plugin d'un dossier. Cela paraît bête de dire ça, mais la phrase montre qu'une partie est identique à notre première méthode. Pour éviter de taper le même code et d'avoir des répétitions, il est donc possible de réutiliser la première méthode.

Dans la seconde méthode, il faut boucler et prendre les fichiers d'un dossier avec QDir.
Il est très similaire au code donné dans la partie théorie, voici donc directement le code :

Code : C++

```
// Toujours une méthode avec un argument template pour éviter qu'il
y en ait 50.
template<typename T> QList<T*> pluginByDir(const QString& dir) {
 QList<T*> ls; // On crée la liste où seront mis tous les plugins
valides.
 QDir plugDir = QDir(dir);
 foreach(QString file, plugDir.entryList(QDir::Files)) { // On prend
la liste des fichiers.
  if(T* plugin =
PluginLoader::pluginByName<T>(plugDir.absoluteFilePath(file))) // On
vérifie si le plugin existe.
   ls.push_back(plugin); // On l'ajoute à la liste si oui.
 }

 return ls;
}
```

Ici on vérifie si le plugin existe avant de le mettre dans la QList. Il est, comme suggéré au-dessus, possible de mettre un else et d'y lancer une exception.

J'ai mis ces méthodes dans un namespace appelé PluginLoader, mais vous pouvez les laisser dans l'espace global (ou les mettre dans une classe).

Voici le code de ma classe PluginLoader :

Code : C++

```
#ifndef PLUGIN_LOADER_H
#define PLUGIN_LOADER_H
    // On n'oublie pas les inclusions.
#include <QList>
#include <QPluginLoader>
#include <QObject>
#include <QString>
#include <QDir>
#include <QObject>

    namespace PluginLoader {
        // C'est une méthode qui demande un argument template pour
éviter de créer un chargeur par type de plugin.
        template<typename T> T* pluginByName(const QString&
fileName) {
            QPluginLoader loader(fileName); // On charge le plugin.
            QObject *plugin = loader.instance(); // On prend
l'instance.
            return qobject_cast<T*>(plugin); // Et on retourne le
plugin casté.
```

```
        }

        // Toujours une méthode avec un argument template pour
éviter qu'il y en ait 50.
        template<typename T> QList<T*> pluginByDir(const QString&
dir) {
            QList<T*> ls; // On crée la liste où seront mis tous
les plugins valides.
            QDir plugDir = QDir(dir);
            foreach(QString file, plugDir.entryList(QDir::Files)) {
// On prend la liste des fichiers.
                if(T* plugin =
PluginLoader::pluginByName<T>(plugDir.absoluteFilePath(file))) // On
vérifie si le plugin existe.
                    ls.push_back(plugin); // On l'ajoute à la liste
si oui.
            }

            return ls;
        }
    }
#endif
```

Vous avez maintenant de beaux chargeurs de plugins qui pourront être utilisés dans n'importe quel programme.
Il est temps maintenant de créer la définition du plugin.

La définition du plugin

Comme dit plus haut, nos plugins auront simplement une méthode pour avoir son nom et un slot brawl().
Il est donc très simple de créer la définition :

Code : C++

```
#ifndef PLUGIN_INTER_H
#define PLUGIN_INTER_H
#include <QtPlugin>

 class QString;

 class PluginInter : public QObject {
  Q_OBJECT //Pour avoir les slots et les signaux

  public:
   virtual ~PluginInter() {} // Le destructeur ne fait rien ici.

   virtual QString getName() const = 0;

  public slots:
   virtual void brawl() = 0;

  protected:
   QString m_Name; // On peut très bien mettre une variable dans
une interface. Ici c'est le nom.
 };
 Q_DECLARE_INTERFACE(PluginInter, "MyProgramme.PluginInter") // On
utilise la macro-définition pour déclarer une interface.
#endif // PLUGIN_INTER_H
```

Comme dans la partie théorique, toutes les méthodes sont virtuelles pures et on utilise la macro-définition **Q_DECLARE_INTERFACE**.

Maintenant que l'on a notre définition, nous pouvons rajouter dynamiquement les actions au menu.
Il faut prendre la liste des plugins avec PluginLoader::pluginByDir() puis rajouter l'action du plugin.

Voici le code qui permet de faire cela :

Code : C++

```
foreach(PluginInter* plugin,
PluginLoader::pluginByDir<PluginInter>(qApp->applicationDirPath() +
"/plugins/")) { // On utilise notre super fonction pour récupérer
la liste des plugins.
 QAction* action = mCri->addAction(plugin->getName()); //On crée
ensuite l'action avec le nom du plugin.
 QObject::connect(action, SIGNAL(triggered()), plugin,
SLOT(brawl())); // Et on connecte l'action au slot du plugin.
}
```

Voici quelques explications sur le code.
À chaque élément contenu dans le tableau de plugins obtenu par notre méthode, on crée un QAction avec le nom du plugin.
Puis on connecte le signal triggered() de l'action au slot du plugin.

Avouez que, une fois traduit, c'est simple. 

Les plugins

Il ne reste que les plugins. Les plugins héritent de notre interface et doivent être concrets. Il faudra donc définir toutes les méthodes virtuelles pures. Les méthodes deviendront alors seulement virtuelles.

Je ne donnerai l'exemple que pour le chien car seul le nom ou le hurlement change.
Dans la déclaration de la classe, il faut bien utiliser les macro-définitions Q_OBJET et Q_INTERFACES.

Voici le code de la déclaration :

Code : C++

```
#ifndef CHIEN_H
#define CHIEN_H
#include "../PluginInter.h" // On doit avoir accès à l'interface.

 class Chien : public PluginInter { // On le fait hériter de
l'interface. Pas besoin de le faire hériter de QObject puisque
PluginInter est déjà un QObject
  Q_OBJECT
  Q_INTERFACES(PluginInter) // On définit l'interface.

  // On définit toutes les méthodes de l'interface.
  public:
   Chien();
   virtual ~Chien();

   virtual QString getName() const; // Les méthodes ne sont plus
abstraites.

  public slots:
   virtual void brawl();
 };
#endif // CHIEN_H
```

L'implémentation dans cet exemple reste assez simple. Il faut quand même ne pas oublier d'utiliser la macro-définition Q_EXPORT_PLUGIN2.

Code : C++

```
#include "Chien.h"

#include <QMessageBox>

Chien::Chien() {
 m_Name = "Chien"; // On définit le nom de l'animal. On peut donc
mettre chat ici.
}

Chien::~Chien() { }

QString Chien::getName() const {
 return m_Name;
}

void Chien::brawl() {
 QMessageBox::information(0, "Cri du chien", "OUAF"); // Pareil que
dans le construction, on peut mettre « MIAOU » dans le QMessageBox.
}

Q_EXPORT_PLUGIN2(plug_chien, Chien)
```

Il suffit juste d'implémenter les fonctions comme d'habitude.

Il reste un dernier détail à régler qui n'est pas vraiment du code : le fichier projet du plugin.

Les fichiers .pro

Le fichier projet doit permettre à Qt de connaître les fichiers source utilisés. Ici nous en avons 3.
Il y a *Chien.cpp*, *Chien.h* et il ne faut pas oublier *PluginInter.h*.

Les fichiers *.cpp* sont mis dans la variable SOURCES. Les fichiers header sont mis dans la variable HEADERS.

Notre fichier projet ressemble à ceci :

Code : Autre

```
SOURCES += Chien.cpp
HEADERS += Chien.h \
HEADERS += ../PluginInter.h
```

Comme dit dans la partie théorique, il y a ensuite quelques petites choses à mettre pour montrer que c'est un plugin.

Il y a donc les variables TEMPLATE, CONFIG et TARGET à mettre.
TEMPLATE attend le type de projet. Un plugin est une lib.
Il faut ensuite préciser dans la configuration que c'est un plugin.
Enfin, il faut mettre le nom du plugin dans TARGET

On doit donc rajouter :

Code : Autre

```
TEMPLATE = lib
CONFIG += plugin
TARGET = plug_chien
```

Si vous faites plusieurs plugins, il faudra alors choisir un nom pour chacun.

Le fichier projet est donc à la fin :

Code : Autre

```
SOURCES += Chien.cpp
HEADERS += Chien.h
HEADERS += ../PluginInter.h

TEMPLATE = lib
CONFIG += plugin
TARGET = plug_chien
```

Récapitulatif

Cet exemple ne sert à rien mais en général, un plugin est plus compliqué et je voulais surtout vous montrer comment faire un plugin avec Qt.

Pour ceux qui sont intéressés, Qt a fait un programme d'exemple utilisant des plugins. La source est disponible dans la documentation et le programme s'appelle Plug and Paint.
Voici le lien vers l'exemple Plug And Paint.

Il est temps de redonner tout le code.

Voici les sources du projet mère.

D'abord le tour de la classe principale de la fenêtre :

mywidget.h

Code : C++

```
#ifndef MYWIDGET_H
#define MYWIDGET_H
#include <QtGui/QMainWindow>

 class MyWidget : public QMainWindow { // On crée un QMainWindow.
  Q_OBJECT

  public:
   MyWidget(QWidget *parent = 0);
   ~MyWidget();
 };
#endif // MYWIDGET_H
```

mywidget.cpp

Code : C++

```
#include "mywidget.h"
#include "PluginLoader.h"
#include "PluginInter.h"

#include <QApplication>
#include <QMenu>
#include <QMenuBar>

MyWidget::MyWidget(QWidget *parent) : QMainWindow(parent) {
 QMenu* mCri = menuBar()->addMenu("Cri"); // On ajoute le menu.

 // On charge les plugins.
 foreach(PluginInter* plugin,
PluginLoader::pluginByDir<PluginInter>(qApp->applicationDirPath() +
"/plugins/")) { // On utilise notre super fonction pour récupérer
la liste des plugins.
  QAction* action = mCri->addAction(plugin->getName()); // On crée
ensuite l'action avec le nom du plugin.
  QObject::connect(action, SIGNAL(triggered()), plugin,
```

```
SLOT(brawl())); // Et on connecte l'action au slot du plugin.
 }
}

MyWidget::~MyWidget() { }
```

Puis vient le tour du chargeur de template :

PluginLoader.h

Code : C++

```
#ifndef PLUGIN_LOADER_H
#define PLUGIN_LOADER_H
    // On n'oublie pas les inclusions.
#include <QList>
#include <QPluginLoader>
#include <QObject>
#include <QString>
#include <QDir>
#include <QObject>

    namespace PluginLoader {
        // C'est une méthode qui demande un argument template pour
éviter de créer un chargeur par type de plugin.
        template<typename T> T* pluginByName(const QString&
fileName) {
            QPluginLoader loader(fileName); // On charge le plugin.
            QObject *plugin = loader.instance(); // On prend
l'instance.
            return qobject_cast<T*>(plugin); // Et on retourne le
plugin casté.
        }

        // Toujours une méthode avec un argument template pour
éviter qu'il y en ait 50.
        template<typename T> QList<T*> pluginByDir(const QString&
dir) {
            QList<T*> ls; // On crée la liste où seront mis tous
les plugins valides.
            QDir plugDir = QDir(dir);
            foreach(QString file, plugDir.entryList(QDir::Files)) {
// On prend la liste des fichiers.
                if(T* plugin =
PluginLoader::pluginByName<T>(plugDir.absoluteFilePath(file))) // On
vérifie si le plugin existe.
                    ls.push_back(plugin); // On l'ajoute à la liste
si oui.
            }

            return ls;
        }
    }
#endif
```

Il ne manque plus que la définition du plugin :

PluginInter.h

Code : C++

```
#ifndef PLUGIN_INTER_H
#define PLUGIN_INTER_H
#include <QtPlugin>
```

```
class QString;

class PluginInter : public QObject {
 Q_OBJECT

 public:
  virtual ~PluginInter() {} // Le destructeur ne fait rien ici.

  virtual QString getName() const = 0;

 public slots:
  virtual void brawl() = 0;

 protected:
  QString m_Name; // On peut très bien mettre une variable dans
une interface. Ici c'est le nom.
 };
 Q_DECLARE_INTERFACE(PluginInter, "MyProgramme.PluginInter") // On
utilise la macro-définition pour déclarer une interface.
#endif // PLUGIN_INTER_H
```

Viennent maintenant les sources du plugin. Je ne remettrai pas l'interface car elle est juste au-dessus. Mais il ne faut pas oublier qu'elle fait aussi partie du plugin.

Chien.h

Code : C++

```
#ifndef CHIEN_H
#define CHIEN_H
#include "../PluginInter.h" // On doit avoir accès à l'interface.

 class Chien : public PluginInter { // On le fait hériter de
l'interface et de QObject.
  Q_OBJECT
  Q_INTERFACES(PluginInter) // On définit l'interface.

  // On définit toutes les méthodes de l'interface.
  public:
   Chien();
   virtual ~Chien();

   virtual QString getName() const; // Les méthodes ne sont plus
abstraites.

  public slots:
   virtual void brawl();
 };
#endif // CHIEN_H
```

Chien.cpp

Code : C++

```
#include "Chien.h"

#include <QMessageBox>

Chien::Chien() {
 m_Name = "Chien"; // On définit le nom de l'animal. On peut donc
mettre chat ici.
}
```

```
Chien::~Chien() { }

QString Chien::getName() const {
 return m_Name;
}

void Chien::brawl() {
 QMessageBox::information(0, "Cri du chien", "OUAF"); // Pareil que
dans le construction, on peut mettre MIAOU dans le QMessageBox.
}

Q_EXPORT_PLUGIN2(plug_chien, Chien)
```

Quelques idées d'exercices

Je ne peux pas donner des idées d'améliorations à faire puisque notre programme ne sert à rien.
Je peux cependant vous donner des idées d'exercices si vous voulez allez plus loin.

- Un logiciel de dessin avec des brush modulable. Cette idée vient de la documentation de Qt.
- Un éditeur de texte proposant une panoplie de plugins comme l'indentation du code, l'exécution de commande en console.
- Un programme dont le style est complètement personnalisable.

Il est assez dur de donner des types de logiciel POUR un plugin puisqu'en général, un plugin est un plus pour le logiciel qui peut aussi se greffer sur d'autres logiciels.

C'est pourquoi la plus grosse partie du travail dans la création de plugins est la conception.

Q.C.M.

Le premier QCM de ce cours vous est offert en libre accès.
Pour accéder aux suivants

Connectez-vous Inscrivez-vous

Quelle est le type de classe utilisé par Qt pour créer une définition ?

- ◯ Une abstraction.
- ◯ Une interface.
- ◯ Une virtuelle pure.
- ◯ La réponse D.

Quelles sont les arguments attendus par la macro-définition Q_EXPORT_PLUGIN2 ?

- ◯ Nom du plugin - Nom de la définition.
- ◯ Nom de la classe du plugin - Nom de la définition.
- ◯ Nom du plugin - Nom de la classe du plugin.
- ◯ Nom du plugin - Nom de la classe du plugin - Nom de la définition.

Quelle classe permet de récupérer l'instance d'un plugin ?

- ◯ QPlugin.
- ◯ QObject.
- ◯ QPluginLoader.

Correction !

Statistiques de réponses au QCM

On peut facilement penser que la création de plugins pour les logiciels est un vrai casse-tête à coder. C'est vrai que c'est un

casse-tête, mais seulement pour la conception car la programmation devient très facile avec Qt, comme vous avez pu le constater dans ce tutoriel.