13- XtLinguist

La traduction de programmes Qt est un processus bien pensé.
Qt suppose que les développeurs ne sont pas des traducteurs. Il suppose donc que ce sont 2 personnes différentes.
Tout a été fait pour que les traducteurs, même si ce ne sont pas des informaticiens, soient capables de traduire votre programme.

1- Préparer son code à une traduction
La toute première étape de la traduction consiste à écrire son code de manière adaptée, afin que des traducteurs puissent ensuite récupérer tous les messages à traduire.

1.1- Utilisez QString pour manipuler des chaînes de caractères

```
QString chaine = "Bonjour"; // Bon : adapté pour la traduction
char chaine[] = "Bonjour"; // Mauvais : inadapté pour la traduction
```

1.2- Faites passer les chaînes à traduire par la méthode tr()

```
quitter = new QPushButton("&Quitter");
```

Vous rédigez donc les textes de votre programme dans votre langue maternelle (ici le français) en les entourant d'un tr().

```
quitter = new QPushButton(tr("&Quitter"));
```

2- Créer les fichiers de traduction .ts
Nous avons maintenant un programme qui fait appel à la méthode tr() pour désigner toutes les chaînes de caractères qui doivent être traduites.
Il va falloir éditer le fichier .pro. Celui-ci se trouve dans le dossier de votre projet et a normalement été généré automatiquement par Qt Creator.

```
#############################################################################
# Automatically generated by qmake (2.01a) ven. 23. mai 16:31:10 2011
#############################################################################

TEMPLATE = app
TARGET =
DEPENDPATH += .
INCLUDEPATH += .

# Input
HEADERS += MainWindow.h Scene.h
SOURCES += MainWindow.cpp Scene.cpp main.cpp
```

Rajoutez à la fin une directive TRANSLATIONS en indiquant les noms des fichiers de traduction à générer. Ici, nous rajoutons un fichier pour la traduction anglaise, et un autre pour la traduction espagnole :

```
#############################################################################
# Automatically generated by qmake (2.01a) ven. 23. mai 16:31:10 2008
#############################################################################

TEMPLATE = app
TARGET =
DEPENDPATH += .
INCLUDEPATH += .

# Input
HEADERS += MainWindow.h Scene.h
SOURCES += MainWindow.cpp Scene.cpp main.cpp
TRANSLATIONS = dictionary_en.ts dictionary_es.ts
```

Ouvrez une console (sous Windows, utilisez le raccourci Qt Command Prompt). Allez dans le dossier de votre projet.
Tapez :

```
lupdate Projet3.pro
```

lupdate est un programme qui va mettre à jour les fichiers de traduction .ts, ou les créer si ceux-ci n'existent pas.

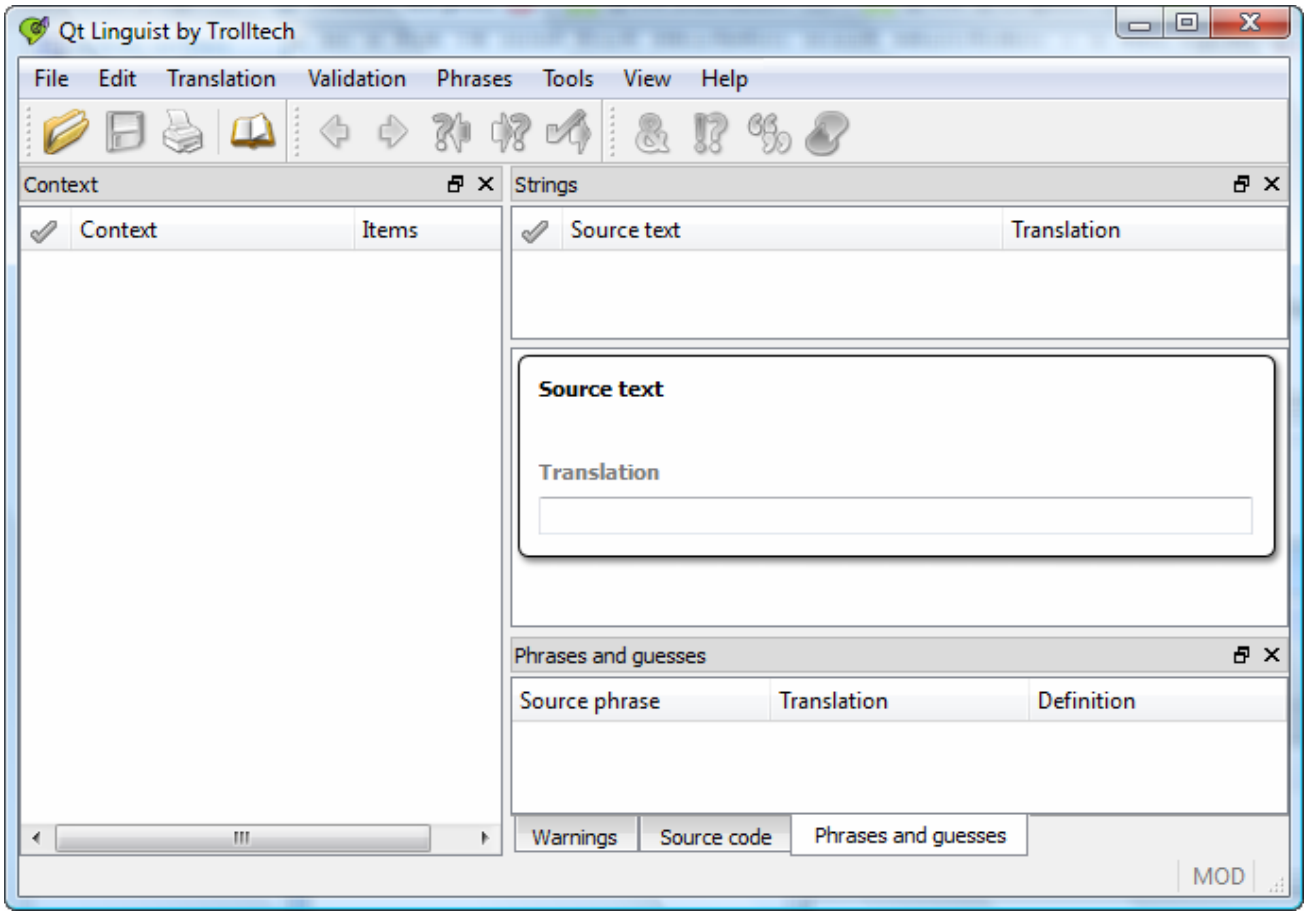
```
C:\Workspace\Projet3\>lupdate Projet3.pro
Updating 'dictionary_en.ts'...
    Found 17 source text(s) (15 new and 0 already existing)
Updating 'dictionary_es.ts'...
    Found 17 source text(s) (14 new and 0 already existing)
```

Il doit y avoir 2 fichiers supplémentaires dans le dossier de votre projet : dictionary_en.ts et dictionary_es.ts
Envoyez-les à votre traducteur

3- Traduire l'application sous Qt Linguist
Votre traducteur a besoin de 2 choses :

Du fichier .ts à traduire
Et de Qt Linguist pour pouvoir le traduire !

Votre traducteur lance donc Qt Linguist. Il devrait voir quelque chose comme ça :



Ouvrez un des fichiers .ts avec Qt Linguist, par exemple `dictionary_en.ts`. La fenêtre se remplit :

Détaillons un peu chaque section de la fenêtre :

- Context: affiche la liste des fichiers source qui contiennent des chaînes à traduire.
- Strings : c'est la liste des chaînes à traduire pour le fichier sélectionné.
- Au milieu : vous avez la version française de la chaîne, et on vous demande d'écrire la version anglaise.
- Warnings : affiche des avertissements bien utiles , comme "Vous avez oublié de mettre un & pour faire un raccourci",

C'est maintenant au traducteur de traduire tout ça !
Lorsqu'il est sûr de sa traduction, il doit marquer la chaîne comme étant validée (en cliquant sur le petit "?" ou en faisant Ctrl + Entrée). Un petit symbole coché vert doit apparaître, et le dock context doit afficher que toutes les chaînes ont bien été traduites (16/16 par exemple).

On procède donc en 2 temps : d'abord on traduit, puis ensuite on se relit et on valide. Lorsque toutes les chaînes sont validées (en vert), le traducteur vous rend le fichier .ts.

Il ne nous reste plus qu'une étape : compiler ce .ts en un .qm, et adapter notre programme pour qu'il charge automatiquement le programme dans la bonne langue.

4- Lancer l'application traduite
Nous avons le .ts entièrement traduit par notre traducteur adoré, il ne nous reste plus qu'à le compiler dans le format final binaire .qm, et à le charger dans l'application.

4.1- Compiler le .ts en .qm
Pour effectuer cette compilation, nous devons utiliser un autre programme de Qt : lrelease.

Ouvrez donc une console Qt (Qt Command Prompt), rendez-vous dans le dossier de votre projet, et tapez :

```
lrelease nomDuFichier.ts
```

... pour compiler le fichier .ts indiqué.
Vous pouvez aussi faire :

```
lrelease nomDuProjet.pro
```

... pour compiler tous les fichiers .ts du projet.

```
C:\Workspace\Projet3\>lrelease dictionary_en.ts
Updating 'dictionary_en.qm'...
    Generated 17 translation(s) (15 finished and 0
unfinished)
```

lrelease ne compile que les chaînes marquées comme terminées (celles qui ont le symbole vert dans Qt Linguist). Si certaines ne sont pas marquées comme terminées, elles ne seront pas compilées dans le .qm.
Nous avons maintenant un fichier dictionary_en.qm dans le dossier de notre projet.

4.2- Charger un fichier de langue .qm dans l'application

Le chargement d'un fichier de langue s'effectue au début de la fonction `main()`.
Pour le moment, votre fonction `main()` devrait ressembler à quelque chose comme ceci :

```
int main(int argc, char* argv[])
{
    QApplication app(argc, argv);

    FenPrincipale fenetre;
    fenetre.show();

    return app.exec();
}
```

Juste après la création de l'objet de type QApplication, nous allons rajouter les lignes suivantes :

```
int main(int argc, char* argv[])
{
    QApplication app(argc, argv);

    QTranslator translator;
    translator.load("dictionary_en");
    app.installTranslator(&translator);

    FenPrincipale fenetre;
    fenetre.show();

    return app.exec();
}
```

Vérifiez bien que le fichier .qm se trouve dans le même dossier que l'exécutable, sinon la traduction ne sera pas chargée et vous aurez toujours l'application en français

Pour que l'application s'adapte à la langue de l'utilisateur

```
int main(int argc, char* argv[])
{
    QApplication app(argc, argv);

    QString locale = QLocale::system().name().section('_', 0, 0);

    QTranslator translator;
    translator.load(QString("dictionary_") + locale);
    app.installTranslator(&translator);

    FenPrincipale fenetre;
    fenetre.show();

    return app.exec();
}
```

on veut récupérer le code à 2 lettres de la langue du PC de l'utilisateur. On utilise une méthode statique de QLocale pour récupérer des informations sur le système d'exploitation sur lequel le programme a été lancé.

La méthode `QLocale::system().name()` renvoie un code ressemblant à ceci : "fr_FR", où "fr" est la langue (français) et "FR" le pays (France).

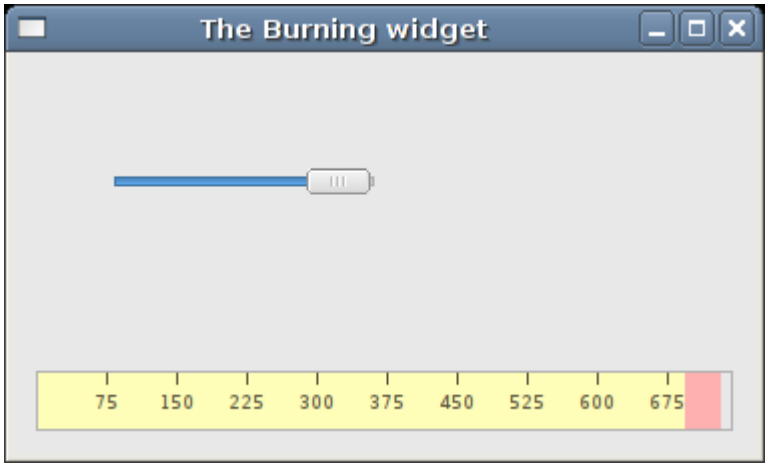
Optionnel Nous allons nous intéresser à la création d'un nouveau composant

1- Création du projet `Projet4`

Avez-vous jamais regardé une application en vous demandant, comment un élément particulier GUI a été créé? Ensuite, vous regardez la liste des widgets fournis par votre bibliothèque préférée GUI. Mais vous ne pouvez pas le trouver. Boîtes à outils ne fournissent habituellement que les widgets les plus courantes comme les boutons, les widgets de texte, etc curseurs Aucune boîte à outils ne peut fournir tous les widgets possibles.

Les programmeurs doivent créer des widgets par eux-mêmes. Ils le font en utilisant les outils de dessin fournis par le Toolkit. Il ya deux possibilités. Un programmeur peut modifier ou améliorer un widget existant. Ou il peut créer un widget personnalisé à partir de zéro.

Dans l'exercice suivant, nous allons créer un widget personnalisés. Ce widget peut être vu dans des applications comme Nero ou k3b. Le widget sera créé à partir de zéro : un slider et une bande graduée



Nous aurons 2 widgets sur la zone cliente de la fenêtre. Un curseur prédéfini et d'un widget personnalisé. La variable `cur_width` tiendra la valeur actuelle du curseur. Cette valeur est utilisée lors de la peinture du widget personnalisé. Définir la classe `Burning`

```
#ifndef BURNING_H
#define BURNING_H

#include <QWidget>
#include <QSlider>
#include "widget.h"


class Burning : public QFrame
{
  Q_OBJECT

  public:
    Burning(QWidget *parent = 0);

  public slots:
    void valueChanged(int);
    int getCurrentWidth();

  private:
    QSlider *slider;
    Widget *widget;
    int cur_width;

};
#endif
```

Burning.cpp

```
#include "burning.h"
#include <QApplication>
#include <QPainter>
#include <QVBoxLayout>
#include <QFrame>
#include <QPushButton>

Burning::Burning(QWidget *parent)
    : QFrame(parent)
{
  slider = new QSlider(Qt::Horizontal , this);
  slider->setMaximum(750);
  slider->setGeometry(50, 50, 130, 30);

  connect(slider, SIGNAL(valueChanged(int)), this, SLOT(valueChanged(int)));

  QVBoxLayout *vbox = new QVBoxLayout(this);
  QHBoxLayout *hbox = new QHBoxLayout();

  vbox->addStretch(1);

  widget = new Widget(this);
  hbox->addWidget(widget, 0);

  vbox->addLayout(hbox);

  setLayout(vbox);

}

void Burning::valueChanged(int val)
{
  cur_width = val;
  widget->repaint();
}

int Burning::getCurrentWidth()
{
  return cur_width;
}
```

2- Création d'une fenêtre principale

Lorsque nous changeons la valeur du slider, nous stockons la nouvelle valeur et repeindre le widget personnalisé.
Définition de la classe Widget qui est la classe principale
widget.h

```
#ifndef WIDGET_H
#define WIDGET_H
class Burning;
#include <QFrame>

class Widget : public QFrame
{
  Q_OBJECT

  public:
    Widget(QWidget *parent = 0);
```

```
  protected:
    void paintEvent(QPaintEvent *event);

  public:
    QWidget *m_parent;
    Burning *burn;
};
#endif
```

Nous stockons un pointeur sur le widget parent. Nous obtenons les `cur_width` par ce pointeur.
`widget.cpp`

```
#include "widget.h"
#include "burning.h"
#include <QPainter>
#include <QFrame>
#include <QHBoxLayout>
#include <QTextStream>

QString num[] = { "75", "150", "225", "300", "375", "450", "525", "600", "675" };
int asize = sizeof(num)/sizeof(num[1]);

Widget::Widget(QWidget *parent)
    : QFrame(parent)
{
  m_parent = parent;
  setFrameShape(QFrame::StyledPanel);
  setMinimumHeight(30);
}

void Widget::paintEvent(QPaintEvent *event)
{
  QPainter painter(this);
  painter.setPen(QColor("#d4d4d4"));

  int width = size().width();

  Burning *burn = (Burning *) m_parent;
  int cur_width = burn->getCurrentWidth();
  int step = (int) qRound(width / 10.0);
  int till = (int) ((width / 750.0) * cur_width);
  int full = (int) ((width / 750.0) * 700);

  if (cur_width >= 700) {
      painter.setPen(QColor(255, 255, 184));
      painter.setBrush(QColor(255, 255, 184));
      painter.drawRect(0, 0, full, 30);
      painter.setPen(QColor(255, 175, 175));
      painter.setBrush(QColor(255, 175, 175));
      painter.drawRect(full, 0, till-full, 30);
  } else {
      painter.setPen(QColor(255, 255, 184));
      painter.setBrush(QColor(255, 255, 184));
      painter.drawRect(0, 0, till, 30);
  }

  painter.setPen(QColor(90, 80, 60));
  for ( int i=1; i <= asize; i++ ) {
    painter.drawLine(i*step, 0, i*step, 6);
    QFont newFont = font();
    newFont.setPointSize(7);
    setFont(newFont);
```

```
    QFontMetrics metrics(font());

    int w = metrics.width(num[i-1]);
    painter.drawText(i*step-w/2, 19, num[i-1]);
  }
  QFrame::paintEvent(event);
}
```

Nous propageons l'événement de peinture supplémentaires au widget parent.

main.cpp

```
#include "burning.h"
#include <QDesktopWidget>
#include <QApplication>

void center(QWidget &widget)
{
  int x, y;
  int screenWidth;
  int screenHeight;

  int WIDTH = 370;
  int HEIGHT = 200;

  QDesktopWidget *desktop = QApplication::desktop();

  screenWidth = desktop->width();
  screenHeight = desktop->height();

  x = (screenWidth - WIDTH) / 2;
  y = (screenHeight - HEIGHT) / 2;

  widget.setGeometry(x, y, WIDTH, HEIGHT);
}

int main(int argc, char *argv[])
{
  QApplication app(argc, argv);
  Burning window;

  window.setWindowTitle("The Burning widget");
  window.show();
  center(window);

  return app.exec();
}
```

Optionnel Nous allons nous intéresser à la création de plugin

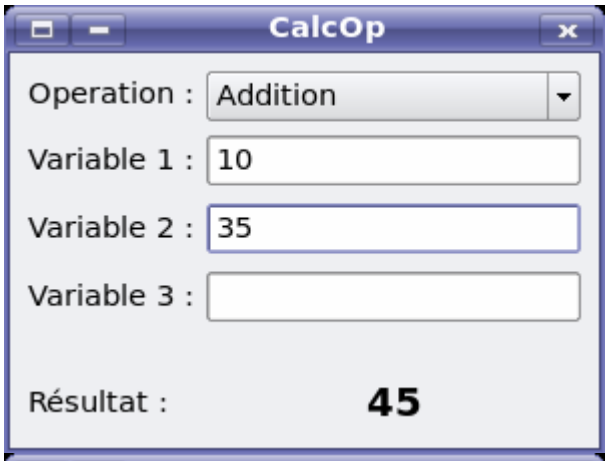
1- Création du projet `Projet5`

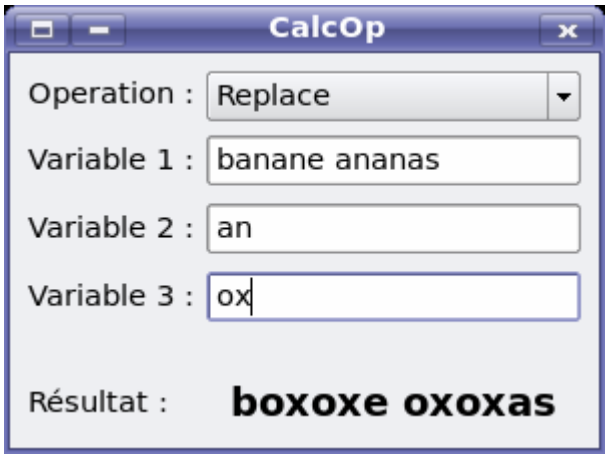
Les avantages d'un système de plug-ins sont nombreux. En voici quelques-uns :

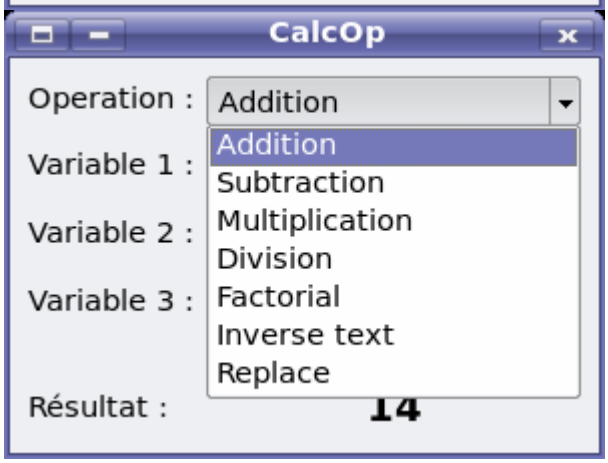
- l'utilisateur ou l'intervenant externe peut étendre les fonctionnalités d'une application sans avoir son code source;
- le logiciel est beaucoup plus modulaire : possibilité de faire des versions réduites, versions complètes… ;
- la mise en place d'un système de plug-in force à réfléchir un peu à l'architecture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

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

Pour illustrer la création de plug-in, nous allons utiliser un exemple assez simple : un petit projet nommé Projet5 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...).







- 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).

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`.

```
#ifndef OPERATIONINTERFACE_H
#define OPERATIONINTERFACE_H

class QString;
class QStringList;
```

```
/* name : renvoie la liste des operations gerees,
 * canCalculate : indique si operations est prise en charge
 * numVariable : renvoie le nombre de variables utilisees pour chaque operations
 * calculate : renvoie le resultat pour une operations
 */
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;
};

#endif // OPERATIONINTERFACE_H
Q_DECLARE_INTERFACE(OperationInterface,
"tp.projet5.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 " `tp.projet5.CalcOp.OperationInterface` " doit obligatoirement se finir par le nom de la classe.

2- 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.

3- 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 faciliter la vie, c'est 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 :

`OperationManager.h`

```
#ifndef OPERATIONMANAGER_H
#define OPERATIONMANAGER_H

#include <QList>

class OperationInterface;
class QString;
class QStringList;

/*!
 * \brief Gestionnaire d'operations
 * Il est utilis pour le chargement des differents plugins. Il garde les
 * plugins en memoire apres le chargement.
 * Cette classe est un singleton.
 */
class OperationManager
{
public:
    ~OperationManager();

    QStringList operationList();
    int numVariable( QString opName );
    QString calculate( QString opName, QStringList variableList );

    static OperationManager * instance();

private:
    OperationManager();
    OperationInterface * operation( QString opName );
```

```
private:
    QList<OperationInterface *> m_operationList;
    static OperationManager * m_instance;
};

#endif // OPERATIONMANAGER_H
```

OperationManager.cpp

```
#include <QtGui>

#include "operationmanager.h"
#include "operationinterface.h"

OperationManager * OperationManager::m_instance = 0;


OperationManager::OperationManager()
{
    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;
            }
        }
    }
}

OperationManager::~OperationManager()
{
    /* Pas besoin de supprimer les plugins en mÃ©moire,
    ils sont supprimÃ©s automatiquement Ã  la fin. */
}

QStringList OperationManager::operationList()
{
    QStringList list;
    foreach(OperationInterface * op, m_operationList) {
        list << op->operationList();
    }
    return list;
}

int OperationManager::numVariable( QString opName )
{
    OperationInterface * op = operation( opName );
    if (op) {
        return op->numVariable( opName );
    }
    return 0;
}

QString OperationManager::calculate( QString opName, QStringList variableList )
{
    OperationInterface * op = operation( opName );
    if (op) {
        return op->calculate( opName, variableList );
```

```
    }
    return QString();
}

OperationManager * OperationManager::instance()
{
    if (m_instance==0) {
        m_instance = new OperationManager();
    }
    return m_instance;
}

OperationInterface * OperationManager::operation( QString opName )
{
    foreach(OperationInterface * op, m_operationList) {
        if (op->canCalculate( opName )) {
            return op;
        }
    }
    return 0;
}
```

Nous avons 3 actions importantes dans ce code :

- déplacement dans le répertoire de plug-in : les 2 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.

Il faut penser à une interface fenêtre

`window.h`

```
#ifndef WINDOW_H
#define WINDOW_H

#include <QWidget>
#include "ui_window.h"


/*!
 * \brief FenÃªtre principale de l'application
 */
class Window : public QWidget, private Ui::Window
{
    Q_OBJECT

public:
    Window( QWidget * parent = 0, Qt::WFlags f = 0 );

private slots:
    void slotChangeOperation( const QString & text );
    void slotCalculate();
};

#endif
```

`window.cpp`

```
#include <QtGui>
```

```
#include "window.h"
#include "operationmanager.h"

Window::Window( QWidget * parent, Qt::WFlags f ) :
    QWidget(parent, f)
{
    setupUi(this);
    connect( m_operation, SIGNAL( activated( const QString & ) ),
            this, SLOT( slotChangeOperation( const QString & ) ) );
    connect( m_var1, SIGNAL( textChanged ( const QString & ) ),
             this, SLOT( slotCalculate() ) );
    connect( m_var2, SIGNAL( textChanged ( const QString & ) ),
             this, SLOT( slotCalculate() ) );
    connect( m_var3, SIGNAL( textChanged ( const QString & ) ),
             this, SLOT( slotCalculate() ) );

    // Initialisation
    m_operation->addItems( OperationManager::instance()->operationList() );
    m_operation->setCurrentIndex(0);
}


void Window::slotChangeOperation( const QString & text )
{
    int num = OperationManager::instance()->numVariable(text);
    m_var1->setEnabled( num >= 1 );
    m_var2->setEnabled( num >= 2 );
    m_var3->setEnabled( num >= 3 );
    slotCalculate();
}


void Window::slotCalculate()
{
    QStringList varList;
    varList << m_var1->text();
    varList << m_var2->text();
    varList << m_var3->text();
    QString result = OperationManager::instance()->calculate(
            m_operation->currentText(), varList );
    m_result->setText( result );
}
```

La classe `QPluginLoader` est au cœur 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).

1- Internationalisation

Qt Linguist est un outil de la bibliothèque Qt. Cet outil permet de traduire un à un les textes d'un programme qui doivent être affichés. Sa prise en main est relativement simple.

AJOUT D'UNE LANGUE

Le but est donc l'ajout d'une langue sur un programme finalisé.

1. DOCUMENT ET LOGICIELS DE REFERENCE

Les étapes de la traduction
Préparer son code à une traduction
Créer les fichiers de traduction .ts
Traduire l'application sous Qt Linguist
Lancer l'application traduite

2. AJOUT DE LANGUE OU MODIFICATION DU TEXTE D'AFFICHAGE

Pour modifier ou ajouter une langue, il y a trois procédures à respecter :
- Programmer l'application afin de créer des fichiers .ts,
- Traduire l'application sous Qt Linguist,
- Compiler l'application pour créer un fichier .qm.

2.1. Programmation

Le programme est construit de la façon suivante :
Dans un premier temps, un fichier « nom.ui » est créé. Dans ce fichier, il est possible de placer toutes sortes d'objets (boutons, tableaux…).
Le fichier « ui_nom.h » est ensuite généré lors de la reconstruction de l'application. C'est dans ce fichier « ui_nom.h » que l'on va intervenir pour la traduction.

2.1.1. Fonction retranslate()
On utilise la fonction : void retranslateUi(TypeClass *Nom_de_la_classe) qui regroupe les textes d'affichage à traduire. Cette fonction est activée au démarrage : elle permet de traduire tous les caractères enveloppés de la fonction translate() décrite ci-après.
2.1.2. Fonction translate()
La fonction translate() doit enveloppée tous les caractères à traduire.
Prenons un exemple pour la traduction : Dans la classe MainWindow, on retrouve des objets.

Par exemple, pushButton_E1.
Lorsque l'on souhaite ajouter un texte associé à l'objet (texte qui s'affiche à l'écran), il suffit d'ajouter un « titre » au bouton lors de la création du bouton.
La ligne de code suivante est ensuite générée :
pushButton_E1->setText("MainWindow", "E1", 0, Qapplication::UnicodeUTF8);
Le « titre » du bouton est ici « E1 ».

Pour permettre de traduire les textes affichés, il faut ajouter la fonction « QApplication :: translate( ) », tel que :
pushButton_E1->setText(QApplication::translate("MainWindow", "E1", 0, Qapplication::UnicodeUTF8));
Il sera ainsi possible de modifier le nom « E1 » diminutif pour évènement 1 par « A1 » pour acontecimiento 1 en espagnol par exemple, lors d'une éventuelle demande de traduction.

Il faut appliquer cette méthode pour tous les textes affichés que l'on souhaite traduire.

2.1.3. Ajout d'un bouton pour le choix de la langue
Pour ajouter une nouvelle langue, il faut également prévoir un nouveau bouton contenant le nom de la langue à traduire. Ce nom est un symbole à 2 lettres de la langue de destination tel que fr (français), en (anglais)…

Il faut ensuite connecter ce bouton à une action qui prend en compte le choix de l'utilisateur et informe l'utilisateur qu'il faut redémarrer l'application pour avoir accès à une autre langue :

Exemple :
connect(ui.ButtonFr, SIGNAL(clicked()), this, SLOT(LangConfFR()));

// Quand on clique sur le bouton FR, on lance la fonction LangConfFR

Remplacer “FR” dans la fonction précédente par le symbole à 2 lettres de la langue choisie. Il faut ensuite créer une fonction LangConfXX() telle que (en modifiant le symbole à 2 lettres) :

```
void mondialog::LangConfFR()
{
        QFile file("./translations/language");
        if ( !file.open(QIODevice::WriteOnly | QIODevice::Text)  )
         {
                file.close();
         }
        QTextStream out(&file);
        out << "FR";
        QMessageBox::information(this, tr("Lang Configuration"), tr("Need to Restart the program."));
}
```

Cette fonction est écrite dans le fichier mainwindown.cpp
Cette fonction appelle un fichier contenu dans le répertoire translations, ce fichier est appelé language. Il faut également ajouter les symboles utilisés dans ce fichier (non vérifié).

2.1.4. Création de fichiers de traduction .ts

Après avoir reconstruit le projet, il faut créer des fichiers .ts qui permettent de traduire les caractères choisis. Pour cela, ouvrir le fichier « nom_du_projet.pro » avec un éditeur de texte et ajouter la ligne suivante :
TRANSLATIONS = nom_du_projet_en.ts
Si la langue à ajouter est l’anglais, sinon modifier le symbole.

Autrement dit si il y a 2 langues disponibles, le fichier « nom_du_projet.pro » ressemblera à
TRANSLATIONS = nom_du_projet_en.ts
nom_du_projet_fr.ts

Il faut ensuite ouvrir un programme dans la console afin de générer automatiquement les fichiers .ts Il faut se placer dans le dossier du projet et taper : lupdate nom_du_projet.pro

Au démarrage, le fichier main.cpp teste quelques variables pour savoir dans quelle langue il doit afficher l’application.

2.2. Traduction

Une fois le programme précédent lancé, un fichier .ts est créé. Il faut alors l’ouvrir avec Qt Linguist.

Toutes les chaines de caractère à modifier sont présentes dans la colonne de gauche : il s’agit de toutes les chaines de caractères au préalablement enveloppées de « QApplication :: translate( ) »
Traduire ces chaines ligne par ligne en validant la traduction effectuée avec l’icône "flèche".
Une fois la traduction de l’application terminée, enregistrer le fichier .ts .
Attention ! Il ne faut pas que des « points d’interrogation » apparaissent. Il doit y avoir uniquement des « tics ».

2.3. Compilation

Il faut compiler l’application avec un programme de Qt : lrelease. A l’issue de cette application, des fichiers .qm sont créés et seront utilisés par le programme pour modifier la langue de l’application.

Pour cela, il faut ouvrir un programme dans la console. Il faut se placer dans le répertoire des fichiers .ts et taper : lrelease nom_du_projet_en.ts Compiler tous les fichiers .ts traduits.

Il faut enfin ajouter quelques lignes de code afin de charger les fichiers de langue .qm dans l’application.
Le chargement d’un fichier .qm s’effectue au début du main(). Il faut ajouter ces lignes juste après la création de l’objet de type QApplication :

```
QTranslator translator ;
translator.load(« zeroclassgenerator_en ») ;
```

```
app.installTranslator(&translator) ;
```