Université de la Rochelle
Licence d'Informatique - IHM (INFO132601)

TD 5 - *Intégration de widgets crées avec Qt Designer*

© B. Besserer, R. Péteri

Année universitaire 2016-2017

Qt Designer

"Qt Designer is Qt's tool for designing and building graphical user interfaces (GUIs) from Qt components. You can compose and customize your widgets or dialogs in a what-you-see-is-what-you-get (WYSIWYG) manner, and test them using different styles and resolutions.

Widgets and forms created with Qt Designer integrated seamlessly with programmed code, using Qt's signals and slots mechanism, that lets you easily assign behavior to graphical elements. All properties set in Qt Designer can be changed dynamically within the code. Furthermore, features like widget promotion and custom plugins allow you to use your own components with Qt Designer."

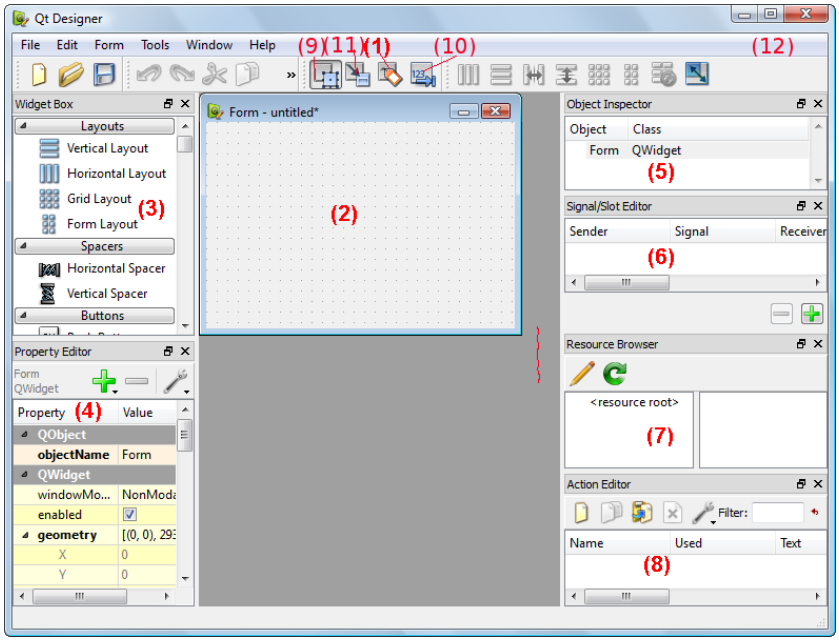

FIGURE 1 – L'interface de QT Designer

1. Associer les descriptions suivantes avec les numéros indiqués sur la figure 1
   a. Mode d'édition qui permet d'insérer des widgets sur la fenêtre et de modifier leurs propriétés.
   b. Editeurs de propriétés
   c. Permet de naviguer à travers les fichiers de ressources de votre application
   d. Affiche la liste des widgets placés sur la fenêtre, en fonction de leur relation de parenté, sous forme d'arbre
   e. Mode d'édition qui permet de créer des connexions entre les signaux et les slots de vos widgets.
   f. Zone de dessin

g. Mode d'édition qui permet de modifier l'ordre de tabulation entre les champs de la fenêtre, pour passer d'un champ à l'autre en appuyant sur la touche "Tab".

h. Barre d'outils

i. Permet de créer des QAction

j. Les connexions du widget sélectionné apparaissent ici

k. Choix des Widgets

l. Mode d'édition qui permet d'associer des QLabel avec leurs champs respectifs.

(12) :. Barre outils layouts
Ils permettent de passer d'un mode d'édition à un autre. Qt Designer propose 4 modes d'édition :

*(9) : * Edit Widgets : le mode par défaut, que vous utiliserez le plus souvent. Il permet d'insérer des widgets sur la fenêtre et de modifier leurs propriétés.*
*(11) * Edit Signals/Slots : permet de créer des connexions entre les signaux et les slots de vos widgets.*
*(1) * Edit Buddies : permet d'associer des QLabel avec leurs champs respectifs.*
*(10) * Edit Tab Order : permet de modifier l'ordre de tabulation entre les champs de la fenêtre, pour ceux qui naviguent au clavier et passent d'un champ à l'autre en appuyant sur la touche "Tab".*

2. Au centre de Qt Designer, vous avez la fenêtre que vous êtes en train de dessiner. Pour le moment celle-ci est vide. Si vous créez une QMainWindow : barre de menus et une barre d'outils.
Si vous créez une QDialog : boutons "OK" et "Annuler" déjà disposés.
3. Widget Box : ce dock vous donne la possibilité de sélectionner un widget à placer sur la fenêtre. assez large choix mais, ceux-ci sont organisés par groupes pour y voir plus clair. Pour placer un de ces widgets sur la fenêtre, il suffit de faire un glisser-déplacer.
4. Property Editor : lorsqu'un widget est sélectionné sur la fenêtre principale, vous pouvez éditer ses propriétés. Les widgets possèdent en général beaucoup de propriétés, celles-ci sont organisées en fonction de la classe dans laquelle elles ont été définies. On peut ainsi modifier toutes les propriétés dont un widget hérite, en plus des propriétés qui lui sont propres.

Comme toutes les classes héritent de QObject, il y a toujours la propriété objectName. C'est le nom de l'objet qui sera créé. Oa interet à le personnaliser, afin d'y voir plus clair tout à l'heure dans le code source (sinon vous aurez par exemple des boutons appelés pushButton, pushButton_2, pushButton_3, ce qui n'est pas très clair).s
Si aucun widget n'est sélectionné, ce sont les propriétés de la fenêtre que vous éditerez. Vous pourrez donc par exemple modifier son titre avec la propriété windowTitle, son icône avec windowIcon, etc.

5. Object Inspector : affiche la liste des widgets placés sur la fenêtre, en fonction de leur relation de parenté, sous forme d'arbre. Ca peut être pratique si vous avez une fenêtre complexe et que vous commencez à vous perdre dedans. Vous pouvez ainsi y voir par exemple que votre fenêtre contient un QGroupBox qui contient 3 cases à cocher.

6. Signal / slot editor : si vous avez associé des signaux et des slots, les connexions du widget sélectionné apparaissent ici.

7. Resource Browser : un petit utilitaire qui vous permet de naviguer à travers les fichiers de ressources de votre application. Ces fichiers de ressources rappellent un peu ceux de Windows . Ici, les fichiers de ressouces portent l'extension .qrc et ont l'avantage d'être compatibles avec tous les OS.
Les fichiers de ressources servent empaqueter des fichiers (images, sons, texte...) au sein même de votre exécutable. Cela permet d'éviter d'avoir à placer ces fichiers dans le même dossier que votre programme, et cela évite donc le risque de les perdre (puisqu'ils se trouveront toujours dans votre exécutable).

8. Action Editor : permet de créer des QAction. C'est donc utile lorsque vous créez une QMainWindow avec des menus et une barre d'outils.

2. A quoi sert le mode d'édition Buddies ? *permet d'associer des QLabel avec leurs champs respectifs, par ex. une spinbox, et d'utiliser le raccourci clavier du QLabel pour activer la spinbox 'buddy'.*

3. Quel est l'avantage d'utiliser Qt Designer au niveau des connections dans votre widget ? *Qt Designer ne propose que les connections SIGNAL/SLOT existantes pour les widgets que vous souhaitez connecter, ce qui permet de trouver les bons signal/slot et d'éviter les erreurs de synthaxe et de mauvais arguments.*
4. Quelles sont les différentes étapes lorsque l'on crée son widget avec QtDesigner ? Ou se trouvent ces modes d'édition sur la figure ? *Edit Widgets, puis Edit Signals/Slots, puis Edit Buddies puis Edit Tab Order.*
5. Quel est le type de fichier produit par QtDesigner ? Comment est-il intégré dans un programme ?
   Qt Designer ne fait que produire un fichier .ui. C'est le petit programme uic qui se charge de transformer le .ui en code source C++. Vous dessinez la fenêtre avec Qt Designer qui produit un fichier .ui. Ce fichier est transformé automatiquement en code source par le petit programme en ligne de commande uic. Celui-ci génèrera un fichier ui_nomDeVotreFenetre.h. Qt met tout le code dans le fichier .h
   Voilà ce que ça donne schématiquement :

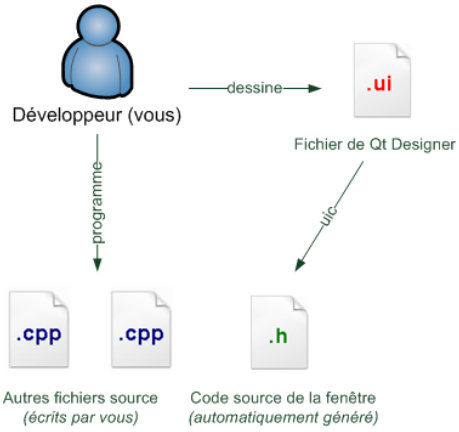

FIGURE 2 – Schéma compilation des UI

6. On souhaite maintenant intégrer un widget cré par Qt Designer qui s'appelle mon_widget.ui dans une application : que faut-il rajouter dans le fichier .pro suivant ?

```
#############################################################################
# Automatically generated by qmake
#############################################################################

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

# Input
SOURCES += main.cpp
```

```
#############################################################################
# Automatically generated by qmake
#############################################################################

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

# Input
FORMS +=  mon_Widget.ui
SOURCES += main.cpp
```

   Ajout de la ligne FORMS. Elle donne la liste des fichiers .ui utilisés par votre application.
   Le fichier ui_mon_Widget.h sera généré par uic au moment de la compilation, lorsque vous taperez make .Le programme uic sera automatiquement appelé juste avant la compilation.
7. **Méthode directe**
   (a) C'est la manière la plus simple pour intégrer dans une application. Voici le fichier main.cpp : rajouter les lignes pour intégrer votre widget.

```
#include <QApplication>
#include <QtGui>

int main(int argc, char *argv[])
{
    QApplication app(argc, argv);
    QWidget *fenetre = new QWidget;


    fenetre->show();

    return app.exec();
}
```

```
#include <QApplication>
#include <QtGui>
#include "ui_mon_Widget.h"

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

    QWidget *fenetre = new QWidget;
    Ui::FenWidget ui; //namespace Ui, voir cours sous Moodle.
    ui.setupUi(fenetre);

    fenetre->show();

    return app.exec();
}
```

on crée une nouvelle fenêtre en créant un nouvel objet de type QWidget.
Au lieu d'afficher cette fenêtre directement, on la précharge avec le contenu que l'on a dessiné dans Qt Designer. Pour cela, on crée un objet de type Ui : :FenWidget(où "FenWidget" est le nom que vous avez donné à votre fenêtre dans Qt Designer). On appelle setupUi(fenetre) pour dessiner le contenu de la fenêtre avec l'interface réalisée sous Qt Designer.
On peut ensuite ouvrir la fenêtre avec fenetre->show() ;
Ensuite : on compile avec make en ligne de commande. Le programme uic sera appelé en arrière-plan pour que le fichier ui_ma_Widget.h soit généré. Puis, l'ensemble des fichiers .cpp et .h seront compilés classiquement.

(b) Quels sont les avantages et les inconvénients de cette méthode ?

Avantages : technique très simple à mettre en oeuvre, à peine quelques lignes à écrire.
Défauts : pas de possibilité de personnaliser la fenêtre, ni d'écrire des slots personnalisés. La fenêtre est "figée".

8. **Méthode avec héritage**

(a) Nous allons hériter de la fenêtre créée avec Qt Designer. Pour faire cela, il faut créer une nouvelle classe dans notre projet intitulée "FenWidget" (du même nom que la fenêtre créée sous Qt Designer) et donc créer un .cpp et un .h. Au final, le projet doit comporter les fichiers suivants :
— main.cpp
— FenWidget.h
— FenWidget.cpp
Ensuite il faut refaire un qmake -project pour mettre à jour le fichier .pro du projet.
Commenter le fichier $FenWidget.h$:

```
#ifndef FENWIDGET_H
#define FENWIDGET_H

#include <QtGui>
#include "ui_mon_Widget.h"

class FenWidget: public QWidget
{
    Q_OBJECT

    public:
        FenWidget(QWidget *parent = 0);

    private slots:
        /* Insérez les prototypes de vos slots personnalisés ici */

    private:
        Ui::FenWidget ui;
```

```
};

#endif // FENWIDGET_H
```

On inclut "ui_ma_Widget" pour pouvoir utiliser la fenêtre créée avec Qt Designer.
On crée une classe FenWidget héritant de QWidget.
On crée un constructeur classique.
On déclare un objet "ui" de type Ui : : FenWidget.

(b) Ecrivez le fichier $FenWidget.cpp$

```
#include "FenWidget.h"

#include "FenWidget.h"

FenWidget::FenWidget(QWidget *parent) : QWidget(parent)
{
    ui.setupUi(this); // A faire en premier

    /*
    Personnalisez vos widgets ici si nécessaire
    Réalisez des connexions supplémentaires entre signaux et slots
    */
}
```

Il faut juste mettre un ui.setupUi(this) pour créer le contenu de la fenêtre. Il faut faire cela en premier dans le constructeur. Ensuite, on peut personnaliser les widgets et créer des connexions supplémentaires entre des signaux et des slots.

(c) Tous les widgets sont accessibles en faisant ui.nomDuWidget. Si on suppose que notre widget créé sous Qt Designer possède un bouton $MonBouton$, quelle ligne de code faut-il rajouter pour changer le label du bouton en “Cliquez ici“ ?

```
#include "FenWidget.h"

FenWidget::FenWidget(QWidget *parent) : QWidget(parent)
{
    ui.setupUi(this); // A faire en premier
    ui.MonBouton->setText("Cliquez ici !");

}
```

(d) Ecrire la ligne pour effectuer la connection entre un clic sur MonBouton et le slot close()

```
#include "FenWidget.h"

FenWidget::FenWidget(QWidget *parent) : QWidget(parent)
{
    ui.setupUi(this); // A faire en premier
    ui.MonBouton->setText("Cliquez ici !");
    connect(ui.MonBouton, SIGNAL(clicked()), this, SLOT(close()));

}
```

(e) Ecrire enfin le fichier $main.cpp$

```
#include <QApplication>
#include <QtGui>
#include "FenWidget.h"

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

    FenWidget fenetre;
    fenetre.show();

    return app.exec();
}
```

(f) Quels sont les avantages et les inconvénients de cette méthode ? *La méthode permet de définir ses slots personnalisés et de personnaliser la fenetre, mais il ne faut pas oublier écrire le préfixe "ui" devant le nom du widget à chaque fois (namespace).*

9. **Ajout d'un compteur lié à un timer**
   On souhaite rajouter dans notre widget `mon_widget.ui` un affichage LCD (variable `mon_lcd`) qui affichera les valeurs d'un timer (variable `mon_timer`) déclaré dans la classe `FenWidget`.

   Ecrivez le code permettant de réaliser cette fonctionnalité.
   Il faut déclarer un SIGNAL incremente_value(int increment) dans la classe principale qui incremente une variable increment dont la valeur sera ensuite passée au LCD de la widget. Le timer sera relié à un slot `void incremente_lcd()` de la classe (pas de signal dans la classe QTimer pour envoyer la valeur du timer).

```
 .H:
 #ifndef FENWIDGET_H
#define FENWIDGET_H

#include <QtGui>
#include "ui_ma_Widget.h"

class FenWidget: public QWidget
{
    Q_OBJECT

    public:
        FenWidget(QWidget *parent = 0);
int increment;
    private slots:
        /* Insérez les prototypes de vos slots personnalisés ici */
void incremente_lcd();

signals:
   void incremente_value(int increment);

    private:
        Ui::FenWidget ui;
};


#endif // FENWIDGET_H



.CPP:
#include "FenWidget.h"
#include<QTimer>

FenWidget::FenWidget(QWidget *parent) : QWidget(parent)
{
    ui.setupUi(this); // A faire en premier
    ui.MonBouton->setText("Cliquez ici !");
    connect(ui.MonBouton, SIGNAL(clicked()), this, SLOT(close()));
QTimer *mon_timer= new QTimer();
mon_timer->start(1000);
this->increment=0;

connect(mon_timer,SIGNAL(timeout()),this,SLOT(incremente_lcd()));
connect(this,SIGNAL(incremente_value(int)),ui.mon_lcd,SLOT(display(int)));
    /*
    Personnalisez vos widgets ici si nécessaire
    Réalisez des connexions supplémentaires entre signaux et slots
    */
}

void FenWidget::incremente_lcd()
{
    this->increment++;
emit incremente_value(this->increment);

}
```