Ce quatrième projet va nous permettre d'aller un peu plus dans le détail d'une programmation orientée objet. Par la même occasion, nous en profiterons pour élaborer des projets en relation avec des structures.

***Nota** : Cette fois-ci, nous nous intéressons plus particulièrement au code source. Nous allons tout simplement reprendre l'étude précédente en allégeant considérablement l'écriture.*

ANALYSE DE L'HÉRITAGE ET DE L'AGRÉGATION

En consultant les sources, nous remarquons que :

- *La classe **Conversion** **hérite** de la classe **QMainWindow** : elle récupère ainsi automatiquement de toutes les compétences de cette classe préfabriquée (attributs et méthodes). De cette manière, sans écritures supplémentaires, notre nouvelle classe est tout à fait capable de se comporter comme une fenêtre principale d'application.*
- *La classe **Conversion** possède également un attribut (un objet) nommé **ui** (**U**nit **I**nterface) qui représente l'**IHM** de l'application. Cette interface est elle-même composée d'autres objets, ceux que nous avons justement choisis pour constituer notre projet, comme la zone d'édition **euro**, la zone **franc** ainsi que le bouton de **conversion**. Finalement, un objet peut être composé d'autres objets, comme une voiture possède un volant, des roues, etc. Dans la terminologie objet, nous appelons cela une **agrégation** (ou **composition** dans le cas d'une **agrégation** forte).*

conversion.h <Select Symbol>

```
1   #ifndef CONVERSION_H
2   #define CONVERSION_H
3
4   #include <QtGui/QMainWindow>
5
6   namespace Ui
7   {
8       class Conversion;
9   }
10
11  class Conversion : public QMainWindow
12  {
13      Q_OBJECT
14  public:
15      Conversion(QWidget *parent = 0);
16      ~Conversion();
17  private:
18      Ui::Conversion *ui;
19  private slots:
20      void convertir();
21  };
22
23  #endif // CONVERSION_H
```

Création d'un espace de nom (optionnel)

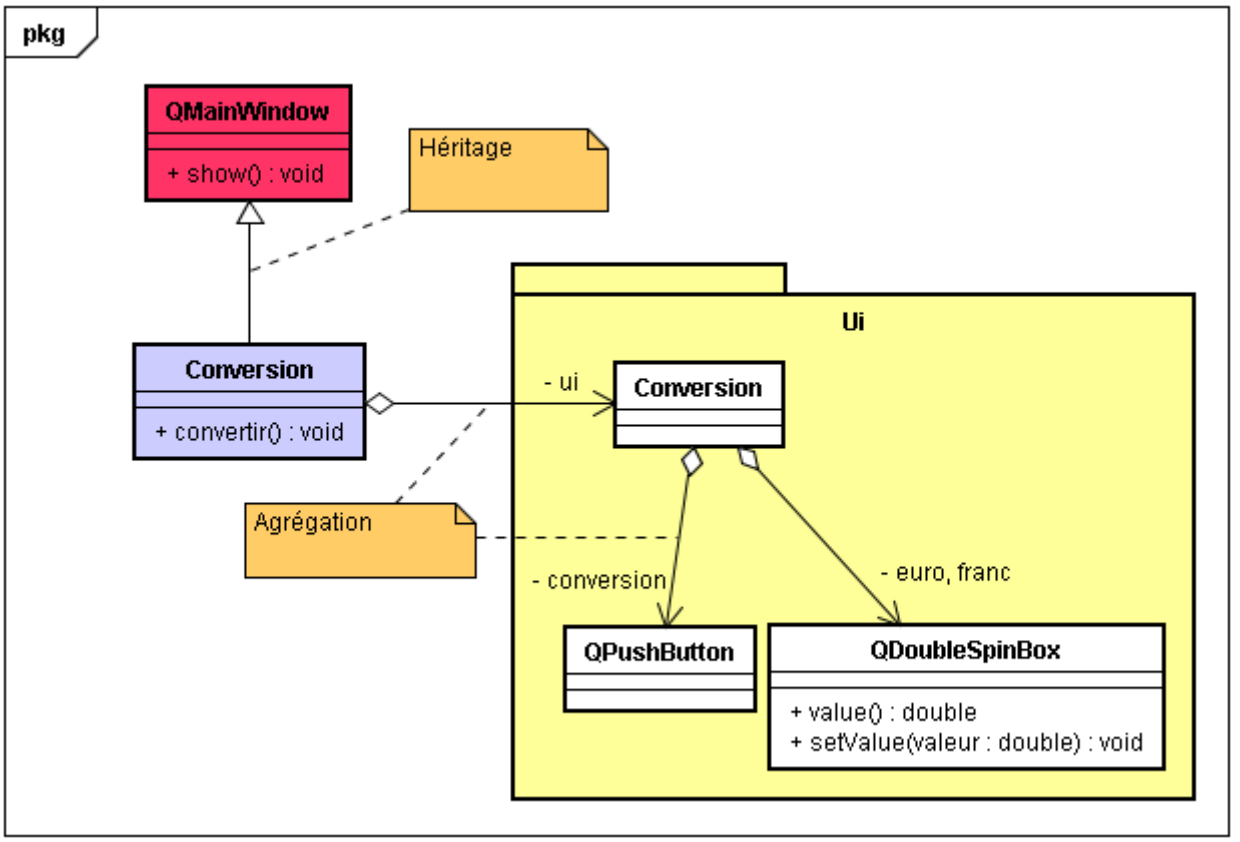
Classe personnalisée qui correspond à l'application désirée

Héritage

Constructeur et destructeur

Agrégation : notre application est composée d'une IHM.

Nouvelle méthode de traitement à prendre en compte.



conversion.cpp <Select Symbol>

```
1   #include "conversion.h"
2   #include "ui_conversion.h"
3
4   Conversion::Conversion(QWidget *parent)
5       : QMainWindow(parent), ui(new Ui::Conversion)
6   {
7       ui->setupUi(this);
8   }
9
10  Conversion::~Conversion()
11  {
12      delete ui;
13  }
14
15  void Conversion::convertir()
16  {
17    const double TAUX = 6.55957;
18    double e = ui->euro->value();
19    double f = e * TAUX;
20    ui->franc->setValue(f);
21  }
22
```

Objet représentant la partie visible de la fenêtre principale de l'application.

*L'intérêt de l'**agrégation**, c'est que chaque objet possède sa propre identité. Le problème toutefois, c'est qu'à chaque fois que vous désirez faire quelque chose sur cet objet, vous êtes obligés d'y faire systématiquement référence. Ainsi, pour récupérer la valeur saisie en €uro, vous devez d'abord faire référence à **ui**, ce qui est tout à fait logique puisque la zone de saisie **euro** est bien dans l'**IHM**. Nous devons donc écrire explicitement **ui->euro (pointeur)**. De façon analogue, pour changer la valeur des Francs, vous devez explicitement écrire **ui->franc**.*

*Si vous avez beaucoup de manipulation à faire sur les composants qui sont sur l'**IHM** (normalement c'est toujours le cas), cela peut s'avérer très vite fastidieux et barbant. Il serait plus judicieux de procéder différemment. L'idée, c'est de faire en sorte que notre classe personnalisée soit à la fois une fenêtre principale d'application, mais qu'elle soit aussi la partie visible, c'est-à-dire l'**IHM** de cette classe principale. Nous connaissons déjà la technique. Il suffit de faire un nouvel héritage. Il est donc nécessaire que la classe **Conversion** **hérite** également de **Ui::Conversion**.*

*Il faut rappeler que la classe **Ui::Conversion** est automatiquement générée. Elle est constituée de tout les composants, avec les différents réglages, que vous avez décidé de placer dans le mode « **Design** ». Grâce à l'héritage, vous récupérez ainsi automatiquement tous les éléments que vous avez choisi à l'aide de la souris, et dans votre code, vous pouvez y faire directement référence. Pour en revenir à notre exemple, pour récupérer la valeur en €uro, vous désignez directement l'attribut **euro** (Même remarque pour l'attribut **franc)**.*

SIMPLIFICATION DU CODE EN TENANT COMPTE DE NOS REMARQUES

Nous allons simplifier les sources précédents en soumettant un héritage sur l'objet représentant la partie visible de la fenêtre principale de l'application plutôt que de prendre l'agrégation proposée par défaut.

Nous en profitons pour supprimer l'espace de nom. Par ailleurs, vu que le pointeur vers l'objet **ui** n'existe plus, la phase de destruction n'est plus nécessaire.

***Attention** : il faut penser à inclure le fichier en-tête **« ui_conversion.h »** dans le fichier **« conversion.h »** plutôt que dans le fichier **« conversion.cpp »**.*

conversion.h — Conversion

```
1   #ifndef CONVERSION_H
2   #define CONVERSION_H
3
4   #include <QDialog>
5   #include "ui_conversion.h"
6
7   class Conversion : public QDialog, public Ui::Conversion
8   {
9     Q_OBJECT
10  public:
11    Conversion(QWidget *parent = 0);
12  private slots:
13    void convertir();
14  };
15
16  #endif // CONVERSION_H
17
```

Penser à faire cette inclusion

Héritage supplémentaire

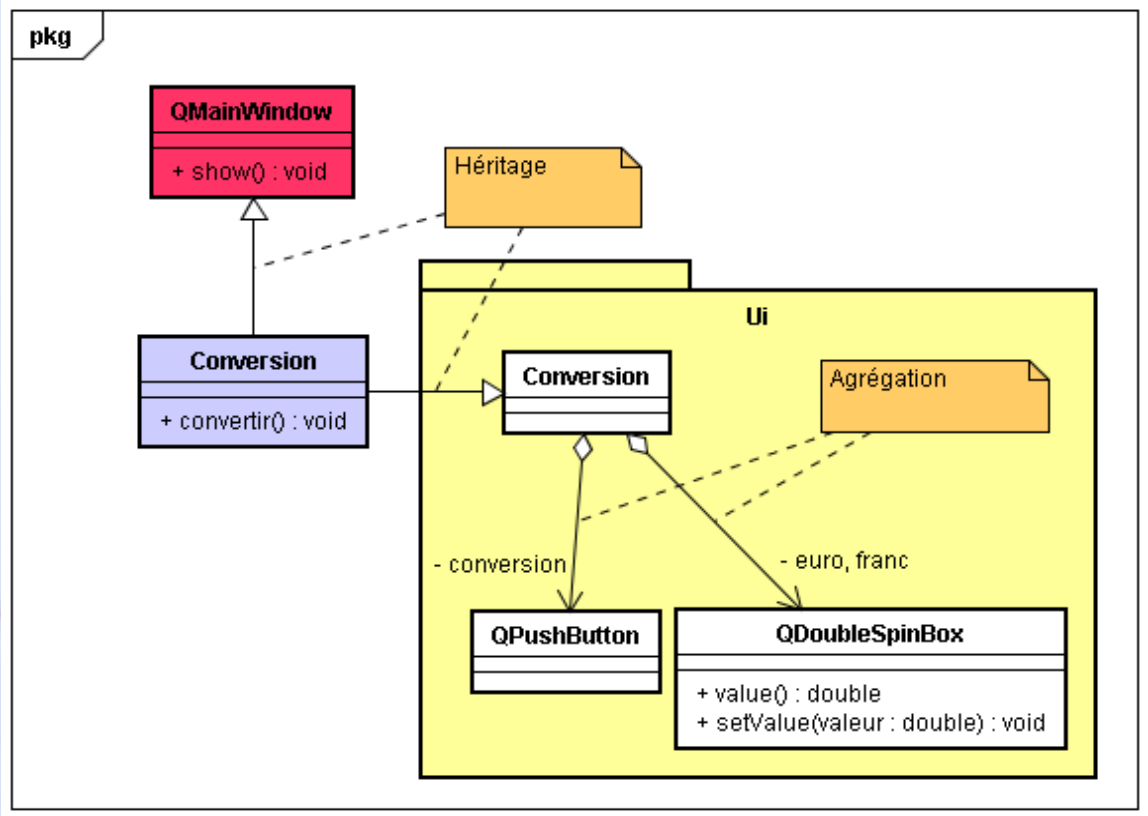
Pas besoin de faire un espace de nom spécifique.

Le destructeur et ui n'existent plus.

conversion.cpp — <Select Symbol>

```
1   #include "conversion.h"
2
3   Conversion::Conversion(QWidget *parent) : QDialog(parent)
4   {
5     setupUi(this);
6   }
7
8   void Conversion::convertir()
9   {
10    const double TAUX = 6.55957;
11    double e = euro->value();
12    double f = e * TAUX;
13    franc->setValue(f);
14  }
15
```

Code très simplifié. Toutes les références à ui ont été supprimées. La gestion du destructeur est également enlevée.



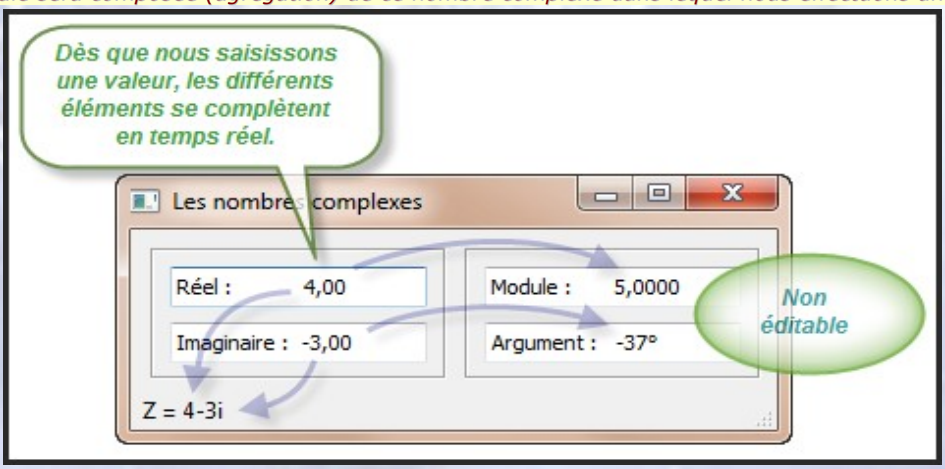
x PRIVILÉGIER L'HÉRITAGE TOUT EN CONSERVANT LE MÉCANISME D'AGRÉGATION

Au vu de ce que nous venons de décrire, nous pourrions nous demander, à juste titre, si l'agrégation possède encore un intérêt.

- *En réalité, la simplification précédente consiste à ne pas détacher la fenêtre principale d'une application de sa partie visuelle, c'est-à-dire de son **IHM**.*
- *Pour le reste, il est évident que la partie visible, l'**IHM**, sera composée de certain nombre d'objets, comme les zones de saisie, les boutons, les labels, etc. et je le rappelle, une composition n'est autre qu'une agrégation forte.*
- *Ainsi pour répondre à la question précédente, l'héritage et l'agrégation peuvent tout-à-fait coexister pour répondre à la plupart des applications souhaitées.*

Afin d'illustrer ces propos, je vous invite à réaliser une nouvelle application qui nous permettra de connaître automatiquement le module et l'argument d'un nombre complexe au travers de la saisie de ses valeurs réelle et imaginaire.

Nous profiterons de ce TP pour voir un exemple concret de la création d'une classe complète vue en cours, la classe Complexe. Ainsi, l'application principale sera composée (agrégation) de ce nombre complexe dans lequel nous effectuons un certain nombre de calcul.



complexe.h — Complexe — Ligne : 14, Col : 28

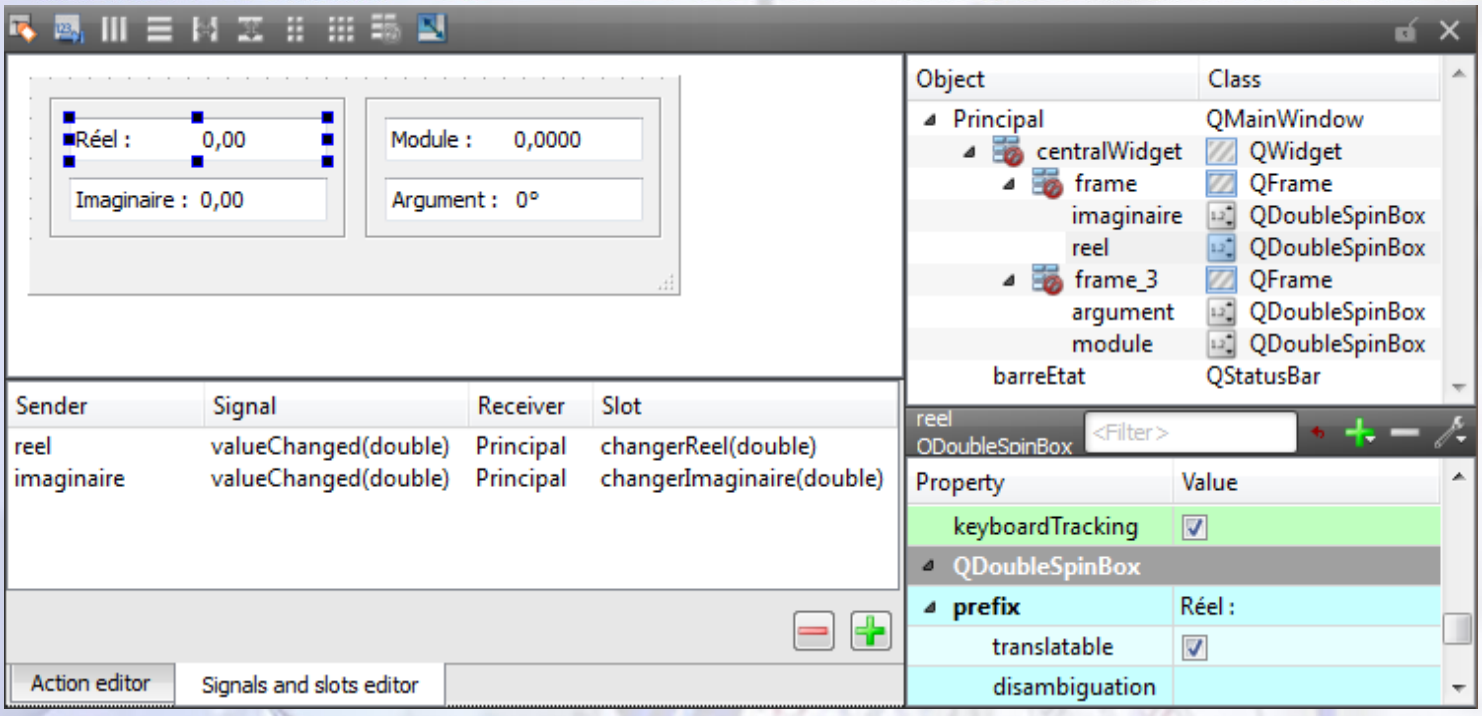
```
1   #ifndef COMPLEXE_H
2   #define COMPLEXE_H
3
4   class Complexe
5   {
6       double reel, imaginaire;
7   public:
8       Complexe();
9       double getReel() const;
10      double getImaginaire() const;
11      void setReel(double r);
12      void setImaginaire(double i);
13      double module() const;
14      double argument() const;
15  };
16
17  #endif // COMPLEXE_H
18
```

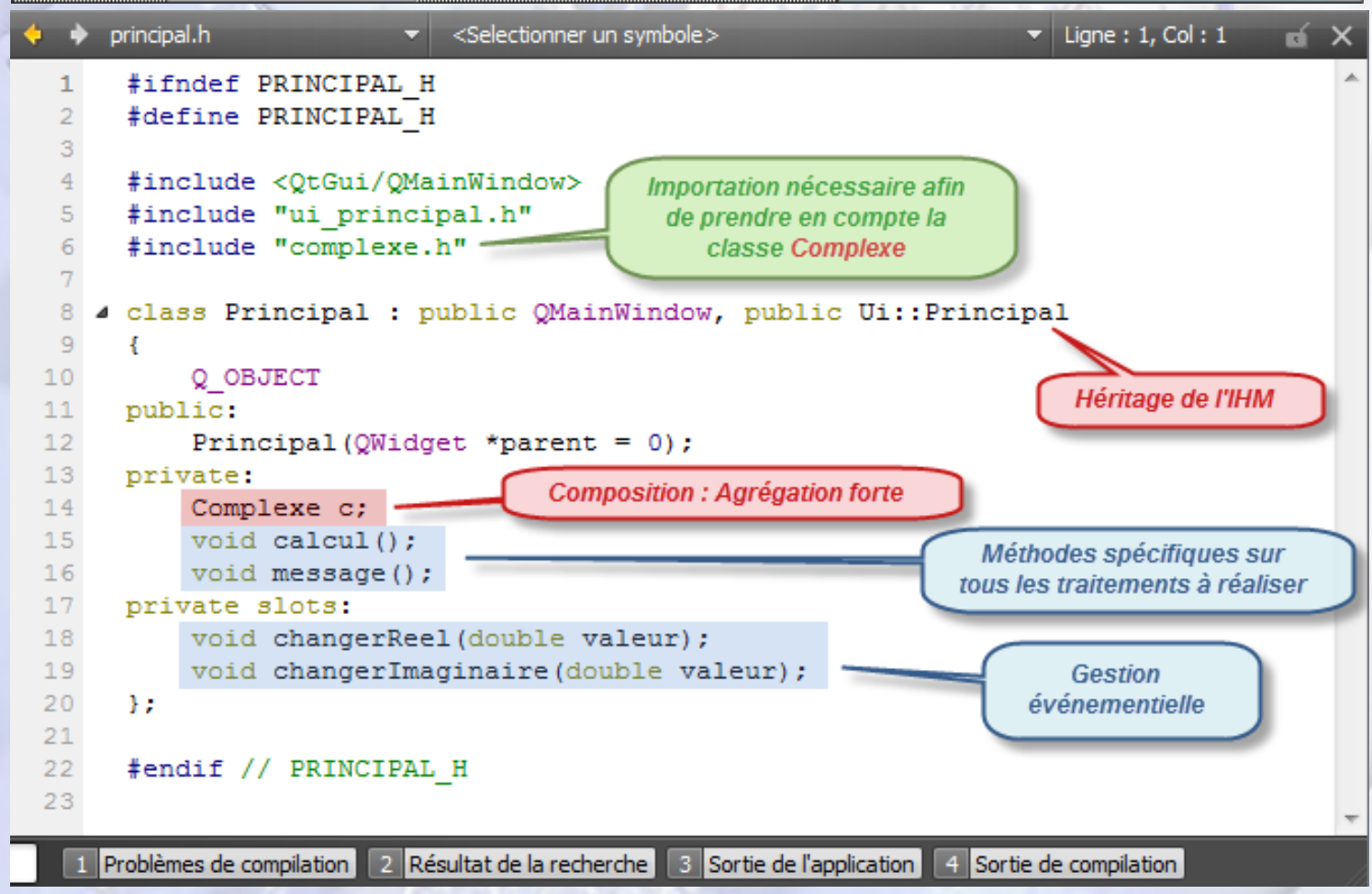
1 Problèmes de compilation | 2 Résultat de la recherche | 3 Sortie de l'application | 4 Sortie de compilation

complexe.cpp — Complexe::argument() const — Ligne : 23, Col : 34

```
1   #include "complexe.h"
2   #include <math.h>
3
4   Complexe::Complexe()
5   {
6     reel = imaginaire = 0;
7   }
8
9   double Complexe::getReel() const       { return reel; }
10  double Complexe::getImaginaire() const { return imaginaire; }
11  void Complexe::setReel(double r)       { reel = r; }
12  void Complexe::setImaginaire(double i) { imaginaire = i; }
13
14  double Complexe::module() const
15  {
16    return sqrt(reel*reel + imaginaire*imaginaire);
17  }
18
19  double Complexe::argument() const
20  {
21    if (imaginaire == 0.0) return 0.0;
22    else if (reel == 0.0) return imaginaire>0.0 ? M_PI_2 : -M_PI_2;
23    return atan(imaginaire / reel);
24  }
25
```

1 Problèmes de compilation | 2 Résultat de la recherche | 3 Sortie de l'application | 4 Sortie de compilation


Réel : 0,00
Imaginaire : 0,00
Module : 0,0000
Argument : 0°
Object
Class
Principal
QMainWindow
centralWidget
QWidget
frame
QFrame
imaginaire
QDoubleSpinBox
reel
QDoubleSpinBox
frame_3
QFrame
argument
QDoubleSpinBox
module
QDoubleSpinBox
barreEtat
QStatusBar
Sender
Signal
Receiver
Slot
reel
valueChanged(double)
Principal
changerReel(double)
imaginaire
valueChanged(double)
Principal
changerImaginaire(double)
reel
QDoubleSpinBox
<Filter>
Property
Value
keyboardTracking
QDoubleSpinBox
prefix
Réel :
translatable
disambiguation
Action editor
Signals and slots editor


principal.h
<Selectionner un symbole>
Ligne : 1, Col : 1
#ifndef PRINCIPAL_H
#define PRINCIPAL_H
#include <QtGui/QMainWindow>
#include "ui_principal.h"
#include "complexe.h"
class Principal : public QMainWindow, public Ui::Principal
{
Q_OBJECT
public:
Principal(QWidget *parent = 0);
private:
Complexe c;
void calcul();
void message();
private slots:
void changerReel(double valeur);
void changerImaginaire(double valeur);
};
#endif // PRINCIPAL_H
Importation nécessaire afin de prendre en compte la classe Complexe
Héritage de l'IHM
Composition : Agrégation forte
Méthodes spécifiques sur tous les traitements à réaliser
Gestion événementielle
1 Problèmes de compilation
2 Résultat de la recherche
3 Sortie de l'application
4 Sortie de compilation

principal.cpp | Principal::calcul() | Ligne : 24, Col : 34

```
1  #include <math.h>
2  #include "principal.h"
3
4  Principal::Principal(QWidget *parent) : QMainWindow(parent)
5  {
6    setupUi(this);
7  }
8
9  void Principal::changerReel(double valeur)
10 {
11   c.setReel(valeur);
12   calcul();
13 }
14
15 void Principal::changerImaginaire(double valeur)
16 {
17   c.setImaginaire(valeur);
18   calcul();
19 }
20
21 void Principal::calcul()
22 {
23   module->setValue(c.module());
24   argument->setValue(c.argument() / M_PI * 180);
25   message();
26 }
27
28 void Principal::message()
29 {
30   double r = c.getReel();
31   double i = c.getImaginaire();
32   barreEtat->showMessage(QString("Z = %1%2%3i").arg(r).arg(i>=0 ? "+" : "").arg(i));
33 }
```

x TRAVAUX PRATIQUES EN AUTONOMIE

Vous allez maintenant valider toutes ces nouvelles compétences pour réaliser un projet plus complet . Il s'agit de réaliser des calculs de base sur deux nombres complexes. Dès que vous saisissez une valeur quelconque, le résultat se répercute sur les autres champs, mais également dans la barre d'état.