ISIMA

PARTIE V

Interfaces graphiques avec Qt

Luc Touraille
Christophe Duhamel
Bruno Bachelet

Bibliothèque Qt

- Sortie de Qt 1.0 en 1996
  - Par la société Trolltech (puis Qt Software, et maintenant Nokia)
  - Bibliothèque objet de composants graphiques
  - Mais bien d'autres choses: réseau, BDD, XML…
- KDE initié en 1997
  - Reposant sur les composants Qt
  - Problème de licence incompatible GNU
    $\Rightarrow$ création de GTK pour Gnome
- Actuellement en licence double
  - Version gratuite : licence libre (LGPL ou GPL)
  - Version payante : licence commerciale + support & mises à jour
- Portable sur différentes plateformes
  - Linux
  - MS Windows
  - Mac OS
- Actuellement version 4.7

Interfaces graphiques

- Histoire
  - 1973 : Premier ordinateur avec interface graphique, Xerox Alto
  - 1980 : Premier système graphique populaire, Apple II
  - 1984 : X11 (libre, en C)
  - 1989 : NextStep (système / interface objet)
  - 1990 : Windows 3.1 (MFC essentiellement en C au début)
  - 1995 : Java avec AWT, puis Swing

- Actuellement
  - C++ : Qt, GTK+
  - Java : Swing
  - C# : composants .net
  - C : X11, GTK, TCL/TK

Interfaces graphiques objets *(1/2)*

- Ensemble de composants graphiques
  - Appelés aussi *«widgets»*
  - Bibliothèque ⇒ widgets de base
  - Un type de widget = une classe
  - Réutilisation
    - Par héritage: extension d'un type de widget
    - Par composition: assemblage de widgets

- Type de composants
  - Widgets de haut niveau
    - Fenêtre, boîte de dialogue…
  - Widgets de bas niveau
    - Bouton, label, zone de texte, case à cocher, bouton radio…
  - Composants invisibles
    - Actions, événements (e.g. clic de souris), conteneur…

Interfaces graphiques objets *(2/2)*

- Interaction entre les composants
  - Inhérent à tout système objet
    - Mécanisme classique des messages
  - Mécanisme de message basé sur les événements
    - Un objet subit un événement ⇒ répercussion sur d'autres

- Gestion des événements
  - Mécanisme des «écouteurs» (e.g. Java / Swing)
    - Utilisation du design pattern observateur
    - Les écouteurs s'enregistrent auprès du widget qu'ils surveillent
    - Le widget subit un événement ⇒ les écouteurs sont informés
  - Mécanisme des «signaux» proposé par Qt
    - Liaison entre deux méthodes: le «signal» et le «slot»
    - Signal déclenché ⇒ slot appelé

Exemple simple en Qt *(1/7)*

- Fichier «**NumberDisplay.hpp**»


```
#ifndef NUMBERDISPLAY_HPP
#define NUMBERDISPLAY_HPP

#include <QtGui/QWidget>

class NumberDisplay : public QWidget
{
    Q_OBJECT
public:
    NumberDisplay(QWidget * parent = 0);
};

#endif // NUMBERDISPLAY_HPP
```

Exemple simple en Qt *(2/7)*

- Extension d'un widget par héritage
  - Ici, extension de **`QWidget`**, classe de base de tous les widgets

- En plus de l'héritage, la macro **`Q_OBJECT`**
  - Indique que la classe représente un objet Qt
  - Rajoute des membres permettant de supporter le mécanisme de messages Qt (entre autres)

- Un code Qt ne se compile pas directement
  - Phase de précompilation supplémentaire
    - Génération du code des membres ajoutés par **`Q_OBJECT`**
    - Génération du code de gestion des signaux/slots
    - Par le programme **`moc`**
      - A partir du fichier «**`NumberDisplay.hpp`**»
      - Génération du fichier «**`moc_NumberDisplay.cpp`**»

Exemple simple en Qt *(3/7)*

- Fichier «**NumberDisplay.cpp**»

```
#include "NumberDisplay.hpp"
#include <QtGui/QLCDNumber>
#include <QtGui/QSlider>
#include <QtGui/QVBoxLayout>

NumberDisplay::NumberDisplay(QWidget *parent)
    : QWidget(parent)
{
    QLCDNumber * number = new QLCDNumber;
    QSlider * slider = new QSlider(Qt::Horizontal);
    QVBoxLayout * mainLayout = new QVBoxLayout;
    mainLayout->addWidget(number);
    mainLayout->addWidget(slider);
    setLayout(mainLayout);

    connect(slider, SIGNAL(valueChanged(int)),
            number, SLOT(display(int)));
}
```

Exemple simple en Qt *(4/7)*

- Création de deux widgets
  - La barre **`QSlider`**
  - L'affichage LCD **`QLCDNumber`**

- Remarque: «**`new`**» mais aucun «**`delete`**»
  - Le **`QWidget`** se charge de la destruction de ses composants
  - Destruction parent ⇒ destruction enfants

- Mécanisme de messages
  - Méthode **`connect()`** relie deux méthodes
    - Signal: méthode déclencheuse
    - Slot: méthode déclenchée
  - Deux macros **`SIGNAL`** et **`SLOT`**
  - Lorsque **`slider.valueChanged(int)`** déclenchée ⇒ **`number.display(int)`** exécutée

Exemple simple en Qt *(5/7)*

- Fichier «**main.cpp**»

```
#include <QtGui/QApplication>
#include "NumberDisplay.hpp"

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

    NumberDisplay w;
    w.show();

    return a.exec(); // boucle principale
}
```

Exemple simple en Qt *(6/7)*

- Fichier projet «**NumberDisplay.pro**»

```
QT += core gui # modules Qt requis

TARGET = NumberDisplay # nom de l'exécutable
TEMPLATE = app         # type de projet

SOURCES += main.cpp NumberDisplay.cpp
HEADERS += NumberDisplay.hpp

LIBS += -L/usr/local/lib –lboost_regex
INCLUDEPATH += /usr/local/boost_1_47_0
```

- Possibilité de générer un squelette de fichier projet
  - **qmake -project**

Exemple simple en Qt *(7/7)*

- Pré-construction : **`qmake`**
  - Génération d'un makefile spécifique à l'environnement, avec les paramètres du fichier projet
  - Support de plusieurs compilateurs
- Construction : **`make`**
  - Invocation de **`qmake`** si fichier projet modifié
  - Appel du préprocesseur **`moc`**
    - Création du fichier «**`moc_NumberDisplay.cpp`**»
  - Compilation du projet
- Possibilité d'utiliser un autre moteur de production (Autotools, CMake, ...)

Classe *QObject*

- Qt est une architecture objet
  - Repose sur la classe **`QObject`** et l'outil **`moc`**
- En dérivant de la classe **`QObject`**
  - Gestion de la mémoire facilitée
  - Mécanisme des signaux et slots
  - Mécanisme des propriétés
  - Introspection avec la classe **`QMetaObject`**
- **`moc`** permet l'implémentation de ces mécanismes

Gestion de la mémoire *(1/4)*

- Arborescence d'objets
  - Un `QObject` peut avoir un parent et des enfants (`QObject`s)
  - Un parent "possède" ses enfants : il les détruit quand il est détruit

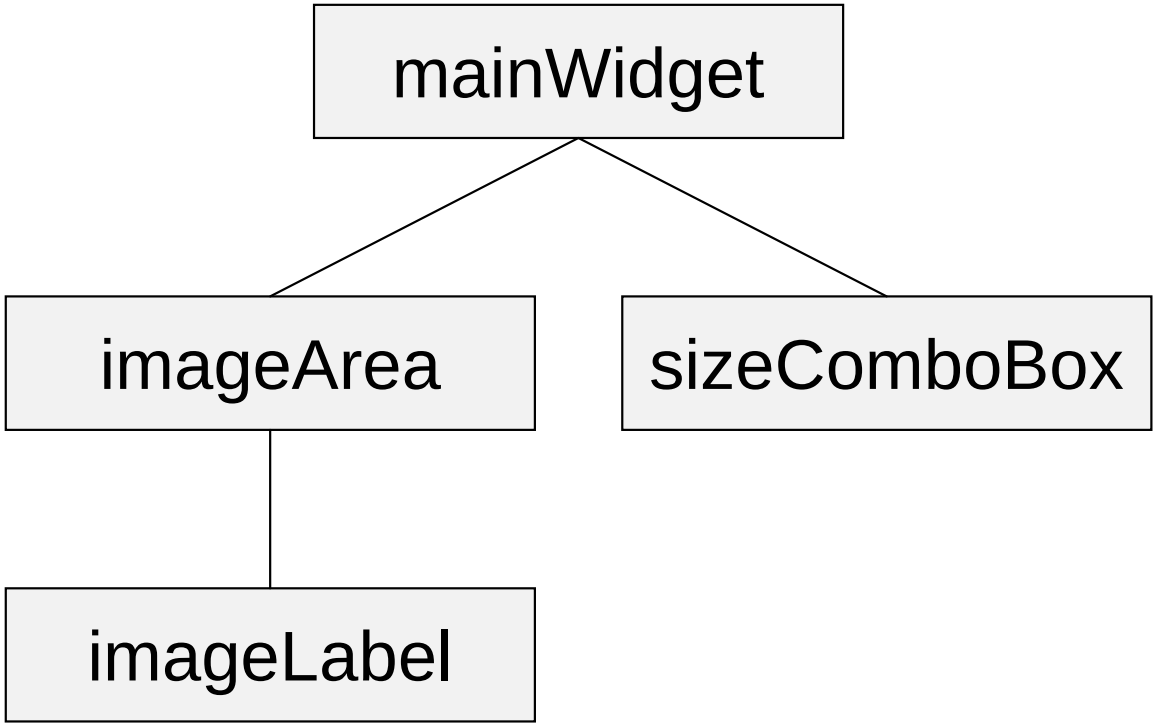
  ⇒ Allocation dynamique des objets (sauf éventuellement les racines)

- Très utile pour les interfaces graphiques
  - Imbrication de widgets

- Affectation du parent
  - À la construction
  - Lors de l'ajout à un conteneur

Gestion de la mémoire *(2/4)*

- Exemple d'arborescence



Gestion de la mémoire *(3/4)*

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

    QLabel * imageLabel = new QLabel;
    imageLabel->setPixmap(QPixmap("arbre.jpg"));

    QScrollArea * imageArea = new QScrollArea;
    imageArea->setWidget(imageLabel);
    // imageArea parent de imageLabel

    QComboBox * sizeComboBox = new QComboBox;
    sizeComboBox->addItem("100x200");
    sizeComboBox->addItem("200x400");
    [...]
```

Gestion de la mémoire *(4/4)*

```
    [...]
    QVBoxLayout * layout = new QVBoxLayout;
    layout->addWidget(imageArea);
    layout->addWidget(sizeComboBox);

    QWidget mainWidget;
    mainWidget.setLayout(layout);
    // mainWidget parent de imageArea et de sizeComboBox

    mainWidget.show();
    return a.exec();
    // destruction de mainWidget
    //    ⇒ destruction de sizeComboBox
    //    ⇒ destruction de imageArea
    //       ⇒ destruction de imageLabel
}
```

Signaux et slots *(1/5)*

- Signaux et slots rendent les composants réutilisables
- Mécanisme qui permet de relier librement les interfaces de composants
- Signal associé à un événement sur un objet
  - Signal = «méthode sans code» de l'objet
  - Evénement = clic souris, touche clavier…
- Un signal est relié à un ou plusieurs slots
  - Slots = méthodes sur d'autres objets
  - Evénement se produit ⇒ appel des slots connectés
- Un signal peut inclure des valeurs
  - Exemple précédent: valeur de la réglette transmise à l'afficheur

Signaux et slots *(2/5)*

- Exemple «**Recepteur.hpp**»

```
#include <QtCore/QObject>

class Recepteur : public QObject
{
    Q_OBJECT
public slots:
    void recevoir(int value)
    {
        std::cout << "Reception : " << value << '\n';
    }
};
```

- **Q_OBJECT** nécessaire dès qu'un dispositif Qt est utilisé
  - A l'exception de la gestion de la mémoire
- Sections «**slots**» pour lister les slots (publics, protégés, privés)

Signaux et slots *(3/5)*

- Exemple «**Compteur.hpp**»

```
class Compteur : public QObject
{
    Q_OBJECT
public:
    explicit Compteur(int valeurInitiale = 0)
    void incrementer();
signals:
    void valeurIncrementee(int nouvelleValeur);
private:
    int valeur_;
};

Compteur::Compteur(int valeurInitiale)
    : valeur_(valeurInitiale)
{}

void Compteur::incrementer()
{
    ++valeur_;
    emit valeurIncrementee(valeur_);
}
```

Signaux et slots *(4/5)*

- Exemple «**Valeur.hpp**»

```
class Valeur : public QObject
{
    Q_OBJECT
public:
    Valeur();
    void set(int nouvelleValeur);
signals:
    void valeurModifiee(int nouvelleValeur);
private:
    int valeur_;
};

Valeur::Valeur()
    : valeur_(0)
{}

void Valeur::set(int nouvelleValeur)
{
    valeur_ = nouvelleValeur;
    emit valeurModifiee(valeur_);
}
```

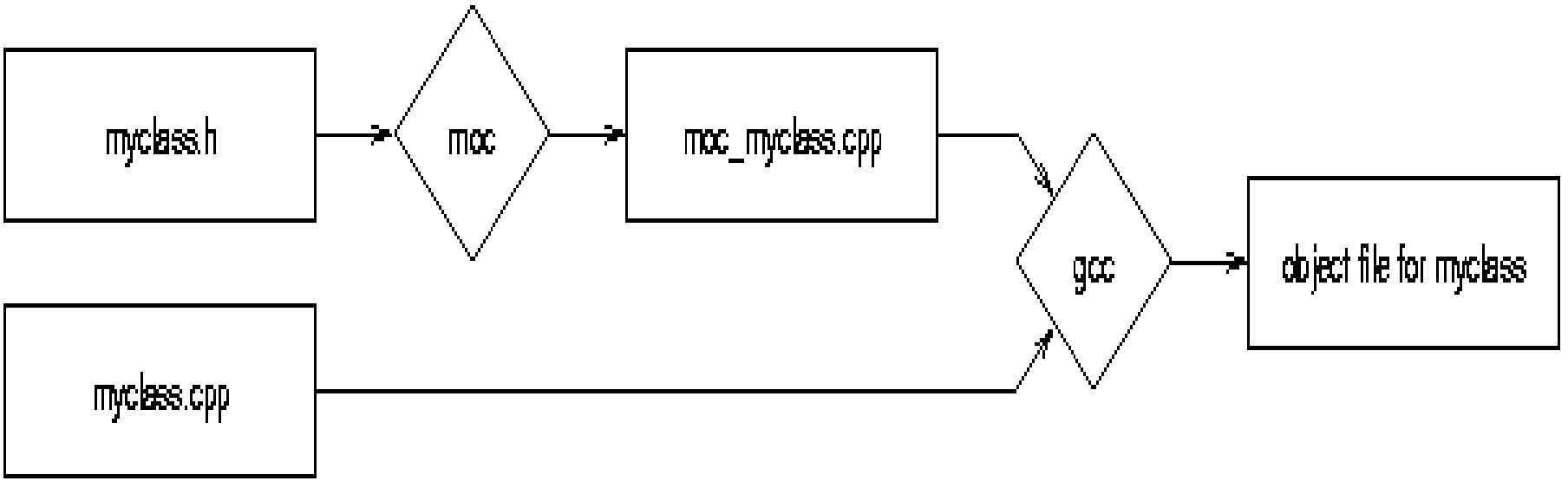
Signaux et slots *(5/5)*

- Exemple de connexion signaux-slots

```
#include <QtCore/QCoreApplication>
#include "Compteur.hpp"
#include "Recepteur.hpp"
#include "Valeur.hpp"

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

    Recepteur r;
    Compteur c(10);
    Valeur v;

    QObject::connect(&c, SIGNAL(valeurIncrementee(int)),
                     &r, SLOT(recevoir(int)));
    QObject::connect(&v, SIGNAL(valeurModifiee(int)),
                     &r, SLOT(recevoir(int)));

    c.incrementer(); // affichage de "Reception 11"
    v.set(42);       // affichage de "Reception 42"
}
```

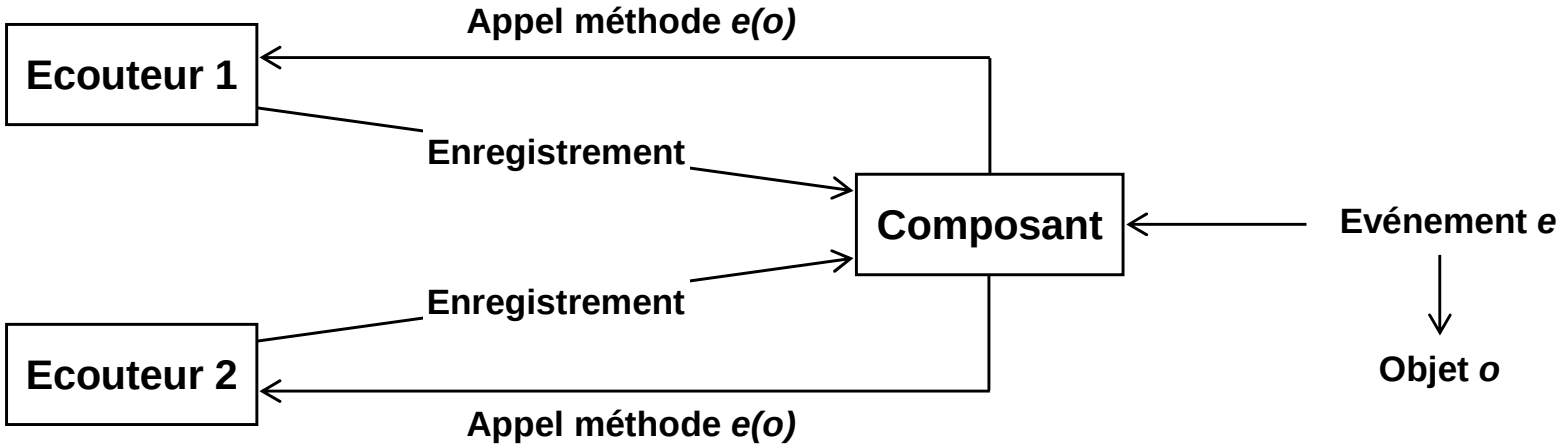
Outil *moc*

- Prétraitement des fichiers entêtes de classes **`QObject`**
  ⇒ préprocesseur **`moc`**



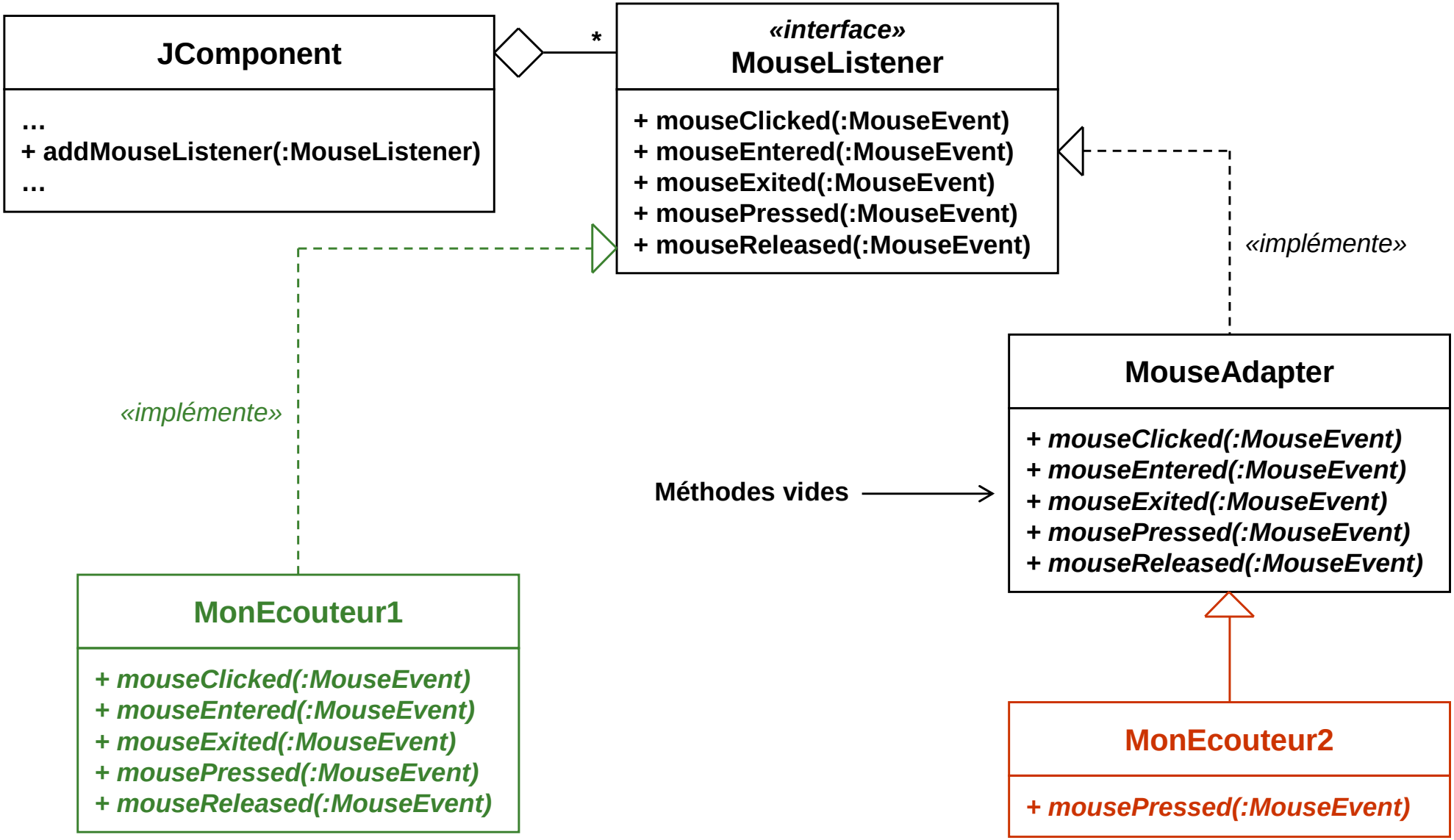
- Application Qt = Application 100% C++
  ⇒ les mot-clés de Qt doivent être remplacés

Ecouteurs: exemple de Swing *(1/2)*



- Capter un événement ⇒ observer, «écouter»
  - Objet chargé d'écouter: «écouteur» (ou «*listener*»)
- Ecouter ⇒ s'enregistrer auprès de l'objet surveillé
- Evénement *e* déclenché sur l'objet ⇒ tous les écouteurs informés
  - Informations sur l'événement encapsulées dans un objet *o*
  - Objet *o* transmis à tous les objets écouteurs
    - Méthode *e()* appelée pour tous les écouteurs
    - Objet *o* passé en paramètre
- Les écouteurs doivent donc respecter une interface commune
  - Exemples: `MouseListener`, `KeyListener`, `WindowListener`…

Ecouteurs: exemple de Swing *(2/2)*



Propriétés d'un *QObject (1/3)*

- Un objet Qt peut avoir des propriétés
  - Ce sont des attributs
  - Qui ont au moins un accesseur en lecture
  - Et éventuellement un accesseur en écriture

- Permet de l'introspection / de la réflexivité
  - Introspection = accéder à des informations sur une classe
    - Héritage, liste des méthodes, liste des attributs

- QtDesigner les utilise pour fournir un inspecteur
  - N'importe quel objet peut accéder aux propriétés
    - Lister
    - Consulter
    - Modifier

Propriétés d'un *QObject (2/3)*

```
class ZoneDeTexte : public QObject
{
    Q_OBJECT
    Q_PROPERTY( QString texte READ texte ) // lecture seule
    Q_PROPERTY( Couleur couleur            // lecture/ecriture
                READ couleur WRITE setCouleur )
    Q_ENUMS( Couleur )
public:
    enum Couleur { Rouge, Vert, Bleu };

    ZoneDeTexte(const QString & texte, Couleur couleur)
        : texte_(texte), couleur_(couleur) {}

    QString texte() const { return texte_; }
    Couleur couleur() const { return couleur_; }
    void setCouleur(Couleur couleur) { couleur_ = couleur; }
private:
    QString texte_;
    Couleur couleur_;
};
```

Propriétés d'un *QObject (3/3)*

- Exemple d'introspection

```
ZoneDeTexte z("bla bla", ZoneDeTexte::Bleu);
QObject * o = &z;

cout << o->property("texte").toString().toStdString();

o->setProperty("couleur", "Rouge");

QMetaEnum couleurEnum =
    ZoneDeTexte::staticMetaObject.enumerator(0);
cout << couleurEnum.valueToKey(z.couleur());
```

Introspection

- C++ ne fournit pas de véritable mécanisme d'introspection
  - Il existe le RTTI (*Run-Time Type Information*)
  - Mais il est très limité
    - Reconnaissance à l'exécution d'un type
    - Mais impossible de lister les méthodes d'une classe par exemple

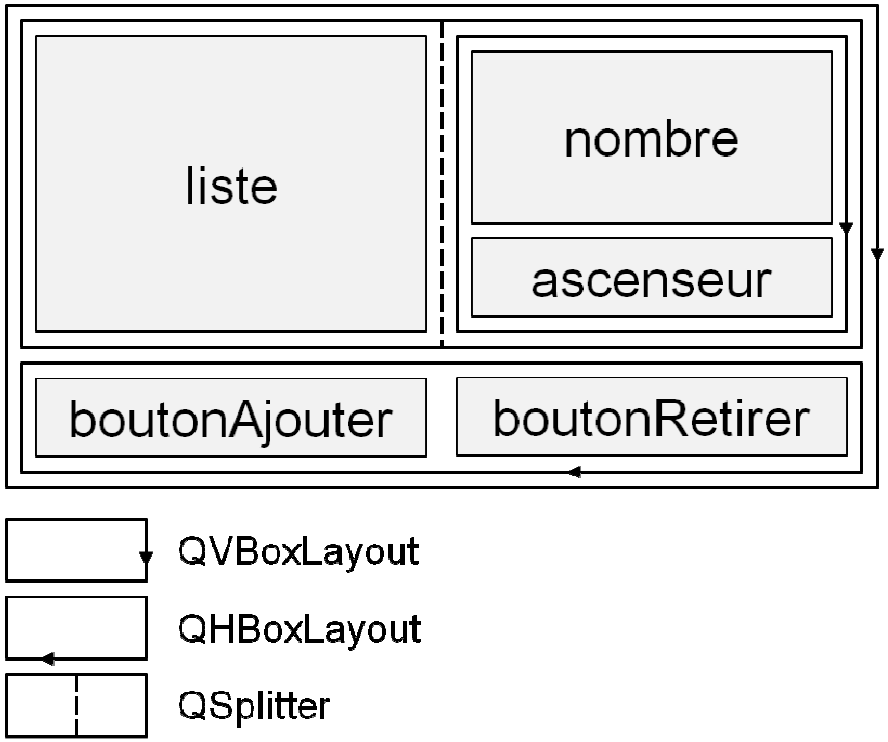
- Qt associe à chaque objet un méta-objet
  - Objet de la classe **`QMetaObject`**
  - Accessible par la méthode **`metaObject()`** ou l'attribut statique **`staticMetaObject`**
  - Possède les méthodes suivantes
    - **`className()`**
    - **`superClass()`**
    - **`constructor(index)`**
    - **`method(index)`**
    - **`property(index)`**
    - …

Exemple d'une application faite main *(1/7)*

- QtDesigner permet de construire des interfaces
  - Environnement de développement graphique
  - Code généré automatiquement

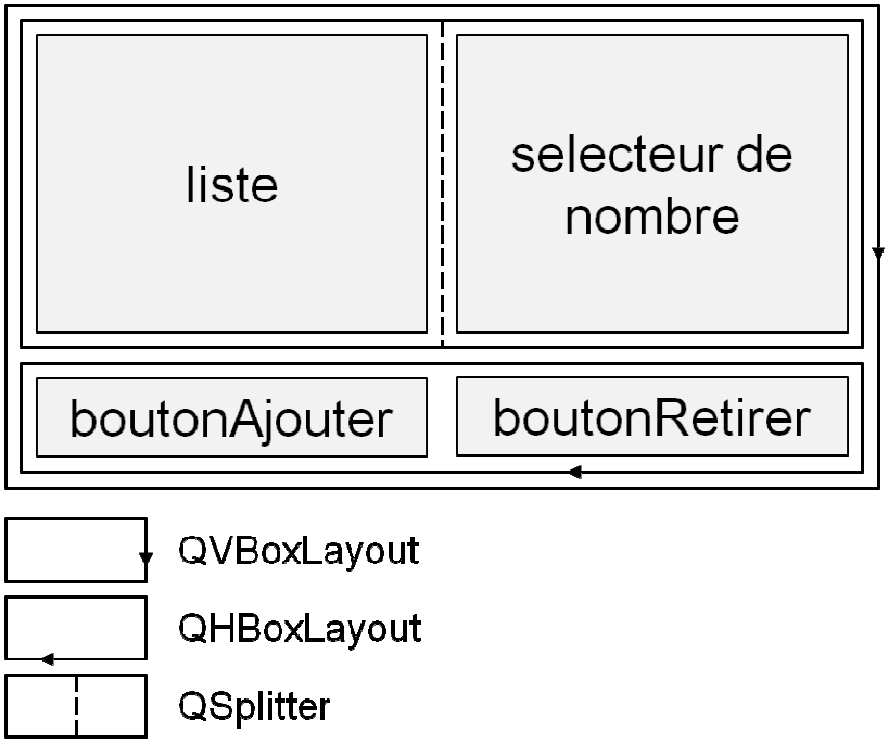
- Essayons de voir comment faire sans

- Objectif



Exemple d'une application faite main *(2/7)*

- Utilisation d'un "sous-widget" : **`SelecteurDeNombre`**
- **`QSplitter`** pour permettre le redimensionnement du sélecteur et de la liste
- Gestionnaires de placement pour le reste



Exemple d'une application faite main *(3/7)*

- Fichier «**SelecteurDeNombre.hpp**»

```
#include <QtGui/QLCDNumber>
#include <QtGui/QSlider>

class SelecteurDeNombre : public QWidget
{
    Q_OBJECT
public:
    explicit SelecteurDeNombre(QWidget *parent = 0);
    int nombreCourant() const;

private:
    QLCDNumber * nombre_;
    QSlider * ascenseur_;
};
```

Exemple d'une application faite main *(4/7)*

- Fichier «**SelecteurDeNombre.cpp**»

```
SelecteurDeNombre::SelecteurDeNombre(QWidget *parent)
    : QWidget(parent)
{
    nombre_ = new QLCDNumber;
    ascenseur_ = new QSlider(Qt::Horizontal);
    QVBoxLayout * layout = new QVBoxLayout;
    layout->addWidget(nombre_);
    layout->addWidget(ascenseur_);
    setLayout(layout);
    connect(ascenseur_, SIGNAL(valueChanged(int)),
            nombre_, SLOT(display(int)));
}

int SelecteurDeNombre::nombreCourant() const
{
    return nombre_->value();
}
```

Exemple d'une application faite main *(5/7)*

- Fichier «**ManagerDeNombre.hpp**»

```
class ManagerDeNombre : public QWidget
{
    Q_OBJECT
public:
    explicit ManagerDeNombre(QWidget *parent = 0);

private slots:
    void ajouterNombre();
    void retirerNombre();

private:
    QListWidget * liste_;
    SelecteurDeNombre * selecteur_;
};
```

Exemple d'une application faite main *(6/7)*

- Fichier « **ManagerDeNombre.cpp** » *(1/2)*

```
ManagerDeNombre::ManagerDeNombre(QWidget *parent)
    :  QWidget(parent)
{
    liste_ = new QListWidget;
    selecteur_ = new SelecteurDeNombre;
    QPushButton * boutonAjouter = new QPushButton("Ajouter");
    QPushButton * boutonRetirer = new QPushButton("Retirer");

    QSplitter * splitterHaut = new QSplitter(Qt::Horizontal);
    splitterHaut->addWidget(liste_);
    splitterHaut->addWidget(selecteur_);

    QHBoxLayout * layoutBas = new QHBoxLayout;
    layoutBas->addWidget(boutonAjouter);
    layoutBas->addWidget(boutonRetirer);
    [...]
```

Exemple d'une application faite main *(7/7)*

- Fichier « **ManagerDeNombre.cpp** » (2/2)

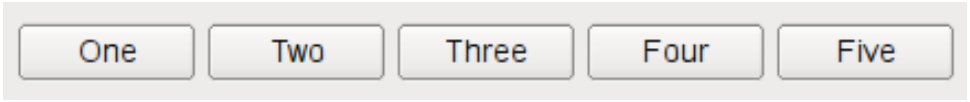
```
    [...]
    QVBoxLayout * layoutPrincipal = new QVBoxLayout;
    layoutPrincipal->addWidget(splitterHaut);
    layoutPrincipal->addLayout(layoutBas);
    setLayout(layoutPrincipal);

    connect(boutonAjouter, SIGNAL(clicked()),
            this, SLOT(ajouterNombre()));
    connect(boutonRetirer, SIGNAL(clicked()),
            this, SLOT(retirerNombre()));
}

void ManagerDeNombre::ajouterNombre() {
    liste_->addItem(
        QString::number(selecteur_->nombreCourant()));
}

void ManagerDeNombre::retirerNombre() {
    delete liste_->takeItem(liste_->currentRow());
}
```

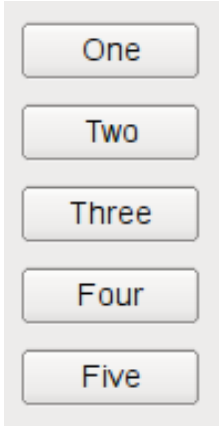
Placement des widgets *(1/2)*

- Gestionnaires de placement = «*Layouts*»
- Classes permettant d'agencer facilement des widgets
  - Positionnement automatique
  - Gestion des tailles
  - Redimensionnement
- Le placement repose sur les propriétés des widgets
  - Taille minimale
  - Taille préférée
  - Politique de dimensionnement
  - Facteur d'étirement

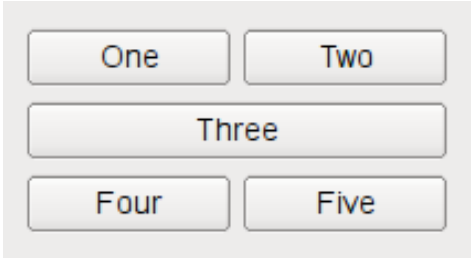
Placement des widgets *(2/2)*

- Quelques layouts
  - Horizontal/vertical : **`QHBoxLayout`**/**`QVBoxLayout`**





  - Grille : **`QGridLayout`**

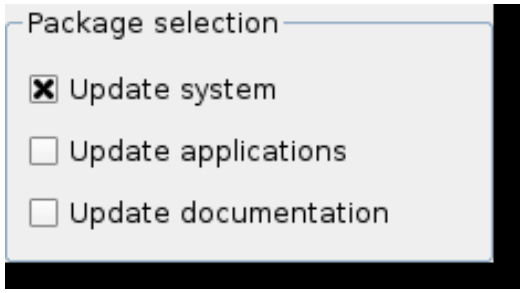


  - Formulaire : **`QFormLayout`**

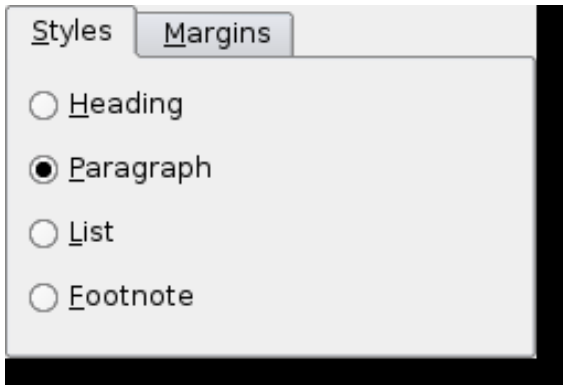
Conteneurs de widgets

- Widgets encapsulant d'autres widgets
  - Fournissant éventuellement des fonctionnalités supplémentaires
    - Signaux
    - Gestion des sélections multiples/uniques
    - …
  - Simple page ou multi-page

- Exemples
  - **`QGroupBox`**

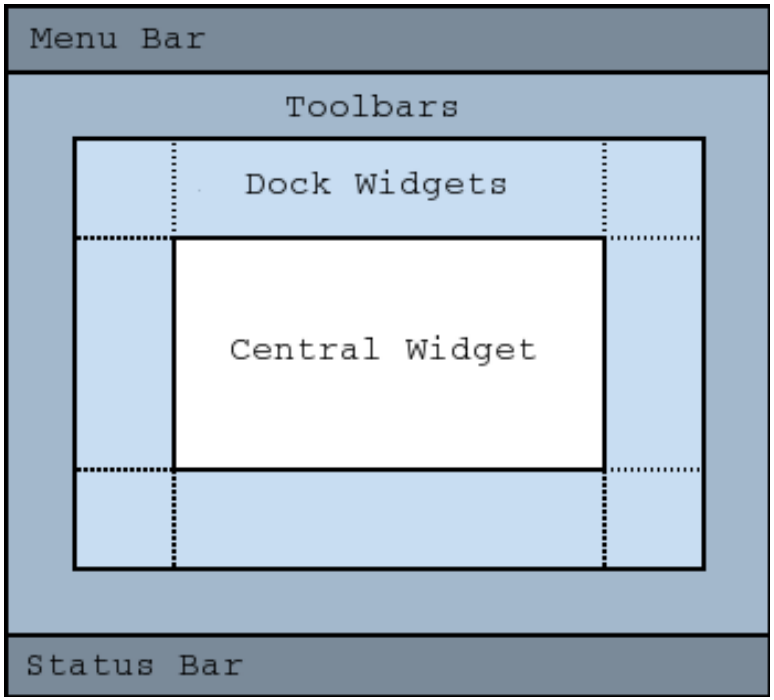


  - **`QTabWidget`**



Fenêtre principale *(1/6)*

- Conteneur de widgets proposant les éléments classiques d'une fenêtre principale
  - Barre de menus
  - Barre d'outils
  - Barre de statut
- Support du mode MDI (*Multiple Document Interface*)
- Système d'ancrage de widgets



Fenêtre principale *(2/6)*

- Possibilité de définir des actions
  - Simplifie le développement de l'interface utilisateur
- Action = Commande invocable de plusieurs façons
  - Entrée de menu
  - Raccourci clavier
  - Icône de barre d'outil
  - Autre : appel direct, connexion à un signal
- Exemple

Fenêtre principale *(3/6)*

- Fichier «**FenetrePrincipale.hpp**»

```
#include <QtGui/QAction>
#include "ManagerDeNombre"

class FenetrePrincipale : public QMainWindow
{
    Q_OBJECT
public:
    explicit FenetrePrincipale(QWidget *parent = 0);

private:
    void creerActions();
    void creerMenus();

    ManagerDeNombre * manager_;

    QAction * exitAction_;
    QAction * resetAction_;
};
```

Fenêtre principale *(4/6)*

- Fichier «**FenetrePrincipale.cpp**» *(1/3)*

```
MainWindow::MainWindow(QWidget * parent)
    : QMainWindow(parent)
{
    manager_ = new ManagerDeNombre;
    setCentralWidget(manager_);

    createActions();
    createMenus();
}
```

Fenêtre principale *(5/6)*

- Fichier «**FenetrePrincipale.cpp**» *(2/3)*

```
void MainWindow::createActions()
{
    exitAction_ = new QAction("&Quitter", this);
    exitAction_->setShortcut(QKeySequence::Close);

    connect(exitAction_, SIGNAL(triggered()),
            this, SLOT(close()));

    resetAction_ = new QAction("&Remettre à zéro", this);
    resetAction_->setIcon(QIcon(":/images/reset.png"));
    resetAction_->setShortcut(QKeySequence("F9"));

    connect(resetAction_, SIGNAL(triggered()),
            manager_, SLOT(reset()));
}
```

Fenêtre principale *(6/6)*

- Fichier « **FenetrePrincipale.cpp** » *(3/3)*

```
void MainWindow::createMenus()
{
    QMenu * fileMenu = menuBar()->addMenu("&Fichier");
    fileMenu->addAction(exitAction_);

    QMenu * toolsMenu = menuBar()->addMenu("&Outils");
    toolsMenu->addAction(resetAction_);

    QToolBar * toolsToolBar = addToolBar("&Outils");
    toolsToolBar->addAction(resetAction_);
}
```

Le Modèle-Vue-Contrôleur *(1/2)*

- MVC, *Model-View-Controller*

- Premier Design Pattern, 1979
  - Travaux sur SmallTalk, laboratoires Xerox PARC
  - Egalement un modèle d'architecture

- Objectif
  - Séparer la présentation (vue) des donnés (modèle)
  - Couplage faible réalisé par un intermédiaire
    - Le contrôleur
    - Il assure la cohérence entre les deux couches

MVC : le modèle

- Couche métier, le cœur de l'application
- Représente le comportement de l'application
  - Contient les données de l'application
  - Effectue les traitements sur ces données
- L'interface du modèle permet
  - La mise à jour des données
  - La consultation des données
- Résultats du modèle dénués de toute présentation

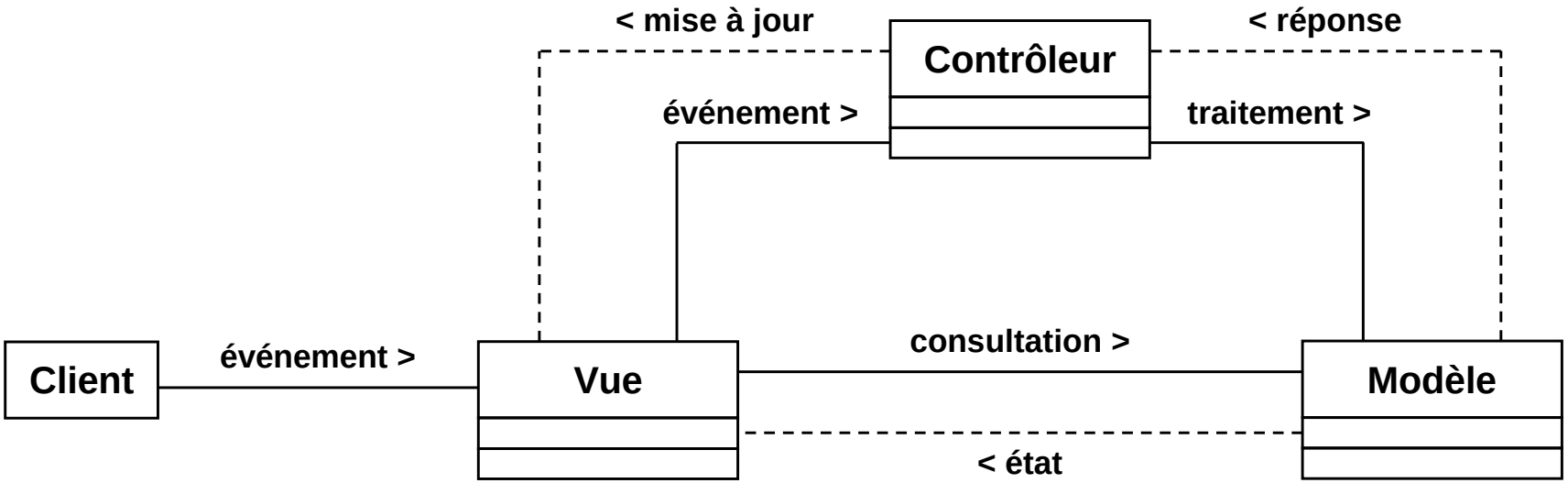
MVC : la vue

- Interface avec l'utilisateur
- 1ère tâche: présenter les résultats du modèle
- 2nde tâche: recevoir toutes les actions de l'utilisateur
  - Ces événements sont redirigés au contrôleur
- La vue n'effectue aucun traitement
- Plusieurs vues possibles pour les mêmes données

MVC : le contrôleur

- Gestion des événements de synchronisation
- Reçoit les événements de l'utilisateur
  - Il analyse la requête
- Et enclenche les actions à effectuer
  - Il enclenche une action sur le modèle
  - Il avertit ensuite la vue de se mettre à jour
  - Parfois, seule la vue est concernée
- Le contrôleur n'effectue aucun traitement et ne modifie aucune donnée

Le Modèle-Vue-Contrôleur *(2/2)*



- Conclusion
  - Maintenance facilitée par le découplage vue(s)-modèle
  - Utilisation des patrons Observateur, Médiateur et Stratégie
  - Utilisé principalement pour les interfaces graphiques
    - Sous forme simplifiée dans Qt et Swing
  - Variantes possibles
    - Contrôleur enlevé
    - Mécanisme supplémentaire: l'inversion de contrôle

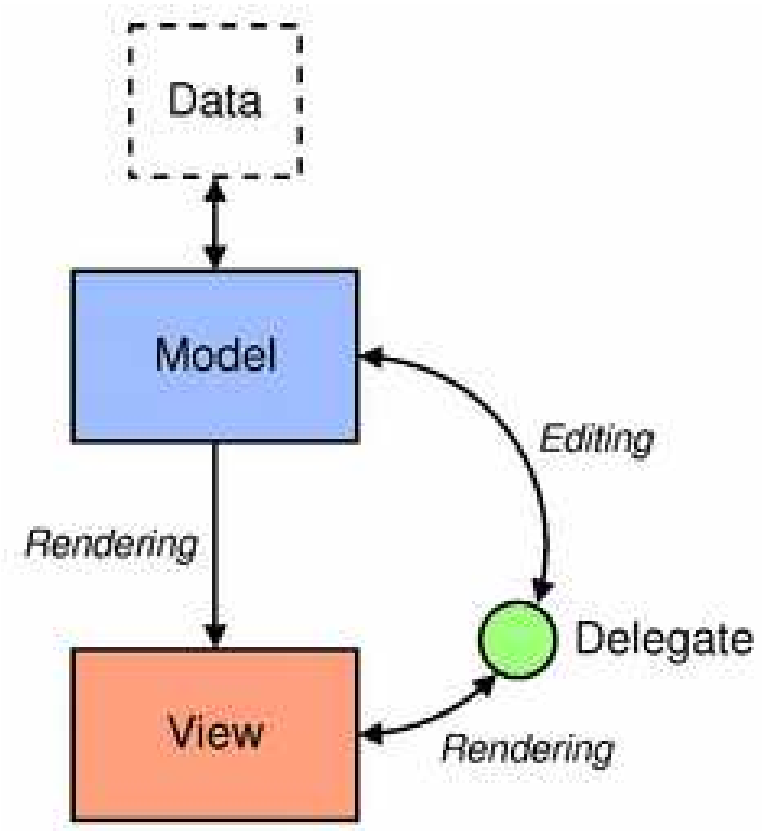
Qt : architecture Modèle-Vue *(1/2)*

- Vue et contrôleur confondus
  - Séparation données / présentations conservée
  - Utilisation plus simple
- Modèles présentent des données avec des interfaces standards
- Vues interrogent les modèles par ces interfaces

$\Rightarrow$ Grande réutilisabilité

Qt : architecture Modèle-Vue *(2/2)*

- Utilisation de délégués pour le rendu et l'édition des données à l'intérieur des vues



Qt : le modèle

- Fournit un accès à des données avec une interface fixée : `QAbstractItemModel`
  - Peut contenir lui-même les données
  - Peut être une façade vers une source autre
- Trois types
  - Données séquentielles (liste)
  - Données tabulaire
  - Données arborescentes
    - Le plus générique : les deux autres sont des cas particuliers
- Fonctionne avec n'importe quelle vue

Qt : la vue

- Présente les données fournies par le modèle
- Permet (éventuellement) de les éditer
- Capture les évènements utilisateur,
  gère les sélections de données,
  …
- Dépend d'une interface
  ⇒ peut être réutilisée avec de nombreux de modèles

Modèle-Vue : conseils en Qt

- Pour les cas simples :
  utiliser les «classes de commodité»
  - Vue et modèle «mélangés»
  - Ajout/suppression de données directement dans la vue
  - Exemple : `QListWidget`

- Pour les cas moins simples :
  utiliser un modèle prédéfini et une vue prédéfinie
  - Exemple : `QFileSystemModel` avec `QListView`

- Pour les cas complexes :
  utiliser un modèle personnalisé et/ou une vue personnalisée
  - Implémenter une interface de modèle
  - Hériter d'une vue existante

Conclusion

- Références
  - `http://www.digitalfanatics.org/projects/qt_tutorial/fr`
  - *«C++ GUI Programming with Qt 4»*
    Jasmin Blanchette et Mark Summerfield
  - Documentation officielle (d'excellente qualité)
    `http://doc.qt.nokia.com`
- Qt s'interface avec OpenGL
- Qt est plus qu'une bibliothèque graphique
  - Structures de données
  - Fichiers, répertoires et flux
  - XML
  - Programmation concurrente
- Concurrent sur ces aspects: bibliothèques Boost