Présentation de Qt

ENSIIE2 : Option LOA

Jean-Yves Didier

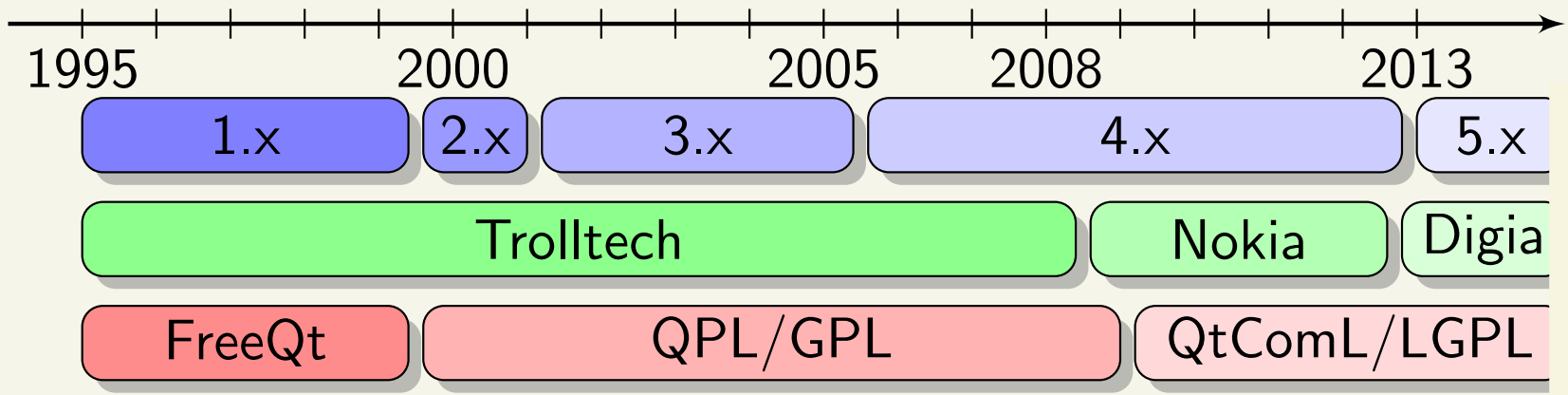
didier@ufrst.univ-evry.fr

Qt en bref

Le *framework* Qt

- Dédié (principalement) aux interfaces graphiques ;
- Développé en C++, donc orienté objet ;
- Ensemble de bibliothèque et d'utilitaires (*framework*) ;
- Portable : GNU/Linux, Windows, MacOSX, SymbianOS, ...

Historique



Qui utilise Qt ?

Parmi les entreprises/gros organismes

- ESA : modélisation et carte numérique de Mars ;
- Siemens : contrôle de ligne de production ;
- Dreamworks : logiciel de rendu ;
- Walt Disney Animation Studios : logiciel d'animation ;
- Google, Hp, Volvo, Thales, le CEA, ...

Logiciels notables

- Maya : modeleur 3D maintenu par AutoDesk ;
- VLC : lecteur multi-média ;
- KDE : environnement de bureau populaire sous Linux ;
- Skype : logiciel de communication.

Le framework Qt

Constituants du *framework*

- Une collection d'exécutables ;
- Une collection de bibliothèques ;
- Une API très fournie (800+ classes) ;
- Une architecture d'applications.

Compléments à l'éco-système Qt

QtJambi Qt pour Java ;

Qtopia Qt pour dispositifs portables et systèmes embarqués ;

QSA applications scriptables ;

Les exécutables

qtcreator	environnement de développement intégré ;
qmake	gestion des projets Qt ;
linguist	internationalisation des applications ;
designer	prototypage d'interfaces graphiques ;
moc	génération de code associé aux méta-objets ;
uic	génération de code associé aux interfaces graphiques.

Les bibliothèques

QtCore	Fonctionnalités communes ;
QtGui	Composants de l'interface graphique ;
QtNetwork	Communication réseau ;
QtOpenGL	Intégration d'OpenGL ;
QtSql	Utilisation de base de données SQL ;
QtXml	Manipulation et génération de fichiers XML ;
QtSVG	Affichage d'images vectorielles au format SVG ;
QtTest	Tests unitaires ;
QtScript	Scripting d'applications ;
QtWebKit	Moteur de rendu web webkit ;
QtHelp	Aide en ligne des applications.

Une première application (1/3)

Etape par étape sous Linux

Les étapes

1. Création d'un fichier de projet ;
2. Création du code source ;
3. Compilation et exécution.

Mise en place d'un projet (fichier *hello.pro*)

```
TEMPLATE=app
SOURCES=hello.cpp
TARGET=hello
```

Une première application (2/3)

Etape par étape sous Linux

Le code source (fichier *hello.cpp*)

```
1  #include <QApplication>
   #include <QPushButton>
3
   int main(int argc,char* argv[]) {
5    QApplication app(argc,argv);
     QPushButton* hello = new QPushButton("Hello␣
         world␣!");
7    QObject::connect(hello, SIGNAL(clicked()), &
         app, SLOT(quit()));
     hello->show();
9    return app.exec();
   }
```

Une première application (3/3)

Etape par étape sous Linux

La compilation et l'exécution

```
$ qmake
$ make
$ ./hello
```

Une classe pour les gouverner toutes

La classe *QObject*

- Classe de base de la hiérarchie d'héritage des classes Qt ;
- Gère des propriétés dynamiques et statiques ;
- Introduit des mécanismes d'introspection ;
- Introduit une hiérarchie compositionnelle (facilite la gestion de la mémoire) ;
- Implémente le mécanisme *signal-slot*.

Propriétés statiques et dynamiques (1/5)

Définition / Utilité

- Propriété : membre d'une classe auquel on adjoint de manière standardisée des accesseurs. Mécanisme présent en C#, Php, Python, Ruby ;
- Utilité :
  - accès au membre sans faire de conversion de type ;
  - utilisation dans des éditeurs spécialisés.
- Deux types de propriétés :
  - Statiques : propriétés connues à la compilation ;
  - Dynamiques : propriétés affectées lors de l'exécution.

Propriétés statiques et dynamiques (2/5)

Propriété à part : le nom

- `objectName()` : retourne le nom ;
- `setObjectName(QString)` : donne un nom à l'objet.

Propriété statiques

- Déclarées par la macro `Q_PROPERTY` ;
- Permet de spécifier :
  - Le nom de la propriété ;
  - L'accesseur en lecture (requis) ;
  - L'accesseur en écriture (optionnel) ;
  - Diverses informations : notifications, revision, etc.

Propriétés statiques et dynamiques (3/5)

Propriété dynamiques

- Reposent sur la classe `QVariant`, sorte de contenant générique pour des valeurs de types différents ;
- Lecture de propriété : `property(char*)`. Retourne un objet de type `QVariant` contenant la valeur (peut-être invalide si la propriété est inexistante) ;
- Ecriture de propriété : `setProperty(char*, QVariant&)`. Affecte à une propriété nommée une valeur stockée dans un `QVariant`. Renvoie `true` si la propriété est statique, `false` sinon.

Propriétés statiques et dynamiques (4/5)

Example de déclaration de propriété statique

```
#include<QColor>
#include<QObject>
class MyColor : public QObject {
  Q_OBJECT
  Q_PROPERTY( QColor color READ color WRITE
      setColor)
public:
  MyColor(QObject* parent=0): QObject(parent) {}
  void setColor(QColor c) { m_color = c; }
  QColor color() const { return m_color; }
private:
  QColor m_color;
};
```

Propriétés statiques et dynamiques (5/5)

Utilisation de la propriété statique

```
#include "mycolor.h"
#include <QVariant>
#include <QColor>
#include <QDebug>
int main(int argc, char* argv[]) {
   MyColor* mc = new MyColor();
   QObject* qmc = mc;
   mc->setColor(Qt::red);
   qDebug() <<  qmc->property("color");
   return 0;
}
```

Hiérarchie compositionnelle

Relations père-fils entre objets

- Permet de gérer la mémoire ;
- Proche du patron de conception composite utilisé dans les interfaces graphiques.

Gestion de la hiérarchie

- `parent()` : retourne le père courant (QObject*) ;
- `setParent(QObject*)` : affecte un nouveau père.
- `findChild(QString name)` : retourne l'enfant avec le nom donné (récursif) ;
- `findChildren(QString name)` : retourne une liste d'enfants avec le nom donné (récursif).

Mécanismes d'introspection

La classe `QMetaObject`

- Tout objet de type `QObject` possède une méthode `metaObject()` qui retourne un objet de type `QMetaObject` ;
- Objet spécial généré lors de la compilation par l'utilitaire `moc` (moc = *Meta Object Compiler*) ;
- Permet de déterminer :
  - le nom de la classe : `className()` ;
  - les constructeurs : `constructor(int)` sous la forme de `QMetaMethod` ;
  - les méthodes : `method(int)` (usuelles, signaux ou slots) sous la forme de `QMetaMethod` ;
  - les propriétés statiques `property(int)` sous la forme de `QMetaProperty` ;
  - le méta-objet décrivant la superclasse : `superClass()`.

Gestion des évènements

Le mécanisme *signal/slot*

- Système de *callbacks* déguisé :
  - Patron de conception *observateur*.
- **Signal** : appel à une fonction de *callback* potentielle ;
- **Slot** : fonction de *callback* ;
- Connection Signal/slot modifiable dynamiquement ;
- Condition pour en bénéficier : hériter de `QObject`.

Créer ses propres signaux/slots

Créer une classe dérivant de QObject

```
#include <qobject.h>
class Toto:public QObject  // ou classe dérivant de
    QObject
{
  Q_OBJECT // nécessaire pour le mécanisme signal/slot
  public:
    Toto( ...);
  public slots: // peuvent être private ou protected
    void slotA(); // void : type de retour imposé
    void slotB(int a);
  signals:
    void signalA();  // void : type de retour imposé
};
```

Créer ses propres signaux/slots

- Un slot a une implémentation ;
- Un signal n'a pas d'implémentation !
- Émettre un signal :

  → `emit monSignal(param1,param2,...);`
- Connecter un signal et un slot :

  → `connect(objectA, SIGNAL(monSignal(int,int)), objectB, SLOT(monSlot(int,int)));`
  - Utilisation des macros `SLOT` et `SIGNAL` ;
  - `objectA` et `objectB` sont de type `QObject*` ;
  - les signatures des signaux et des slots doivent correspondre.

Conteneurs Qt (1/4)

Conteneurs séquentiels

- `QList<T>` : liste indexée d'éléments de type `T` ;
- `QLinkedList<T>` : liste chaînée ;
- `QVector<T>` : liste d'éléments contigus en mémoire ;
- `QStack<T>` et `QQueue<T>` : dérivés de `QVector<T>`.

Tableaux associatifs

- `QMap<T>` et `QMultiMap<T>` : tableaux associatifs simples ;
- `QHash<T>` : table de hachage ;
- `QSet<T>` : gestion d'ensembles de valeurs.

Conteneurs Qt (2/4)

Critères de choix du conteneur : regards sur la complexité

	Recherche par index	Insertion	Insertion en tête	Insertion en queue
`QLinkedList<T>`	O(n)	O(1)	O(1)	O(1)
`QList<T>`	O(1)	O(n)	→ O(1)	→ O(1)
`QVector<T>`	O(1)	O(n)	O(n)	→ O(1)

	Recherche par clé		Insertion	
	Moy.	Pire cas	Moy.	Pire cas
`QMap<Key,T>`	O(log n)	O(log n)	O(log n)	O(log n)
`QMultiMap<Key,T>`	O(log n)	O(log n)	O(log n)	O(log n)
`QHash<Key,T>`	→ O(1)	O(n)	→ O(1)	O(n)
`QSet<Key,T>`	→ O(1)	O(n)	→ O(1)	O(n)

Conteneurs Qt (3/4)

Parcours des conteneurs

Deux styles d'itérateurs :

- Les itérateurs de type C++/STL ;
- Les itérateurs de type Java.

Exemple d'itérateur de type Java

```
QList<QString> list;
list << "A" << "B" << "C" << "D";

QListIterator<QString> i(list);
while (i.hasNext())
    qDebug() << i.next();
```

Conteneurs Qt (4/4)

Exemple d'itérateur de type C++/STL

```
QList<QString> list;
list << "A" << "B" << "C" << "D";

QList<QString>::iterator i;
for (i = list.begin(); i != list.end(); ++i)
    qDebug() << *i;
```

Les widgets Qt

- Pas de liste exhaustive ici, juste les principaux ;
- Dérivent d'une classe `QWidget` qui dérive de `QObject`
  → chacun possède des signaux et des slots !
- Widgets organisés en hiérarchie : les fenêtres, menus et *layout* sont les parents d'autres widgets ;
- Pour chaque widget, nous verrons :
  - le constructeur ;
  - l'aspect ;
  - les principaux signaux et slots.

Les widgets Qt

QWidget : la classe mère

→ `QWidget(QWidget* parent=0);`

- Slots :
  - `void show(); void move(int, int);`
  - `void hide(); void resize(int, int);`
  - `void enabled(bool);`

QLabel : affichage de texte

→ `QLabel(QString text, QWidget* parent=0);`

- Slots :
  - `void setText(Qstring);`
  - `void clear();`

Les widgets Qt

Les boutons

QPushButton : le bouton classique

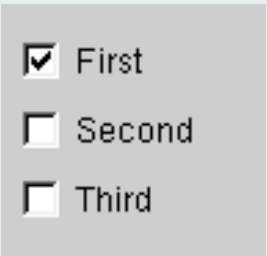
→ `QPushButton(QString text, QWidget* parent=0);`

- Signaux :
  - `void clicked();`

QCheckBox : la case à cocher

→ `QCheckBox(QString text, QWidget* parent=0);`

- Slots :
  - `void setChecked(bool);`
- Signaux :
  - `void clicked();`
  - `void toggled(bool);`



Les widgets Qt

Les boutons

QButtonGroup : grouper les boutons radio

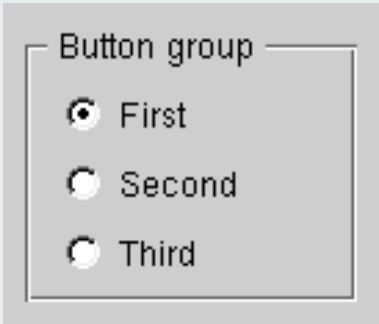
→ `QButtonGroup(QString text, QWidget* parent=0);`

- Signaux :
  - `void buttonClicked(int);`

QRadioButton : le bouton radio

→ `QRadioButton(QString text, QWidget* parent=0);`

- Slots :
  - `void setChecked(bool);`
- Signaux :
  - `void clicked();`
  - `void toggled(bool);`



Les widgets Qt (1/2)

Les listes

`QListWidget` : liste en lecture seule

→ `QListWidget(QWidget* parent=0);`

- Slots :
  - `void clear();`
- Signaux :
  - `void currentRowChanged(int);`
  - `void currentTextChanged(QString);`

Les widgets Qt (2/2)

Les listes

QComboBox : liste déroulante (éditable)

→ `QComboBox(QWidget* parent=0);`

- Slots :
  - `void clear();`
- Signaux :
  - `void currentIndexChanged(int);`
  - `void currentIndexChanged(QString);`

Les widgets Qt

Saisir du texte

QLineEdit : pour du texte et des valeurs

→ `QLineEdit(QString contents,QWidget* parent);`

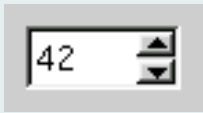
- Slots :
  - `void setText(QString);`
  - `void clear();`
- Signaux :
  - `void textChanged(QString);`
  - `void returnPressed();`
- Pour des valeurs entières ou flottantes :
  - `void setValidator(QValidator* );`
  - `QIntValidator`, `QDoubleValidator`

Les widgets Qt

Spinner et slider

QSpinBox : spinner

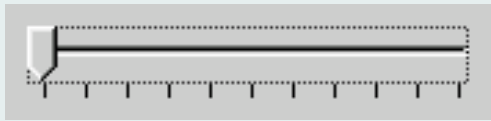
→ `QSpinBox(QWidget* parent=0);`



- Slots :
  - `void setValue(int);`
- Signaux :
  - `void valueChanged(int);`

QSlider : ascenseurs

→ `QSlider(Qt::Orientation ori={Qt::Vertical|Qt::Horizontal}, QWidget* parent=0 );`



- Signaux/slots :
  - cf `QSpinBox`

Les widgets Qt (1/2)

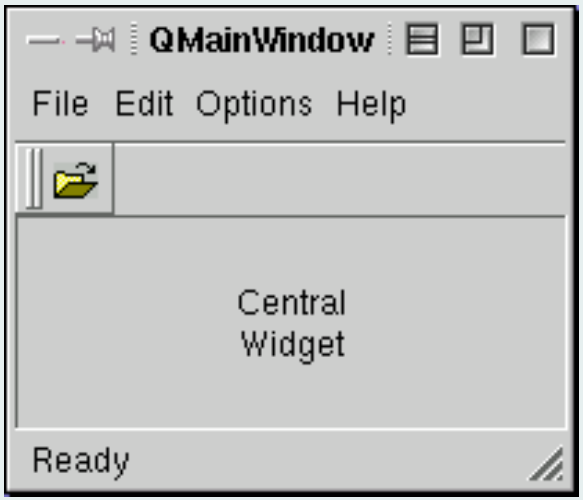
Fenêtre principale

`QMainWindow` : la fenêtre principale

→ `QMainWindow(QWidget* parent);`

Contient :

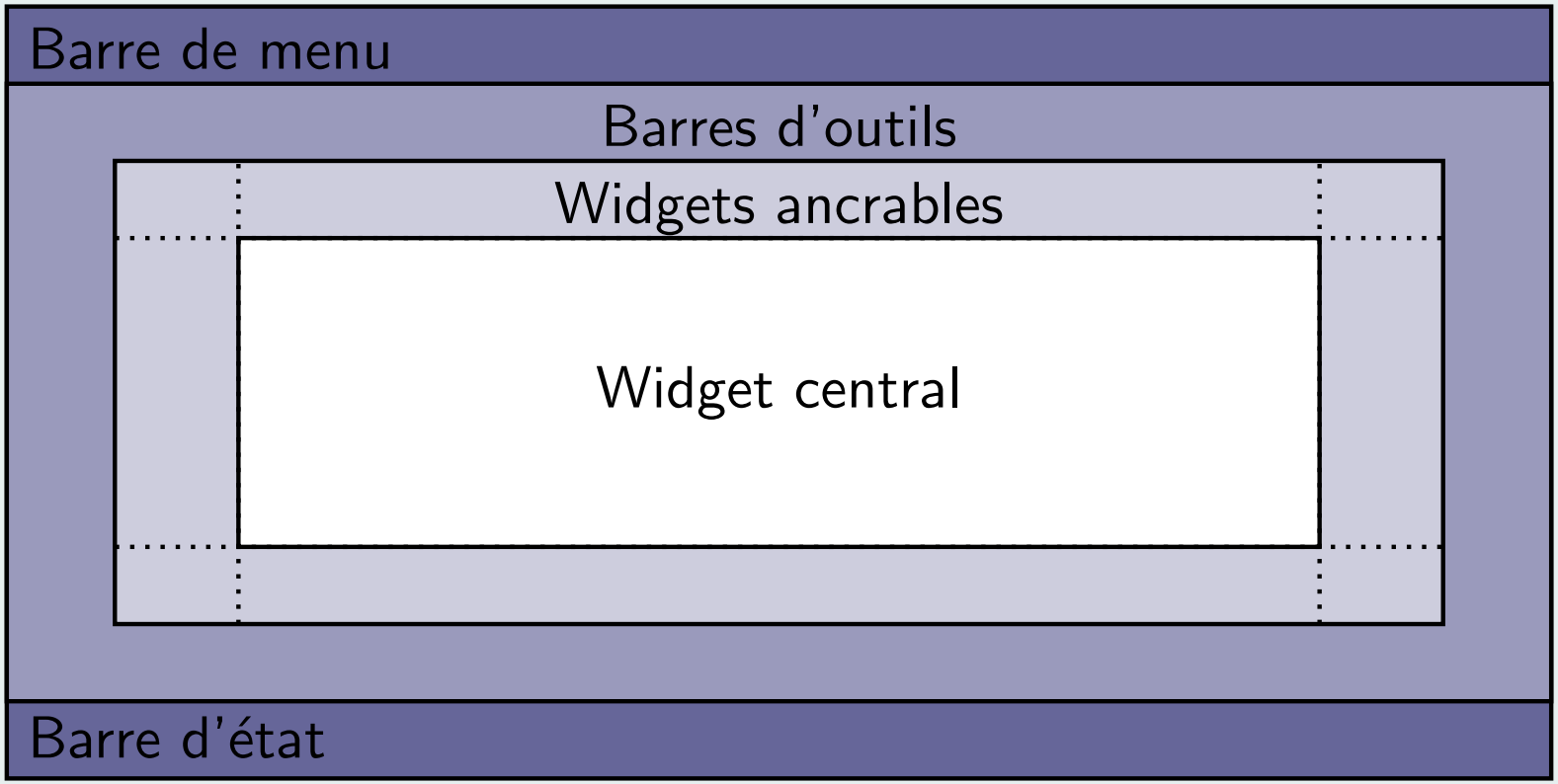
- Une barre de menu : `QMenuBar` ;
- Une barre d'état : `QStatusBar` ;
- Des barres d'outils : `QToolBar` ;
- Un widget central : `QWidget` ;
- Des widgets ancrables : `QDockWidget`.



Les widgets Qt (2/2)

Fenêtre principale

Mise en page d'une fenêtre principale



Gestion de la disposition (1/2)

Liens entre les classes `QWidget` et `QLayout`

- `layout()` retourne pour chaque *widget* la disposition courante ;
- `setLayout(QLayout*)` permet d'imposer une disposition. S'applique aux enfants du *widget* ;
- Attention ! Il faut détruire la disposition courante avant d'en appliquer une nouvelle ;
- Les enfants du *widget* sont à ajouter au *layout* en utilisant la méthode `addWidget(QWidget*)` de la classe `QLayout`.

Gestion de la disposition (2/2)

Principaux gestionnaires de disposition

Héritent de la classe `QLayout` :

- QHBoxLayout : rangée horizontale ;
- QVBoxLayout : colonne verticale ;
- QGridLayout : arrangement selon une grille ;
- QFormLayout : arrangement sur deux colonnes.

QMake

Caractéristiques

- Générateur de projets multi-plateformes :
  - génére des *makefiles* sous Unix ;
  - génére des projets *Visual studio* sous Windows ;
  - ...
- Se base sur un profil (*qmakespec*) ;
- Utilise un fichier de description textuelle d'un projet ;
- Est une moulinette à utiliser (en principe) une seule fois en début de projet.

QMake

Paramètres

Balises

SOURCES	fichiers cpp à compiler ;
HEADERS	fichier d'en tête ;
TEMPLATE	type de sortie à générer :
TARGET	nom de la sortie ;
CONFIG	extensions à utiliser ;
LIBS	bibliothèques additionnelles à charger ;
INCLUDEPATH	chemin vers répertoires contenant des en-têtes associés à des bibliothèques.

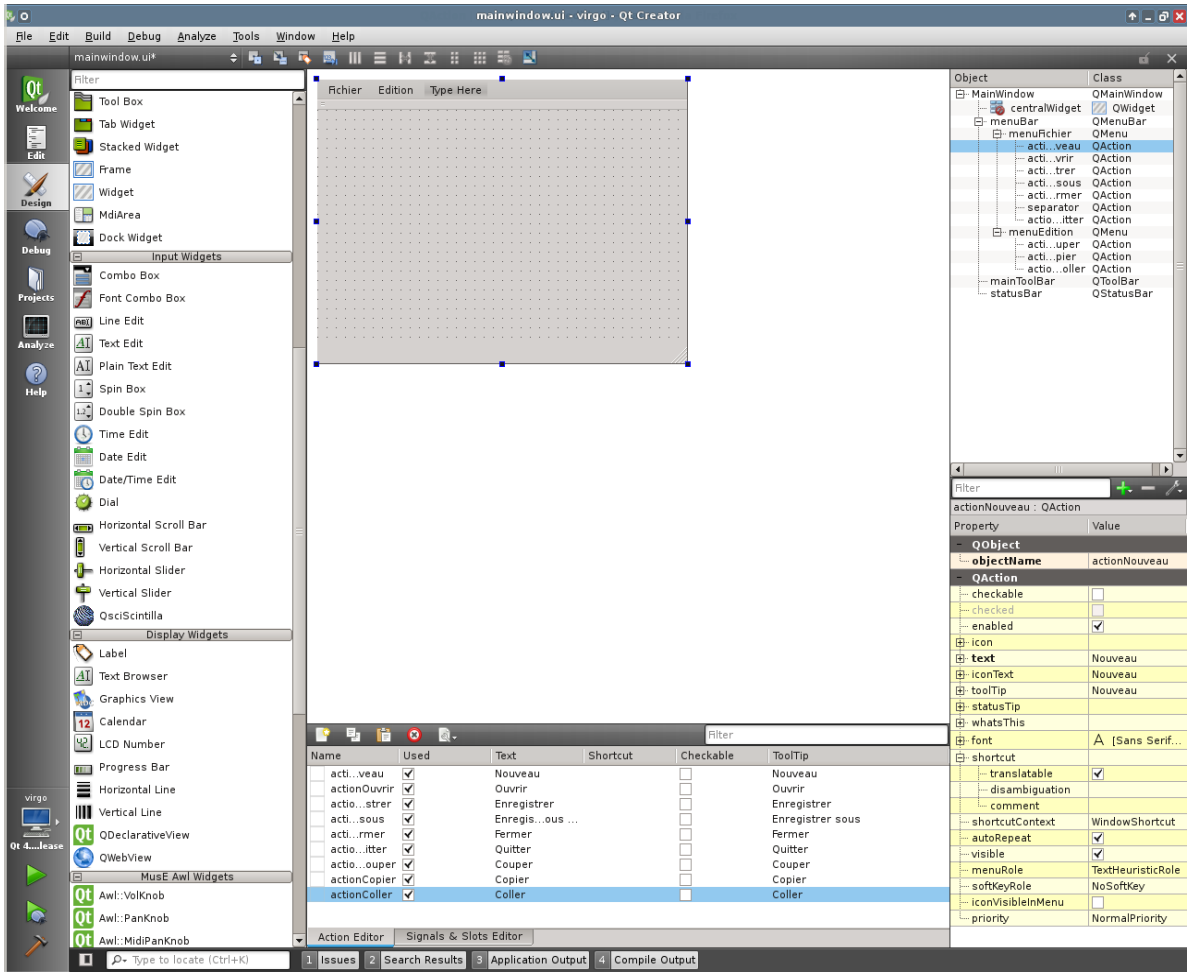
Qt creator

Un IDE pour Qt

- IDE = *Integrated Development Environment* ;
- Gère les projets et les chaînes de compilations avec `QMake` ;
- Edition du code avec coloration syntaxique, auto-complétion et auto-correction ;
- Permet le débogage en ligne du programme (front-end graphique à GDB) ;
- Facilite la conception des interfaces graphiques en mode WYSIWYG.

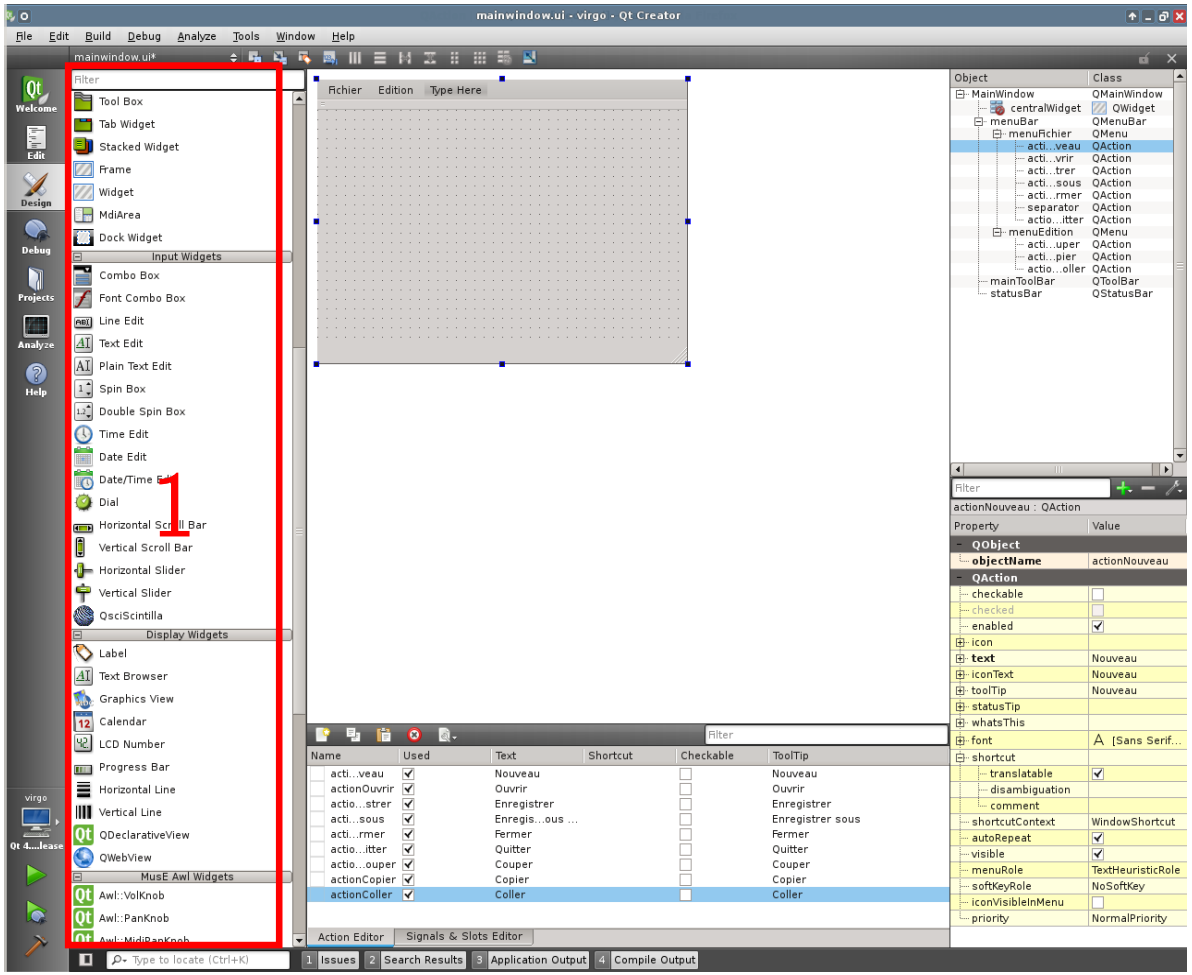
Qt Creator

Edition d'interface graphique



Qt Creator

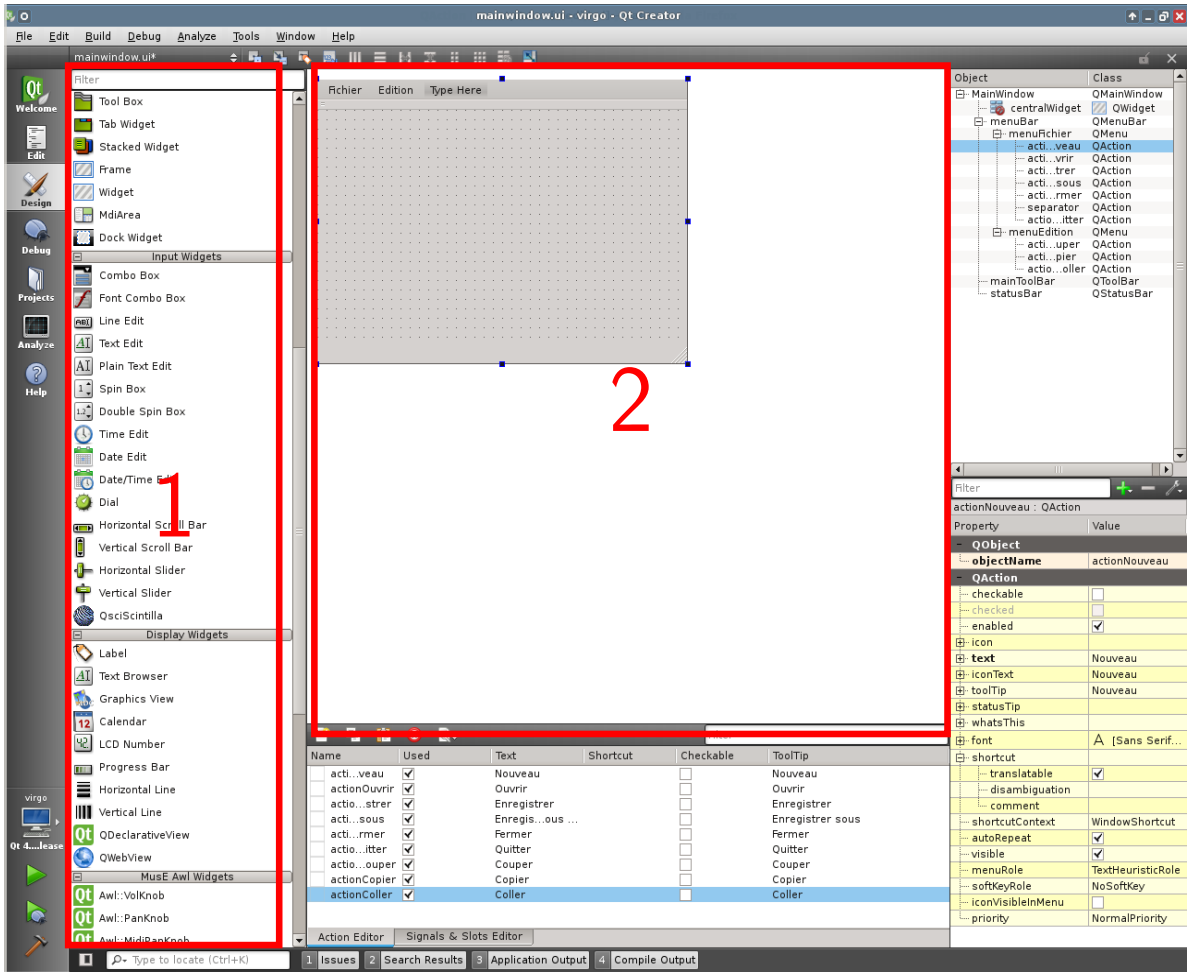
Edition d'interface graphique



1. Widgets et layouts disponibles par glisser/déplacer ;

Qt Creator

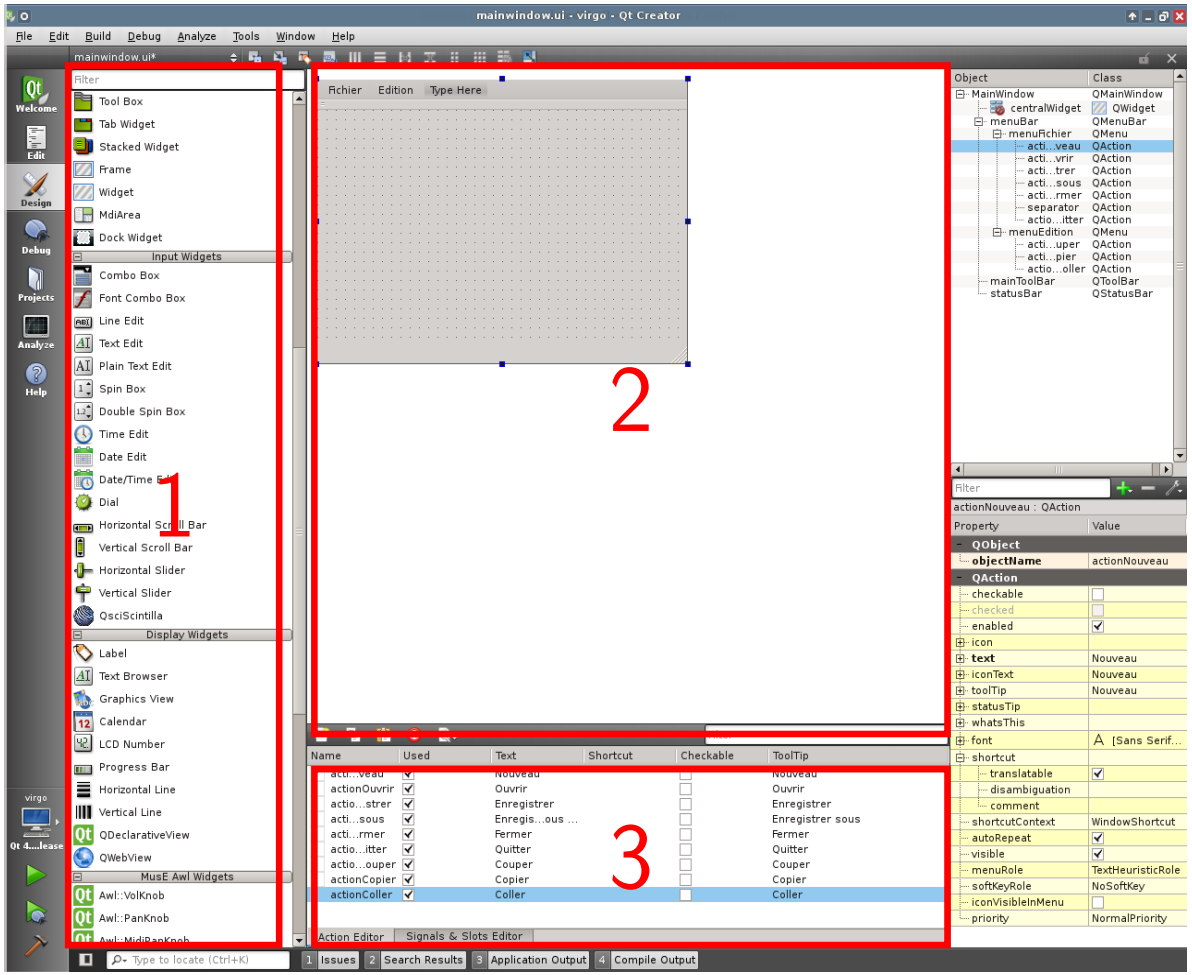
Edition d'interface graphique



1. Widgets et layouts disponibles par glisser/déplacer ;
2. Zone d'édition proprement dite ;

Qt Creator

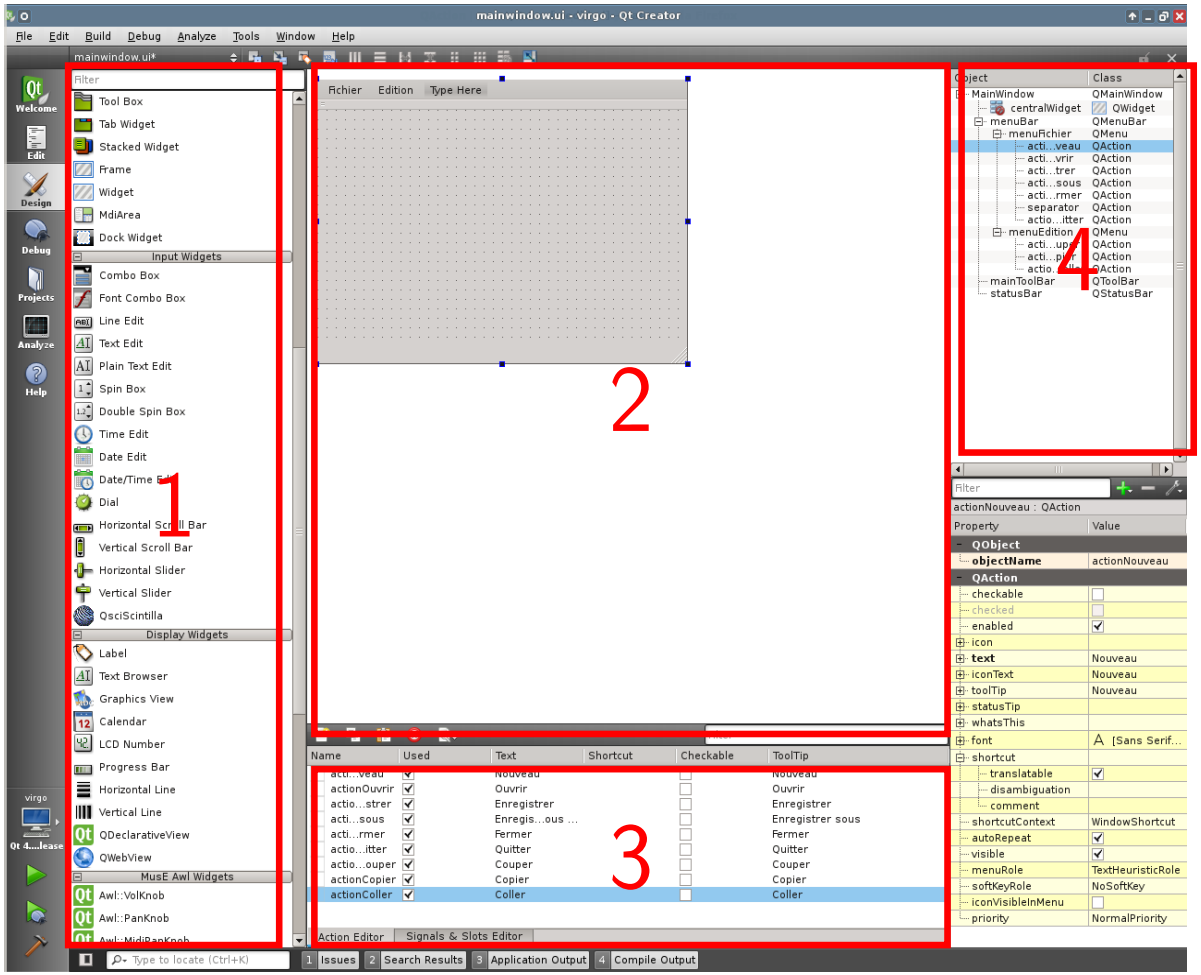
Edition d'interface graphique



1. Widgets et layouts disponibles par glisser/déplacer ;
2. Zone d'édition proprement dite ;
3. Zone gérant les actions et les connexions signal/slot ;

Qt Creator

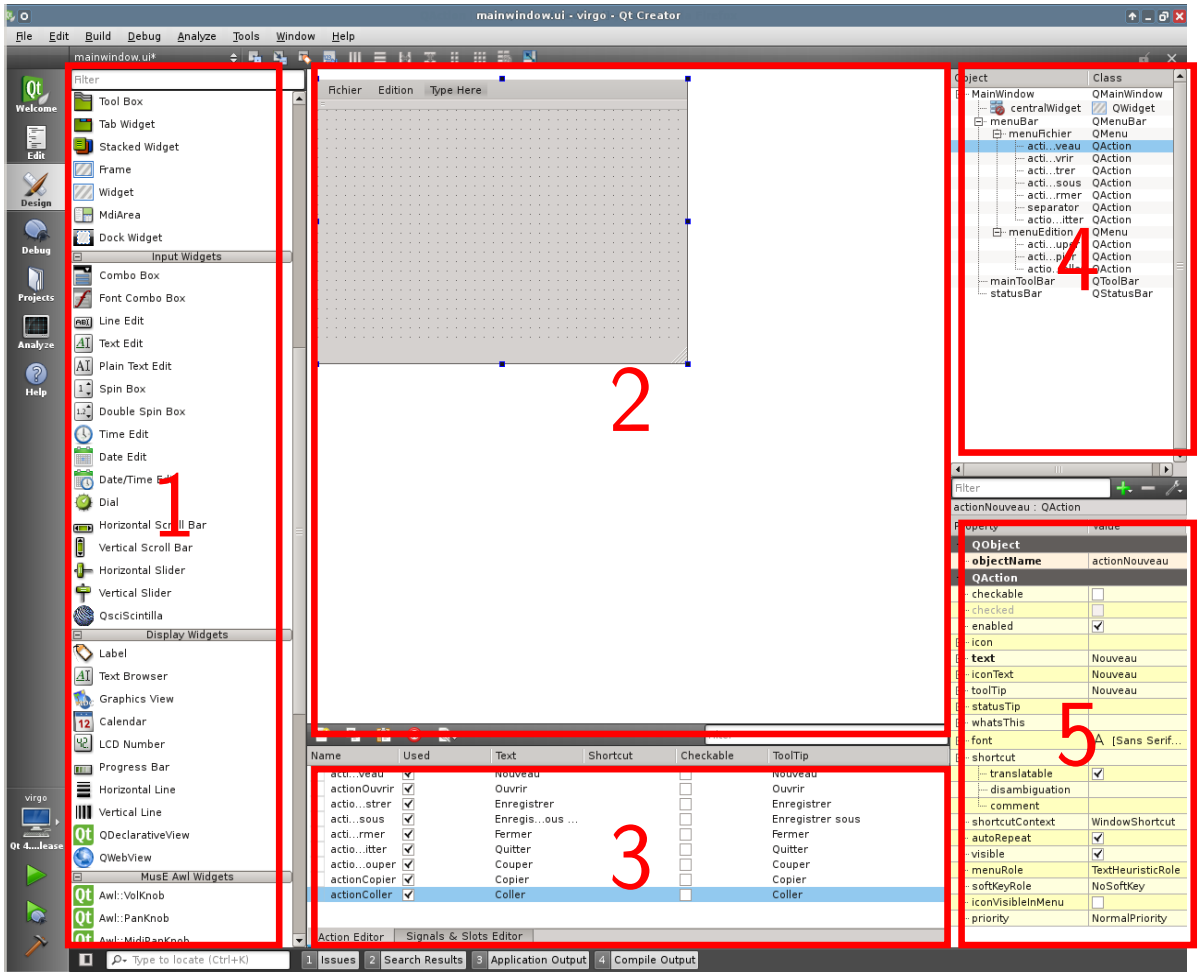
Edition d'interface graphique



1. Widgets et layouts disponibles par glisser/déplacer ;
2. Zone d'édition proprement dite ;
3. Zone gérant les actions et les connexions signal/slot ;
4. Hiérarchie des widgets ;

Qt Creator

Edition d'interface graphique



1. Widgets et layouts disponibles par glisser/déplacer ;
2. Zone d'édition proprement dite ;
3. Zone gérant les actions et les connexions signal/slot ;
4. Hiérarchie des widgets ;
5. Propriétés du widget sélectionné dans la zone d'édition.

Sources

- `http://qt-project.org` :
  - Site de la communauté, avec code sous licence LGPL, documentation des APIs et tutoriaux ;
- `http://qt.digia.com` :
  - Site commercial officiel de Qt par Digia.
- `http://qt.developpez.com` :
  - Site de la communauté francophone avec tutoriaux et documentation en français.