MANUEL DE QT

**ASCENSEUR A POISSONS**



**SOMMAIRE**

INTRODUCTION page 0

LES BASES DE L’INTERFACE page 0

LES FICHIERS RESSOURCES page 0

ORGANISATION DES LAYOUTS page 0

**INTRODUCTION**

Pour pouvoir développer l’interface de contrôle de l’ascenseur à poisson, on a besoin d’une bibliothèque fournissant tous les outils nécessaires au développement de GUIs (**G**raphical **U**ser **I**nterface). Le cahier des charges spécifie deux environnements au choix pour pouvoir concevoir l’interface logicielle : le framework .NET de Microsoft à partir de Visual Studio ou la bibliothèque multiplateforme Qt.

Nous avons décidé de choisir Qt car nous l’avons déjà utilisé auparavant en cours donc il était plus facile de l’appréhender et de commencer le développement plus rapidement.

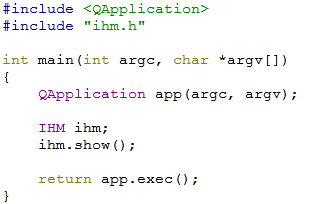
L’environnement de développement fournit avec Qt se nomme Qt Creator, il dispose de tous les outils nécessaires à la conception d’interfaces graphiques.



**Environnement de développement**

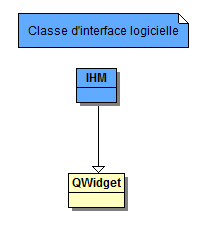
**LES BASES DE L’INTERFACE**

Le point d’entrée de tous les programmes développés en C++ est le fichier **main.cpp**. Nous allons devoir inclure la classe d’interface logicielle **ihm.cpp,** elle est très importante car elle créée tous les objets des classes d’interface intermédiaires représentants les différents onglets de l’application ainsi que l’objet **Seance**.



***Main.cpp***

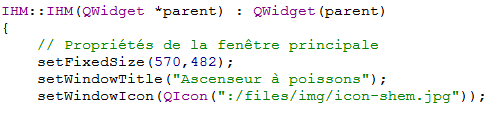
La classe **ihm.cpp** hérite de la classe **QWidget** qui est propre à Qt ainsi notre interface est considérée comme un widget donc grâce à l’héritage elle possède toutes les propriétés et les méthodes d’un widget (afin de changer les dimensions, la police, la couleur…etc).



*La classe IHM hérite de la classe QWidget*

Pour pouvoir faire apparaître l’IHM, on utilise la méthode **show()** de la classe QWidget qui permet d’activer le visibilité du widget, équivalent à la méthode setVisible(true).

Les propriétés de la fenêtre doivent être définies dans le constructeur de la classe IHM :



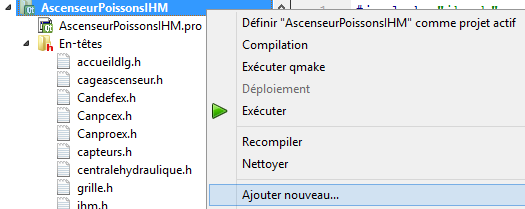
Trois propriétés sont définies :

* La taille de la fenêtre (correspondant à un écran de 6.5 pouces environ)
* Le titre de la fenêtre
* L’icône apparaissant dans la barre des tâches

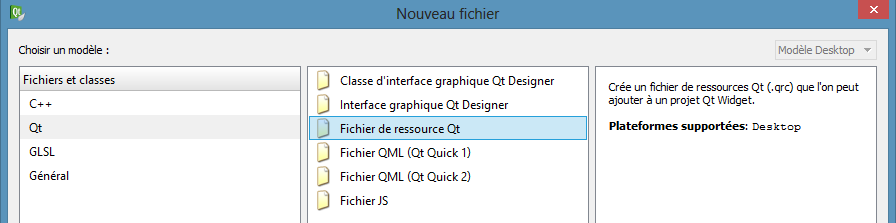
**LES FICHIERS RESSOURCES**

Les ressources sous Qt sont un moyen simple et rapide afin d’importer des assets (fichiers son, images, videos…etc) au sein du projet et de pouvoir les utiliser quand on le souhaite.

Pour cela, il faut ajouter un nouveau fichier au projet :



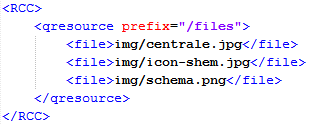
De type « *Fichier ressource Qt* » :



Une fois le fichier crée, il est accessible depuis l’arborescence du projet dans le sous-menu « *ressources »*. Dans notre cas, les images de l’application sont contenues dans un dossier nommé « img » à la racine du projet Qt.

Ensuite il faut définir un préfixe afin d’organiser les ressources comme on le souhaite, le nom est arbitraire, par exemple : « */files* ». Ensuite il suffit d’ajouter les assets à partir du bouton « *ajouter des fichiers* »

La structure d’un fichier ressource est assez simple, il s’agit en fait d’un fichier au format XML listant les fichiers importés avec le préfixe correspondant :



*Fichiers de ressources .qrc*

Pour pouvoir utiliser une image ajoutée précédemment, le moyen le plus simple et d’ajouter un QLabel et de définir sa propriété Pixmap à l’aide de la classe QPixmap :



Le chemin vers le fichier désiré doit être mis en paramètre du constructeur de la classe QPixmap, le chemin se compose ainsi :

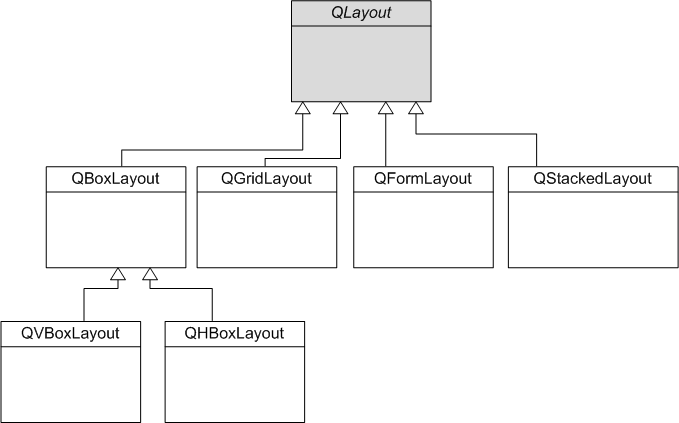


**: / [PREFIX]** **/** **[DOSSIER]** **/** **[FICHIER]**

Une autre solution est d’utiliser la classe QIcon qui a un fonctionnement similaire à QPixmap mais pour la gestion des icônes.

**ORGANISATION DES LAYOUTS**

Pour positionner intelligemment les widgets, il faut utiliser des classes de Qt gérant les layouts. Il existe par exemple des classes gérant le positionnement horizontal et vertical des widgets ou encore le positionnement sous forme de grille.



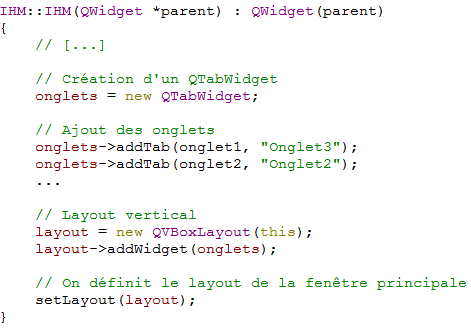
**QLayout** est une classe abstraite regroupant les propriétés et méthodes communes des classes représentants les différents types de Layout. C’est un cas concret de **polymorphisme**.

Pour décrire et expliquer l’organisation des layouts au sein de l’application, nous allons voir deux cas présents dans le projet :

* **1er CAS : Organisation du layout de la fenêtre principal**

La fenêtre principale est composée d’un seul widget : un **QTabWidget**

Pour positionner correctement les onglets afin qu’ils prennent toute la largeur de la fenêtre, on doit utiliser un layout vertical à partir de la classe **QVBoxLayout**.



L’organisation de la fenêtre principale reste assez simple. La première chose à faire est de créer un ensemble d’onglets grâce à la classe **QTabWidget**. Ensuite on ajoute les différents onglets via la méthode **addTab()** :

int QTabWidget::addTab ( QWidget \* page, const QString & label );

Cette méthode attend en premier argument un pointeur sur un élément de type QWidget, dans notre cas ça sera un objet d’une classe d’interface intermédiaire et en deuxième argument, la méthode attend une chaîne de caractère en référence constante.

Maintenant il faut ajouter un nouveau layout de type QVBoxLayout (qui hérite de la classe **QBoxLayout**) puis y ajouter nos onglets via la méthode **addWidget()**:

void QBoxLayout::addWidget (

QWidget \* widget,

int stretch = 0,

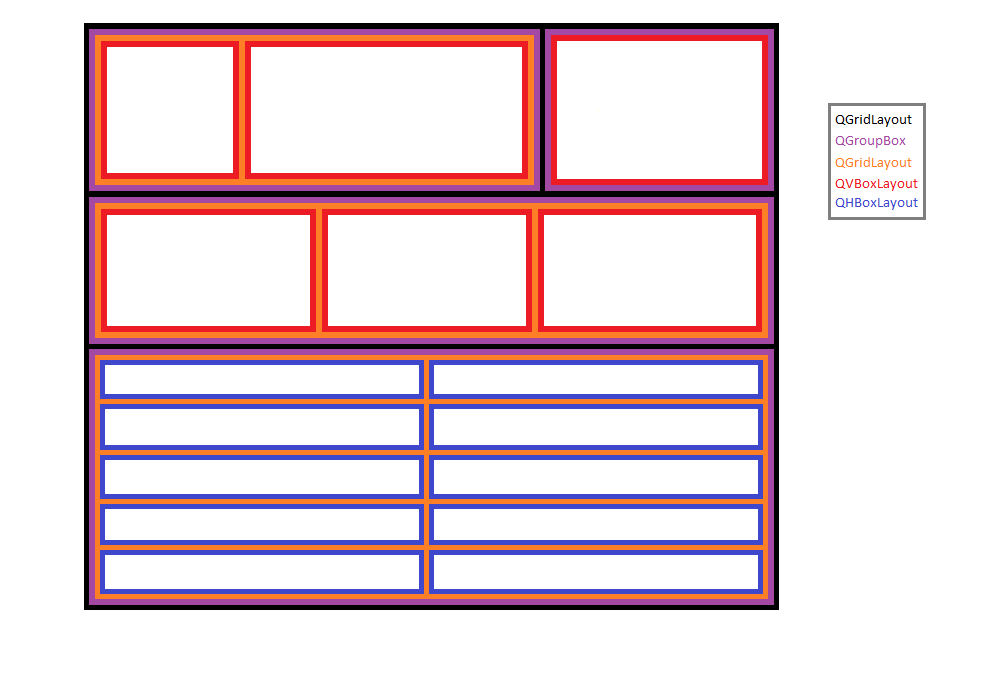
Qt::Alignment alignment = 0

);

Pas besoin de définir un quelconque étirement ou alignement lors de l’ajout des onglets au layout, donc on utilise uniquement le premier argument pour ajouter notre widget à la fenêtre.

La dernière étape consiste à définir le layout de la fenêtre principal. Pour ce faire, on utilise la méthode **setLayout()** de la classe QWidget en mettant en premier argument le layout défini précédemment.

* **2er CAS : Organisation du layout de l’onglet « statuts »**

Voici un exemple un peu plus complexe d’association de plusieurs layouts. Pour l’onglet statuts, on a besoin de 3 types de layout :

* QGridLayout
* QVBoxLayout