Notes :

A. But du TP

Il s'agit d'apprendre à

☞ *afficher une image en utilisant Qt*

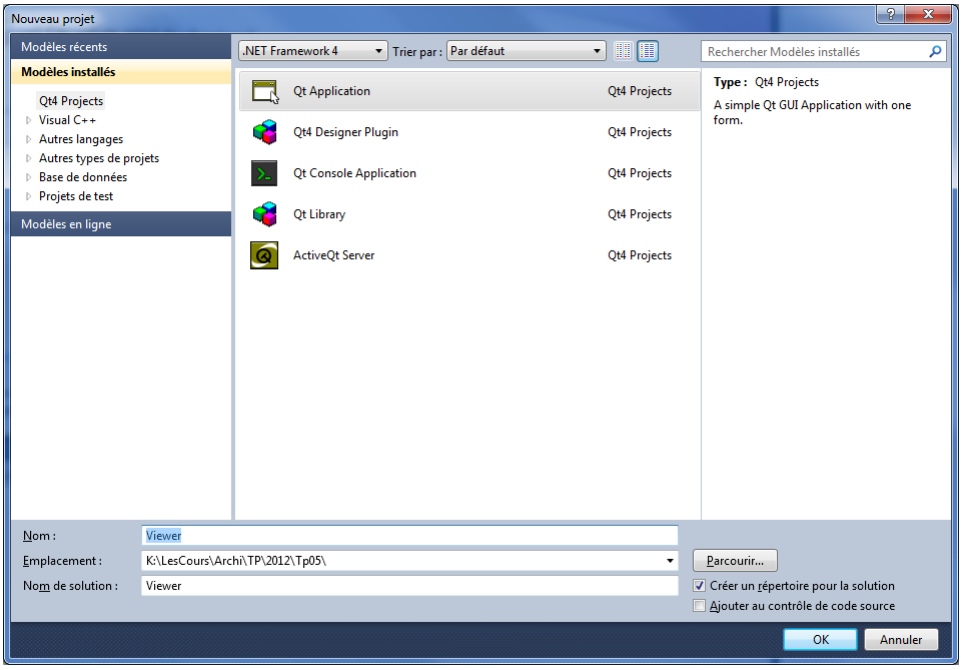
☞ *faire des traitements simples sur celle-ci*

Nota : On est prié de lire tout le sujet avant de commencer

« Il est véritable que qui ôte à l'esprit la réflexion lui ôte toute sa force. »
BOSSUET

B. Projet à développer

On créera un projet Qt avec Visual 2010 C++ que l'on nommera : Viewer

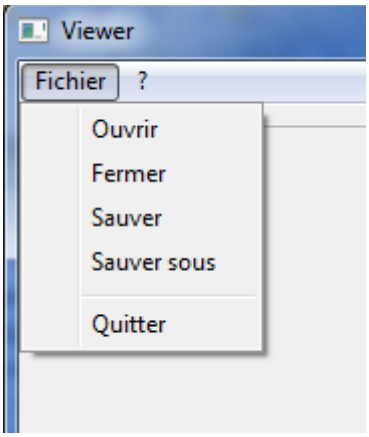


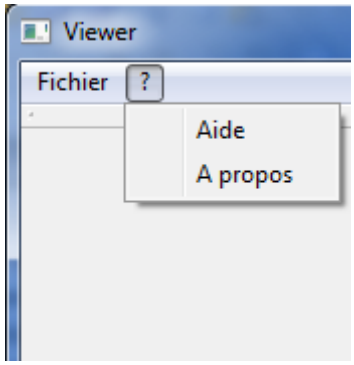
La classe Viewer dérivera de QMainWindow

Travail à faire

1. **Préparation du fichier Viewer.ui**

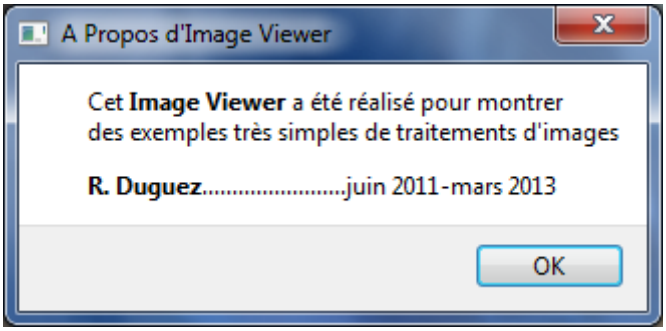
A l'aide de Qt Designer, on va ajouter des menus comme ceci :





2. **Le menu A propos**

*Le menu **A propos** utilisera un QMessageBox et permettra d'afficher:*



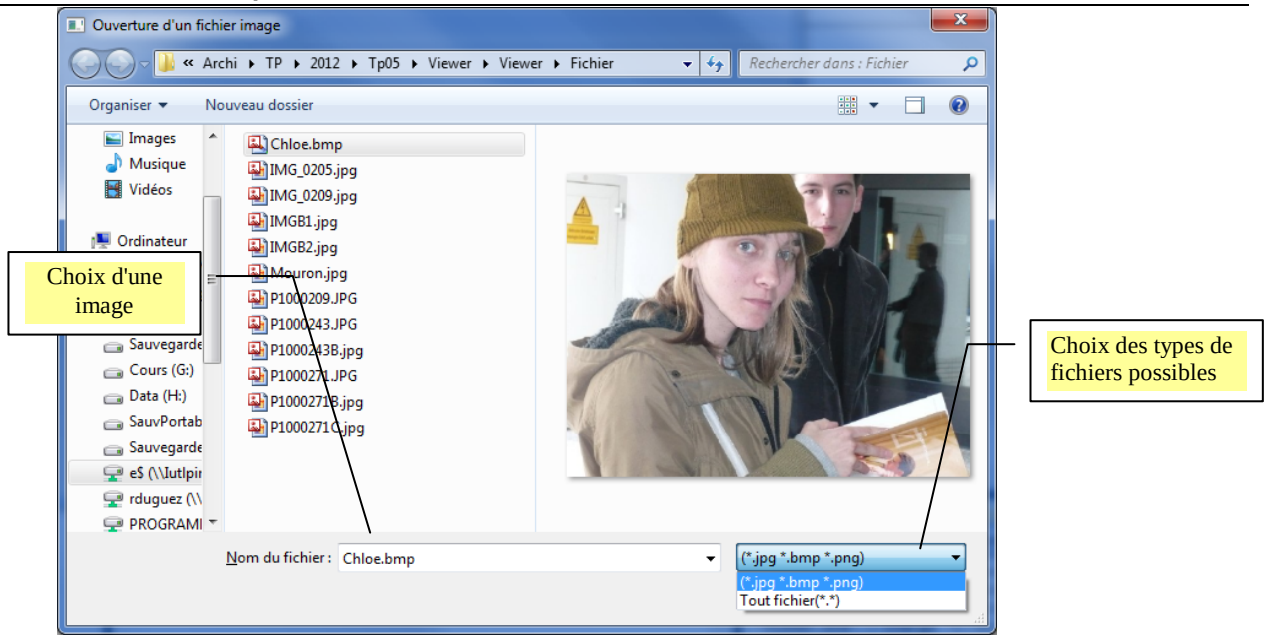
3. **Le menu Fichier**

☞ **Nota :** les images seront gérées à partir d'une classe QXImage dérivée de QImage que l'on enrichira au fur et a mesure du développement

- **Ouvrir une image**

Les fichiers Images seront tous stockés dans un même répertoire 'Fichier'; on gardera l'adresse de ce répertoire par la variable MonRep . Celle-ci sera initialisée dans le constructeur de Viewer

L'ouverture et la sauvegarde d'un fichier image se fera par l'utilisation d'un QFileDIalog, comme ceci



On créera un QLabel dans Viewer.ui : on le nommera imageLabel.

.

Les SLOTS d'ouverture et de sauvegarde (Sauver et Sauver sous) permettront d'envoyer un nom de fichier correct à la méthode Sauver de la classe Viewer qui fera le travail

L'image lue sera stockée dans directement dans une variable Image de type QImage.*

On se servira d'une variable pixmap de type QPixmap pour afficher celle-ci.*

La méthode paintEvevent de Viewer permettra d'afficher l'image comme ceci:

```
void Viewer::paintEvent(QPaintEvent * evt)
{
   if (Image!= NULL)
   {
      *pixmap = QPixmap::fromImage(*Image) ;
      int h = pixmap->height() ;
      int w = pixmap->width() ;
      Dim.setX(w);
      Dim.setY(h) ;
      ui.imageLabel->setGeometry(0,0,Dim.x(),Dim.y()) ;
      ui.imageLabel->setPixmap(*pixmap) ;
   }

}
```

Permet de récupérer la taille de l'image dans un QPoint Dim

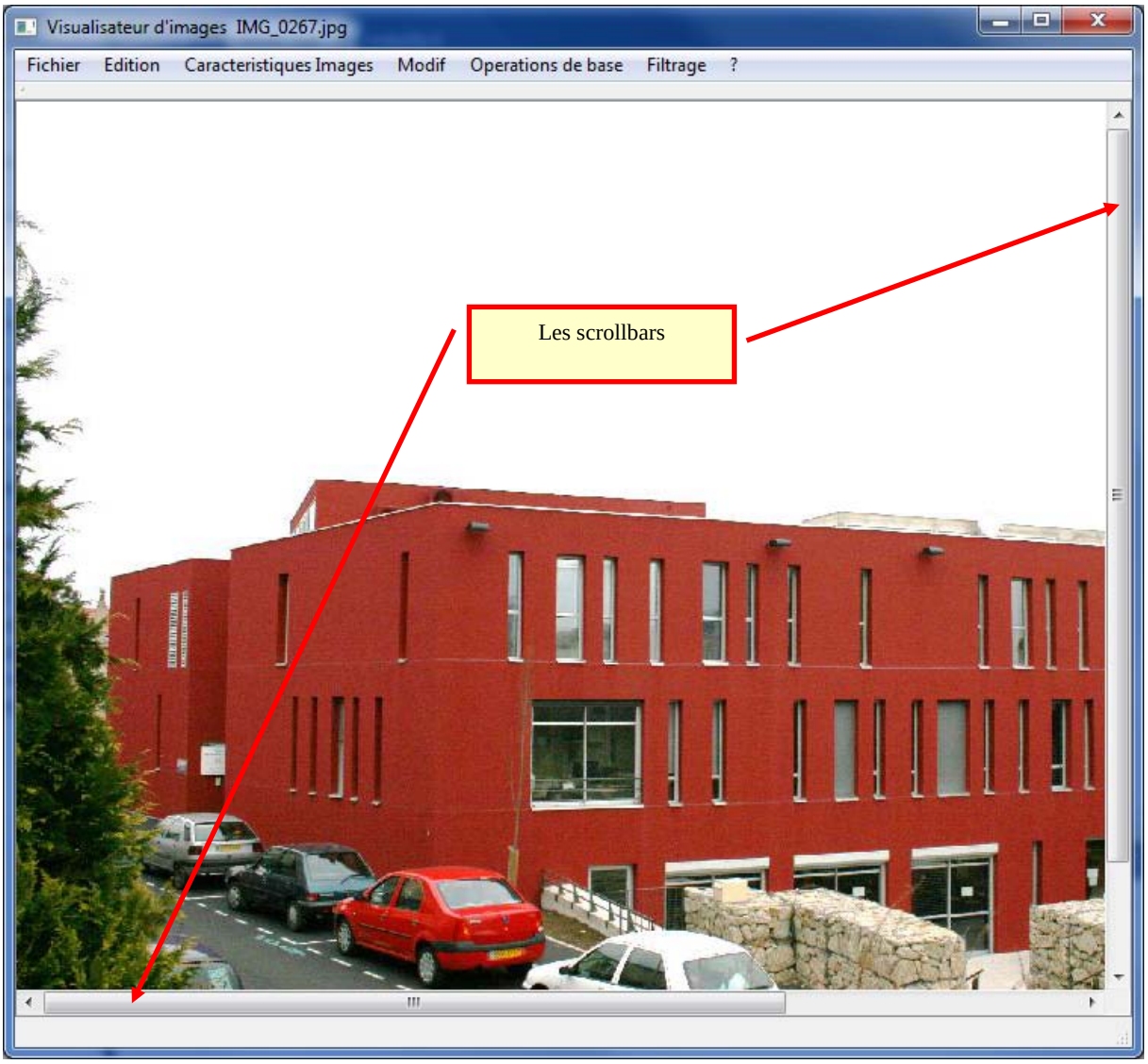
On met imageLabel à la dimension de l'image

L'image est affichée dans le QLabel imageLabel

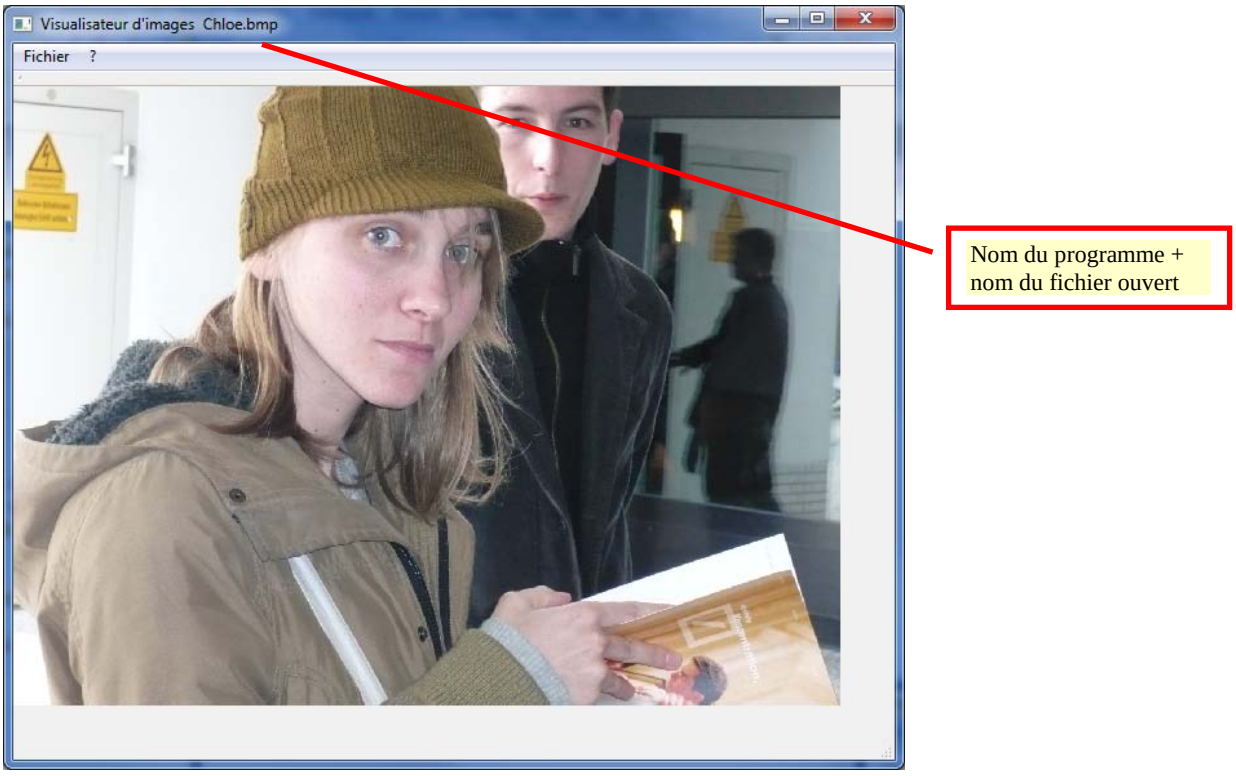
On mettra en place une ScrollArea, pour pouvoir afficher des images de grande taille.

On associera cela au centralWidget de cette façon :

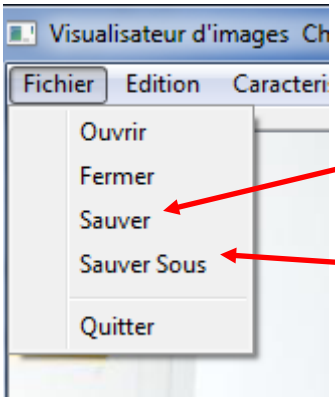
```
scrollArea = new QScrollArea;
scrollArea->setBackgroundRole(QPalette::Dark);
scrollArea->setWidget(ui.imageLabel);
setCentralWidget(scrollArea);
```



On veillera à afficher le nom du programme et le nom du fichier image ouvert dans WindowTitle , on prévoira d'ajouter un Astérix si le fichier a eu une modification

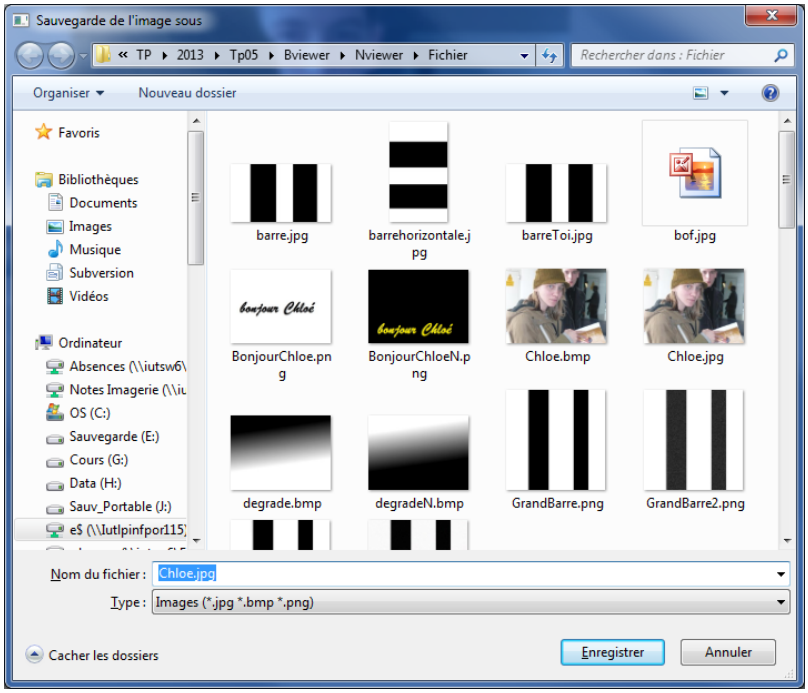


- **Mise en place de la sauvegarde**



L'appel de Sauver entrainera la sauvegarde du fichier en cours en l'état

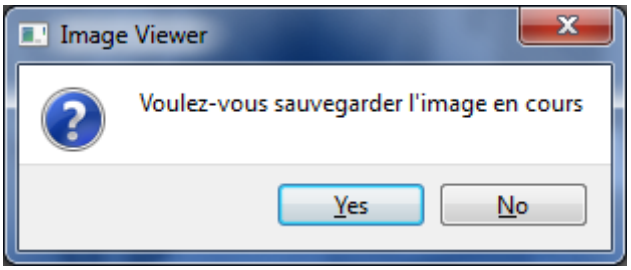
L'appel de Sauver Sous entrainera la l'ouverture d'une fenêtre permettant de choisir le nom sous le lequel on veut le sauver



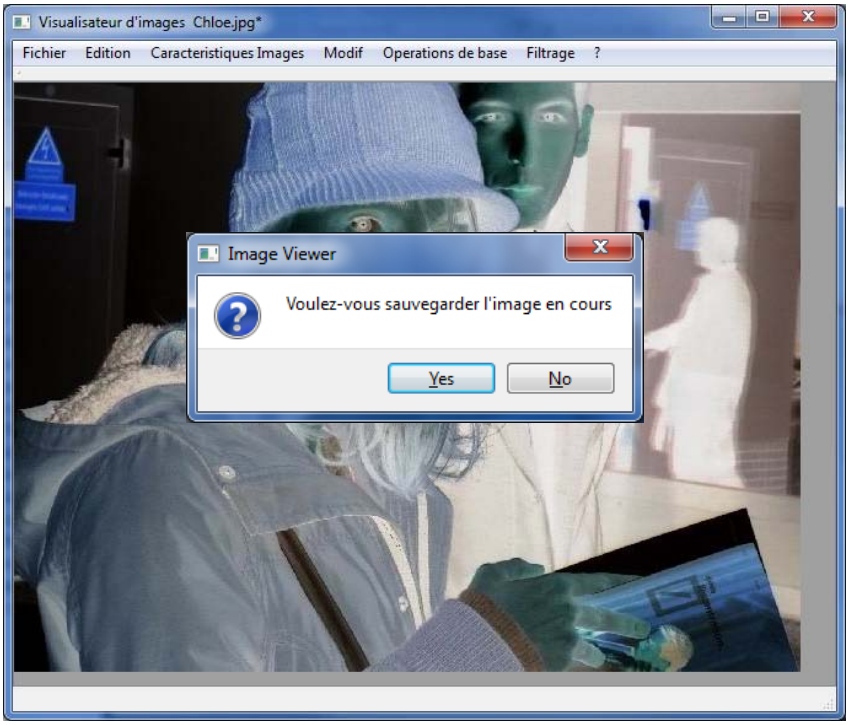
Dans tous les cas on veillera à supprimer l'étoile indiquant une modification

- **L'action Quitter**

Elle doit permette de quitter l'application, en demandant la sauvegarde de l'image en cours si elle a été modifiée



L'appui sur la croix de fermeture de l'application doit générer la même action



On pourra pour cela créer une méthode protected de viewer :

```
void closeEvent(QCloseEvent *event);
```

qui est toujours appelé en cas de fermeture

4. Le menu Edition

On va donc créer un menu Edition , que l'on placera en Fichier et ?.

- **Explication du travail à faire:**

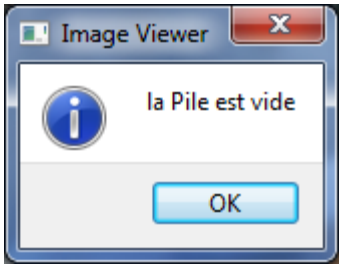
La gestion du Undo et du Refaire se fera à l'aide de Pile Qt

```
// ***************************************
//       gestion du Undo
// ***************************************
QStack <QXImage *> Pile ;          // pile des images Annulées
QStack<QString>  PileNom;          // pile des noms d'images

// ***************************************
//       gestion du Refaire
// ***************************************
QStack <QXImage *> PileR ;         // pile des images de Refaire
QStack<QString>  PileNomR;         // pile des noms d'images
```

	Pile de Annuler	*Actuel*	*Pile de Refaire*
Au départ		*Chloe.jpg*	
On charge une autre image	*Chloe.jpg*	*Mouron.jpg*	
On charge une autre image	*Chloe.jpg* *Mouron.jpg*	*Lettre.bmp*	
On demande annulation	*Chloe.jpg*	*Mouron.jpg*	*Lettre.bmp*
On charge une autre image	*Chloe.jpg* *Mouron.jpg*	*Barre.jpg*	*Lettre.bmp*
On demande Refaire	*Chloe.jpg* *Mouron.jpg* *Barre.jpg*	*Lettre.bmp*	

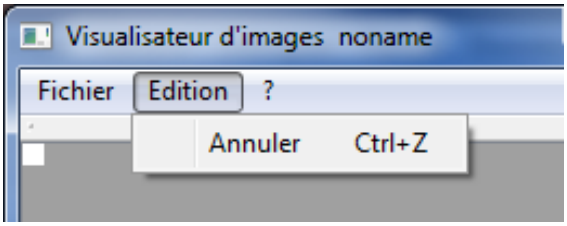
L'appel des fonctions Undo et Refaire entrainera l'envoi d'un message si les piles sont vides



- **Le Undo**

On va être amené à faire des modifications sur des images, on veut donc pouvoir revenir en arrière.

Il comprendra une action Annuler, qui sera associé au shortcut Ctrl+Z



*Pour cela on pourra utiliser des piles de QT comme ceci:**

```
// ***************************************
//       gestion du Undo
// ***************************************
QStack <QXImage *> Pile ;          // pile des images
QStack<QString>  PileNom;          // pile des noms d'images
```

On y empilera les images et les noms des images. On créera pour cela deux méthodes:

```
void Empiler ( QImage * img) ;
void Depiler () ;
```

on utilisera judicieusement celles-ci dans le slot open et la méthode sauver

Ce undo devra fonctionner pour toutes les modifications images que nous feront par la suite.

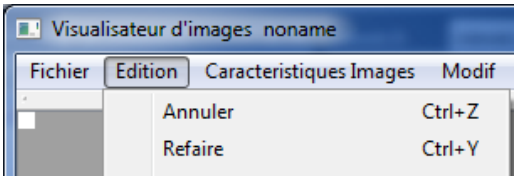
On créera une méthode privée Travail permettant de gérer le Undo. Elle pourra avoir le prototype suivant :

```
void Travail(QXImage *tmp, QString NomImage) ;
```

- **Le Refaire**

On veut pouvoir mettre en place une annulation des retours arrières

On va donc ajouter une action Refaire au menu Edition qui sera associer au shortcut Ctrl +Y

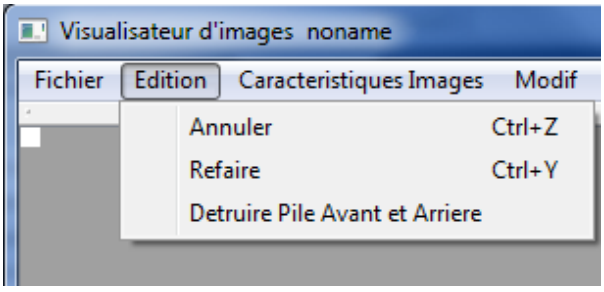


Pour cela on pourra utiliser des piles de QT comme ceci:

```
// ***************************************
//      gestion du Refaire
// ***************************************
QStack <QXImage *> PileR ;           // pile des images
QStack<QString>  PileNomR;           // pile des noms d'images
```

- **Détruire**

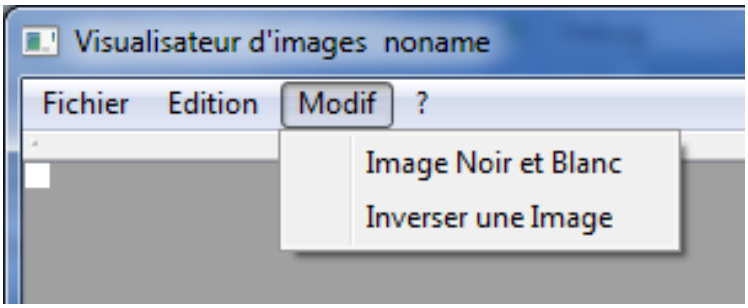
Les piles peuvent être très grosses au bout de quelques temps, on demandera de pouvoir les détruire et pour cela on ajoutera une 3° action au menu Edition:



5. Le menu Modif

- **Mise en place du menu Modif**

On va créer un menu Modif que l'on va placer entre Edition et ? . Il comprendra les actions Image Noir et Blanc, Inverser une image comme ceci:



Gestion des modifications d'images

Pour pouvoir gérer les modifications d'images facilement, on va créer , si nécessaire, des méthodes dans la classe QXImage dérivant de QImage. Ses constructeurs et son destructeur feront appel à ceux de QImage.

On enrichira au fur et à mesure cette classe avec des méthodes appropriées.

- **Mettre l'image en Noir et Blanc**

On créera une méthode publique de QXImage : `toGrayscale` permettant de transformer l'image courante en image Noir et Blanc

Elle aura le prototype suivant:

```
QXImage  toGrayscale     ( bool  keepAlpha = true);
```

Si l'image est vide, elle retourne un QXImage par défaut, sinon elle retourne une image dont chaque pixel est transformé en niveau de gris. Pour cela on récupérera chaque pixel de l'image dans une donnée de type QRgb que l'on transformera en niveau de gris par la fonction qGray , ceci nous permettra de créer un pixel dont les trois couleurs seront au même niveau de gris.

```
QRgb dstPixel = qRgba(grayPixel, grayPixel, grayPixel, (keepAlpha) ? qAlpha(srcPixel) : 255);
```

Ceux-ci permettront de créer l'image résultante.

L'appel à cette méthode à partir du menu concerné permettra de transformer l'image en niveau de gris

- **Mettre l'image en Négatif**

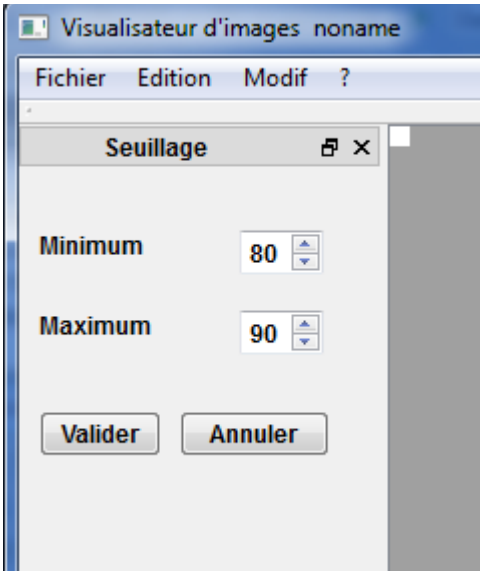
*On utilisera la méthode **`invertPixels ()` de QImage (donc visible dans QXImage)***

- **Mettre en place d'un seuillage**

*Le seuillage consiste à transformer une image en niveau de gris. À partir de cette image en niveau de gris, le seuillage d'image peut être utilisé pour créer une image comportant uniquement deux valeurs, noir ou blanc. Le **seuillage d'image** remplace un à un les pixels d'une image à l'aide d'une valeur **seuil** fixée (par exemple **123**). Ainsi, si un pixel à une valeur supérieure au seuil (par exemple **150**), il prendra la valeur **255** (blanc), et si sa valeur est inférieure (par exemple **100**), il prendra la valeur **0** (noir).*

Nous créerons un seuillage à deux niveaux, Min et Max . pour chaque pixel de l'image si la valeur du pixel appartient à [min,max], celui-ci prendra la valeur 0 (noir) et 255 dans les autres cas

Pour cela on va introduire un QDockWidget dans viewer.ui que l'on appellera DWSeuillage comme ceci::



On ajoutera une action Seuillage dans le menu Modif qui sera checkable. Au lancement du programme de DockWidget sera caché.

On écrira une méthode publique de QXImage permettant d'effectuer un seuillage sur l'image courante. Le prototype de cette méthode pourra être :

```
QXImage * Seuillage             ( int   Min, int Max   );
```

Ou

```
void  Seuillage                 ( int   Min, int Max   );
```

Le seuillage s'effectuera lorsque un des seuils sera modifié, l'image obtenu sera empilée (pour le Undo) lorsque l'on aura validé, une annulation permettra de revenir en arrière.

Exemple :

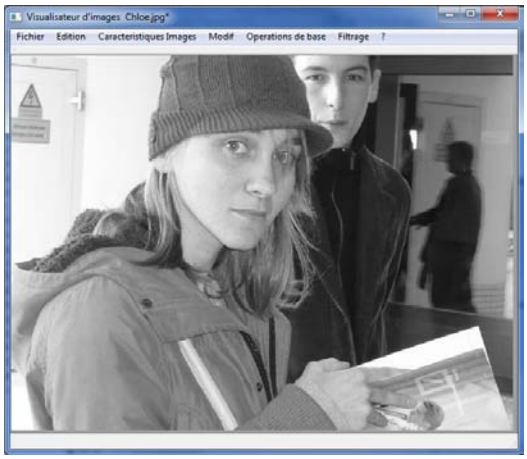
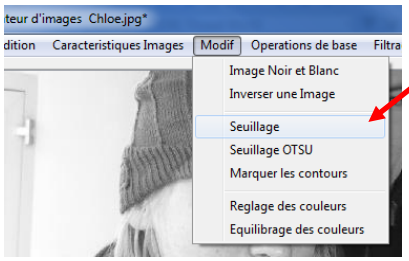
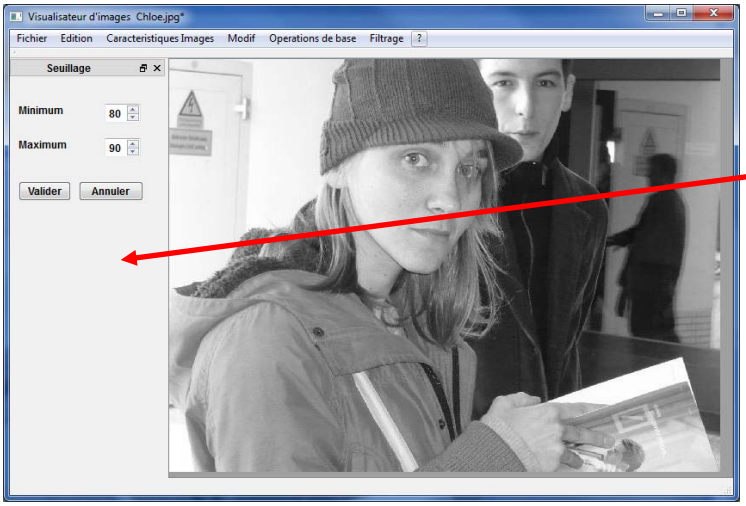


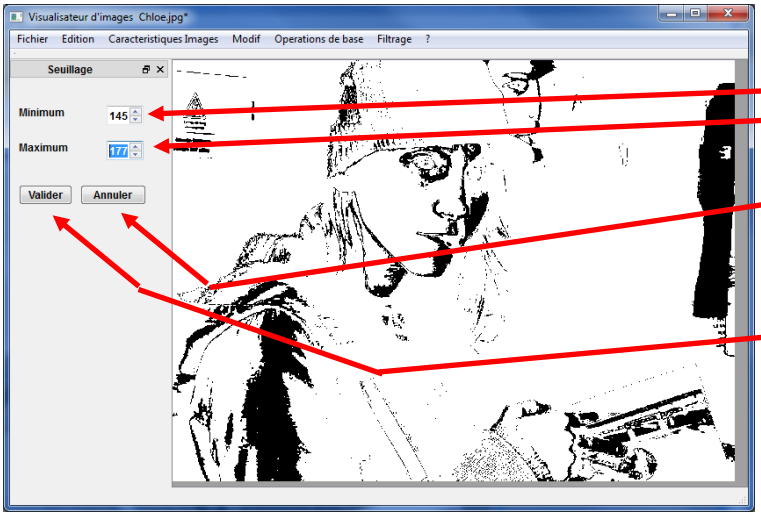
Image initiale

Appel de seuillage





Ouverture du DockWidget QWSeuillage

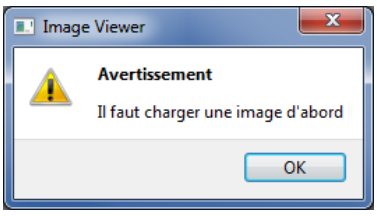


Le choix d'un minimum et d'un maximum permet de voir l'image seuillée

L'appui sur Annuler permet de revenir à l'image initiale

Le résultat sera définitif une fois valider

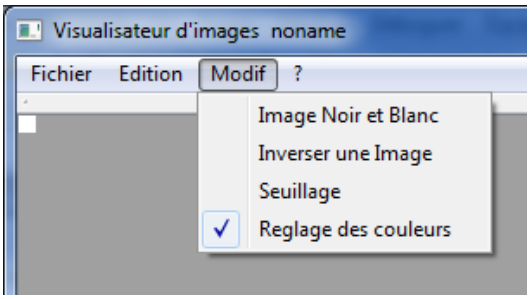
Le lancement sans image entrainera un message :



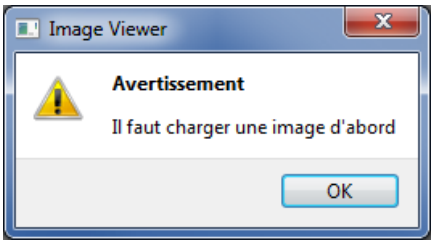
- **Mettre en place les réglages de couleurs**

On veut maintenant pouvoir modifier les niveaux de chaque couleur (R, G, B) et/ou la luminosité(augmentation ou diminution du niveau de tous les couleurs à la fois .

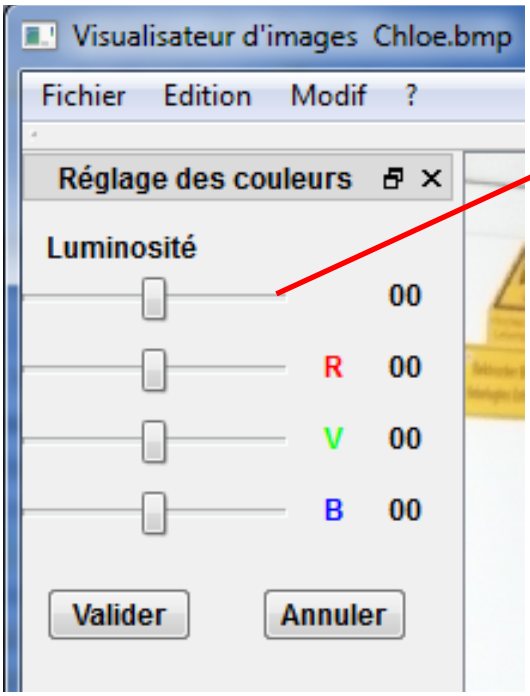
On ajoutera une action Réglage des couleurs dans le menu Modif. Elle sera checkable, mais on créera un dispositif qui ne permettra de checker qu'une seule des actions checkables possibles.



Le lancement sans image entrainera un message :



Pour cela on va introduire un QDockWidget dans viewer.ui que l'on appellera DwReglage comme ceci::



Les sliders sont réglés entre -100 et +100
Valeur par défaut 0

La valeur des éléments du slider correspondent à un pourcentage dont on veut diminuer ou augmenter chaque couleur, toutes les couleurs à la fois pour la luminosité.

Les modifications dans l'image courante seront effectuer par une méthode publique Reglage pouvant avoir ce prototype :

```
QXImage * Reglage        ( int ValR, int ValG, int ValB       );
```

ou

```
void      Reglage        ( int ValR, int ValG, int ValB       );
```

Les paramètres ValR, ValG et ValB corespondent à un pourcentage de diminution ou augmentation de la couleur concernées.

Exemple :

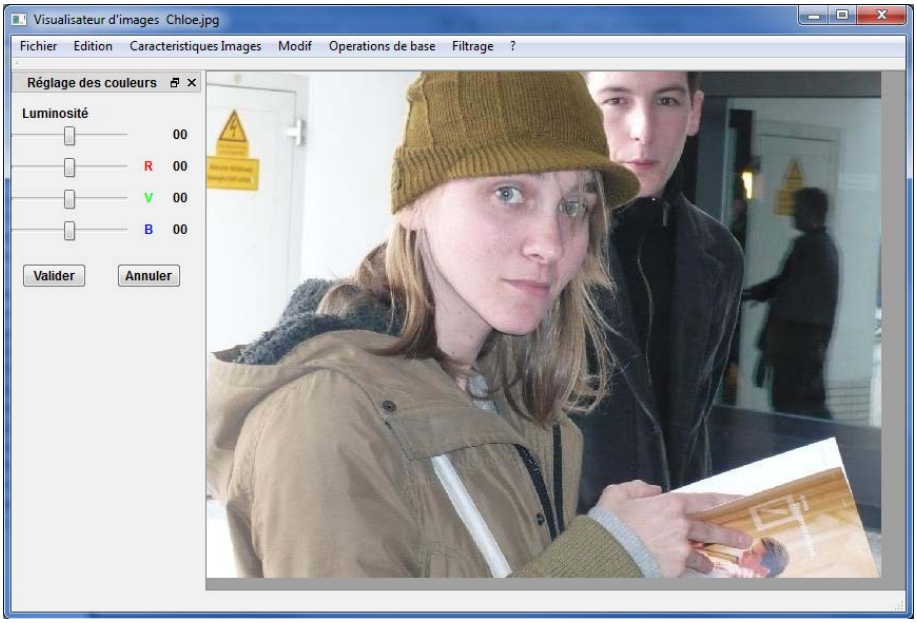
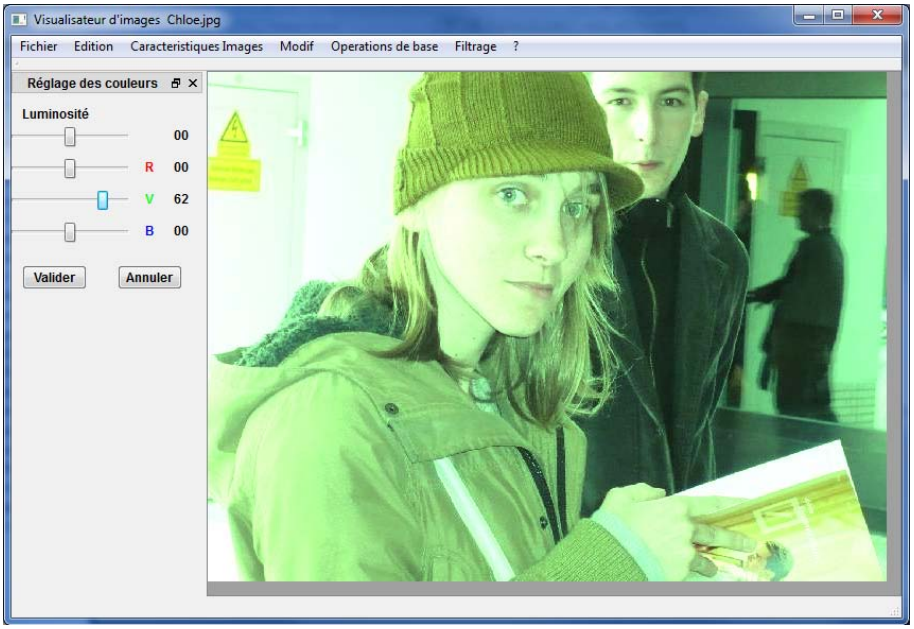


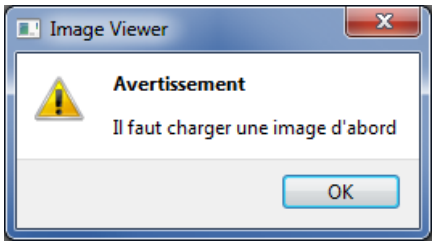
Image initiale



6. Le menu Caractéristiques Images

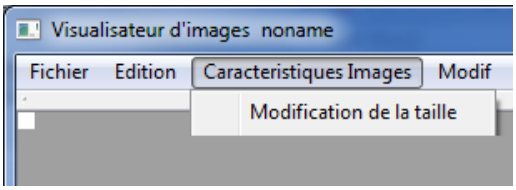
On va créer un nouveau menu Caractéristiques Images entre Edition et Modif

L'appel d'un sous-menu sans avoir une image chargée entrainera toujours un message :

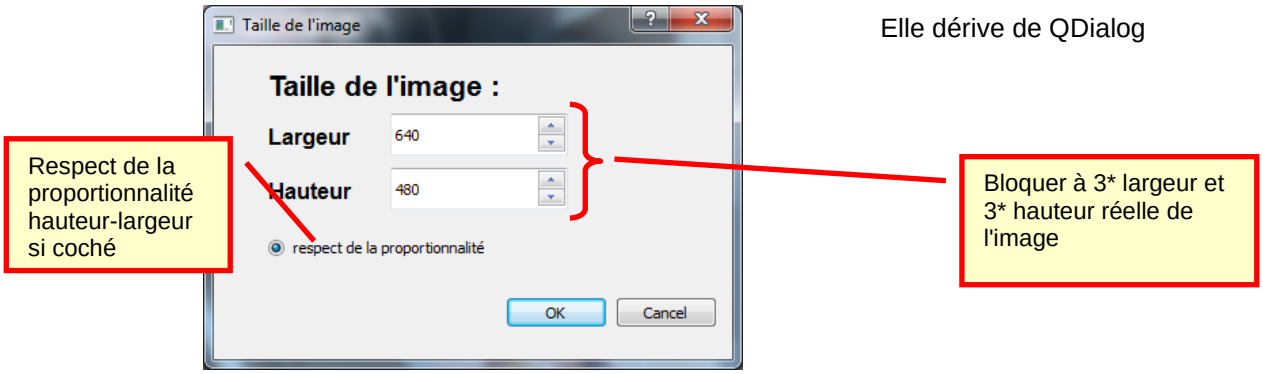


- **Mettre en place la modification de la taille d'une image**

On va créer une entrée Modification de la taille, comme ceci



On va créer une boite nommée MaTaille.ui avec QtDesigner comme ceci:



On va également créer une classe QT , nommée CMaTaille dérivant de QDialog dont l'interface pourra être:

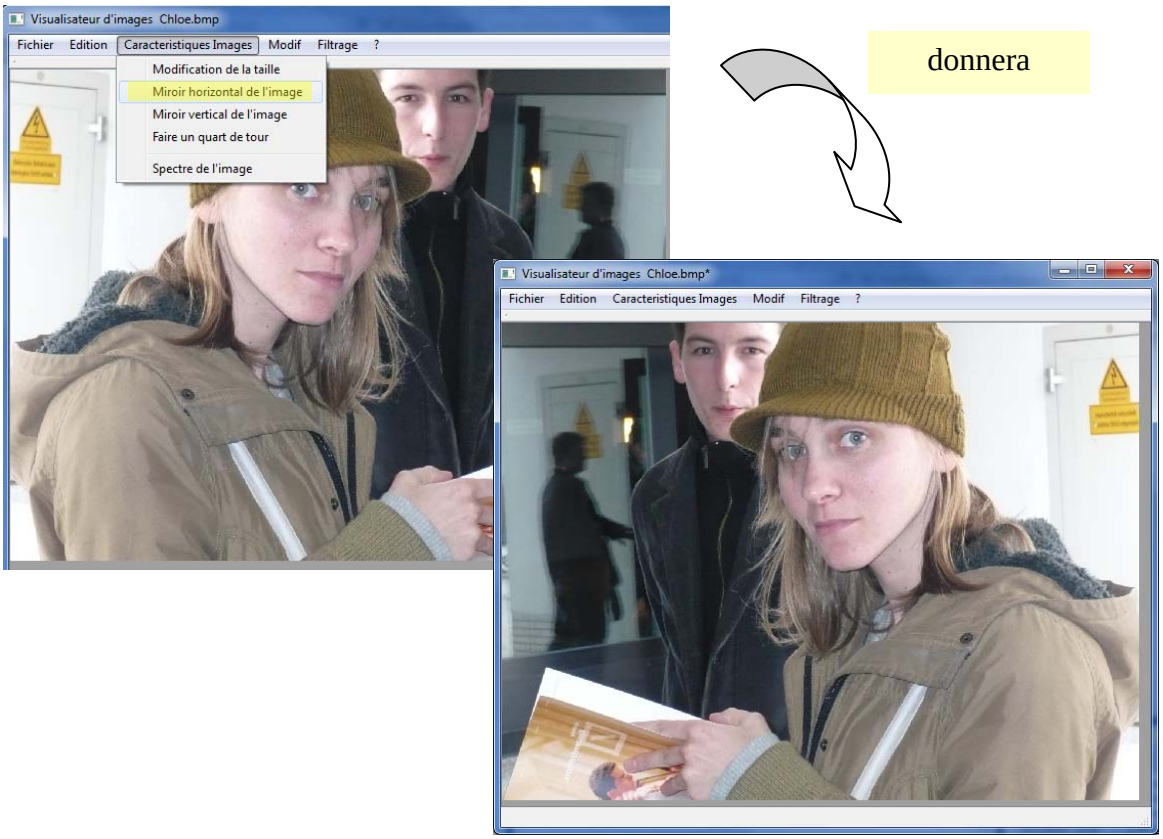
```
class CMaTaille : public QDialog
{
   Q_OBJECT
public:
   CMaTaille(QDialog *parent, int Largeur, int Hauteur);
   ~CMaTaille();
   QPoint getDim () ;
   bool IsModifier ;
private:
   Ui::MaTaille ui;
   int m_nLargeur;
   int m_nHauteur;
private slots:
   void MonOk() ;
   void Larg(int Val);
   void Haut(int Val);
};
```

Sinon une image est chargée, on ouvrira la boite d'une manière Modale, si une modification des dimensions est faite, on appellera une méthode scaled de QImage

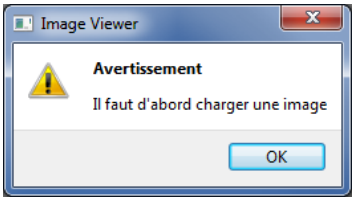
- **Afficher une image Miroir (horizontal/vertical) de l'image initiale**

On va créer deux nouvelles actions : ActionMiroirH et ActionMiroirV, qui appelleront une méthode mirrored de QImage.

On aura par exemple:



L'appel de l'action devra déclencher un message si aucune image n'est chargée comme ceci



- **Faire tourner l'image d'un quart de tour**

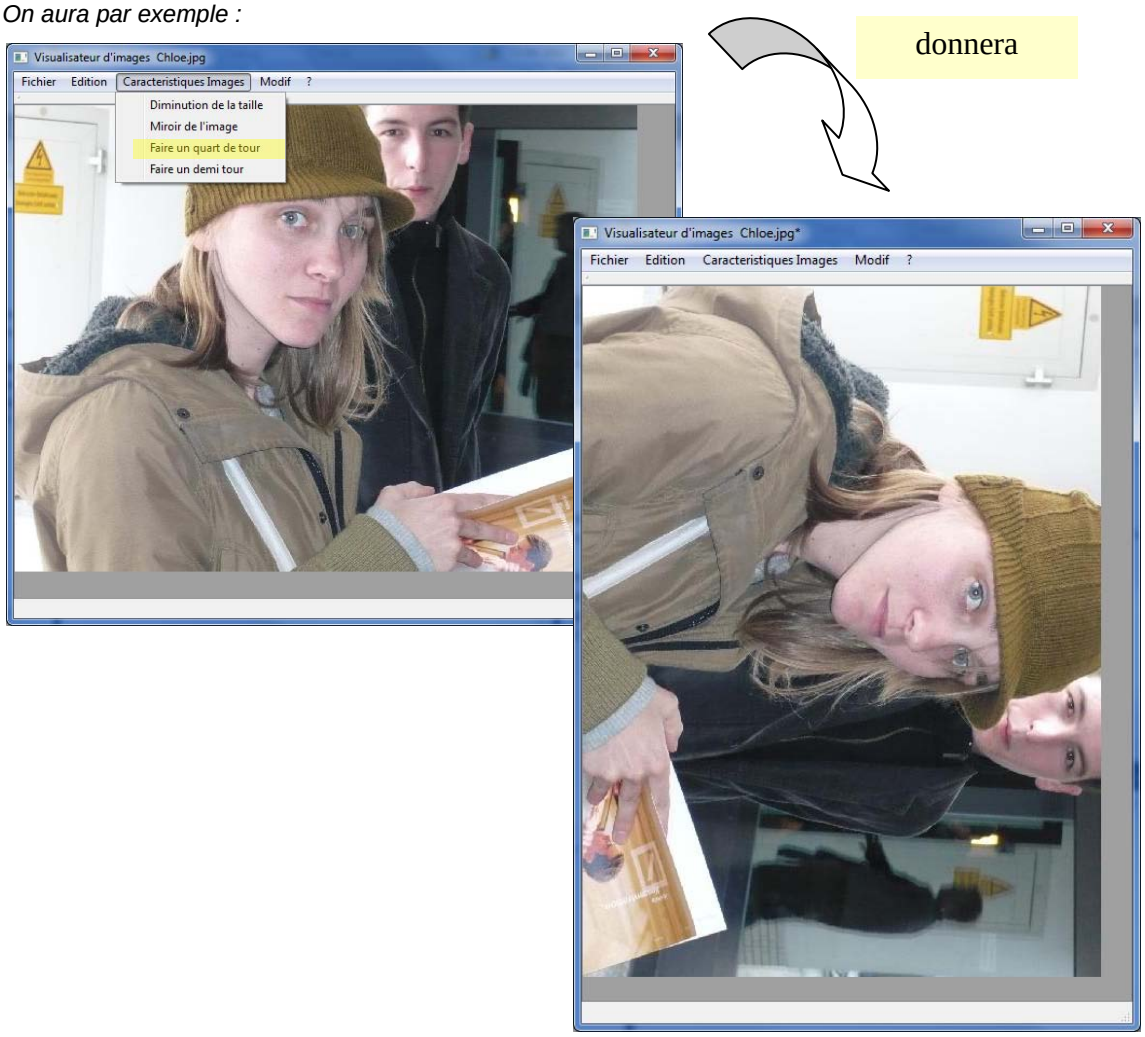
On va créer deux nouvelles actions : actionQuart, qui appellera une méthode Tourner() à écrire.

Celle-ci peut avoir ce prototype :

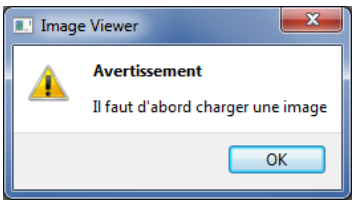
```
QXImage * Tourner        ( int Quart);
```

Nombre de quart de tours

On aura par exemple :

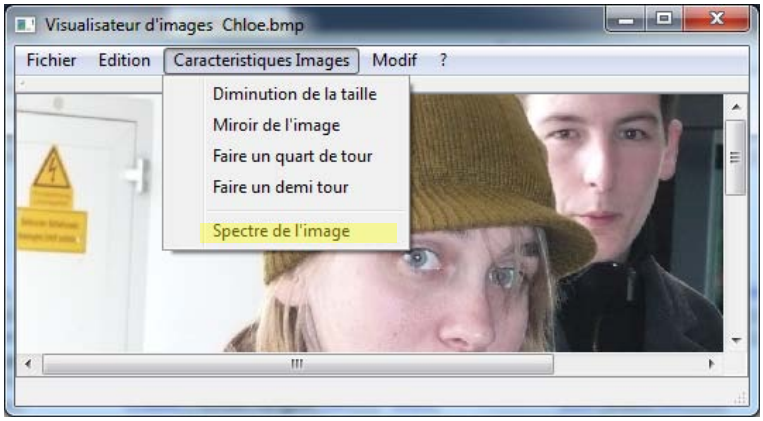


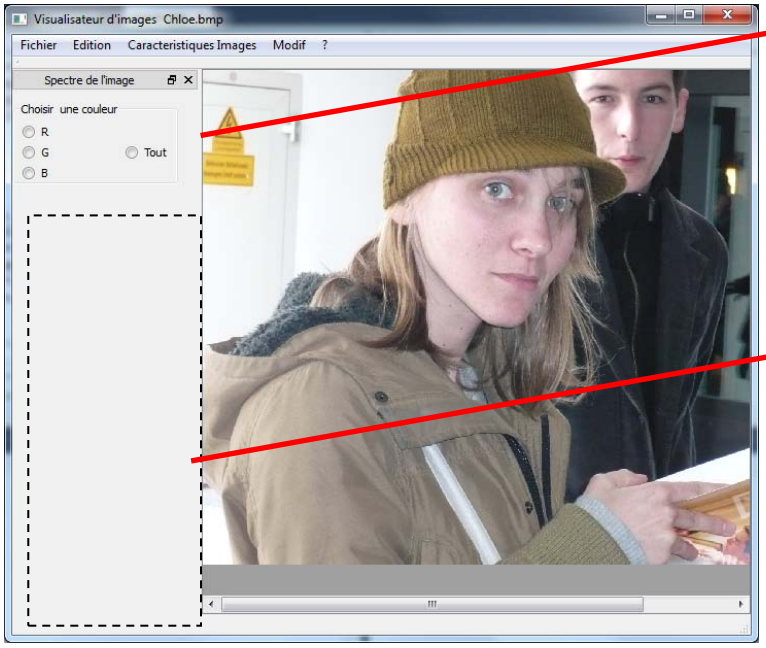
L'appel de l'action devra déclencher un message si aucune image n'est chargée comme ceci



- **Mettre en place d'un histogramme des couleurs**

On va créer une nouvelle action : actionHisto qui appellera le SLOT Histo. Elle sera checkable, mais on la fera entrer dans le dispositif déjà créer qui ne permet de checker qu'une seule des actions checkables possibles.





Les QRadioButton sont dans un même QGroupBox

On place ici un QWidget que l'on nommera MonHisto

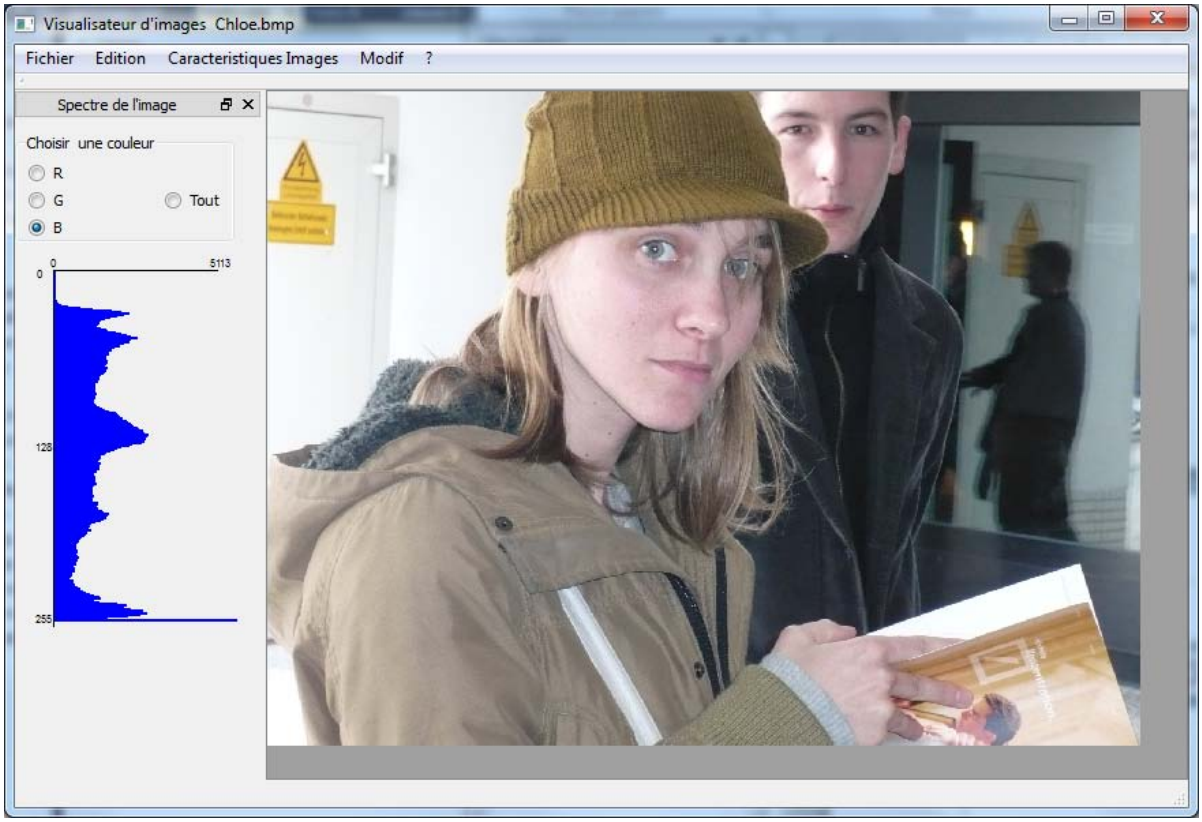
On créera une classe CHisto dérivant de QWidget dont l'interface pourra être :

```
class CHisto : public QWidget
{
   Q_OBJECT

public:
   CHisto(QWidget *parent);
   ~CHisto();
   void setHisto( int * Tab, int Maxi, int Coul) ;
private:
   int * m_nTab;
   int      m_nMaxi ;
   int      m_nCoul ;  // 0 = Noir, 1= Red, 2 = Green, 3= Bleu
protected:
   void paintEvent(QPaintEvent * evt) ;
};
```

On fera une promotion de MonHisto en CHisto.

La classe CHisto aura pour rôle de dessiner les histogrammes souhaités dans le QWidget MonHisto comme ci par exemple :



L'histogramme sera coloré de la couleur choisi et Noir si Tout est choisi. Les SLOTS de chaque bouton feront appel à une méthode privée de Viewer: qui pourra avoir ce prototype:

```
void AfficheHistogramme(int Coul) ;
```

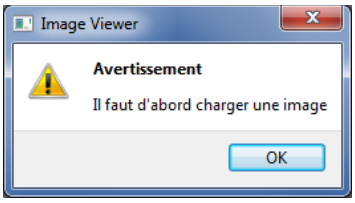
Coul sera un numéro de couleur, par exemple Tout = 0, Rouge = 1, Vert = 2 et Bleu = 3

Cette méthode fera appel à une méthode publique de QXImage que vous écrirez, elle pourra avoir ce prototype:

```
int * Histo ( int Coul, int &Maxi) ;
```

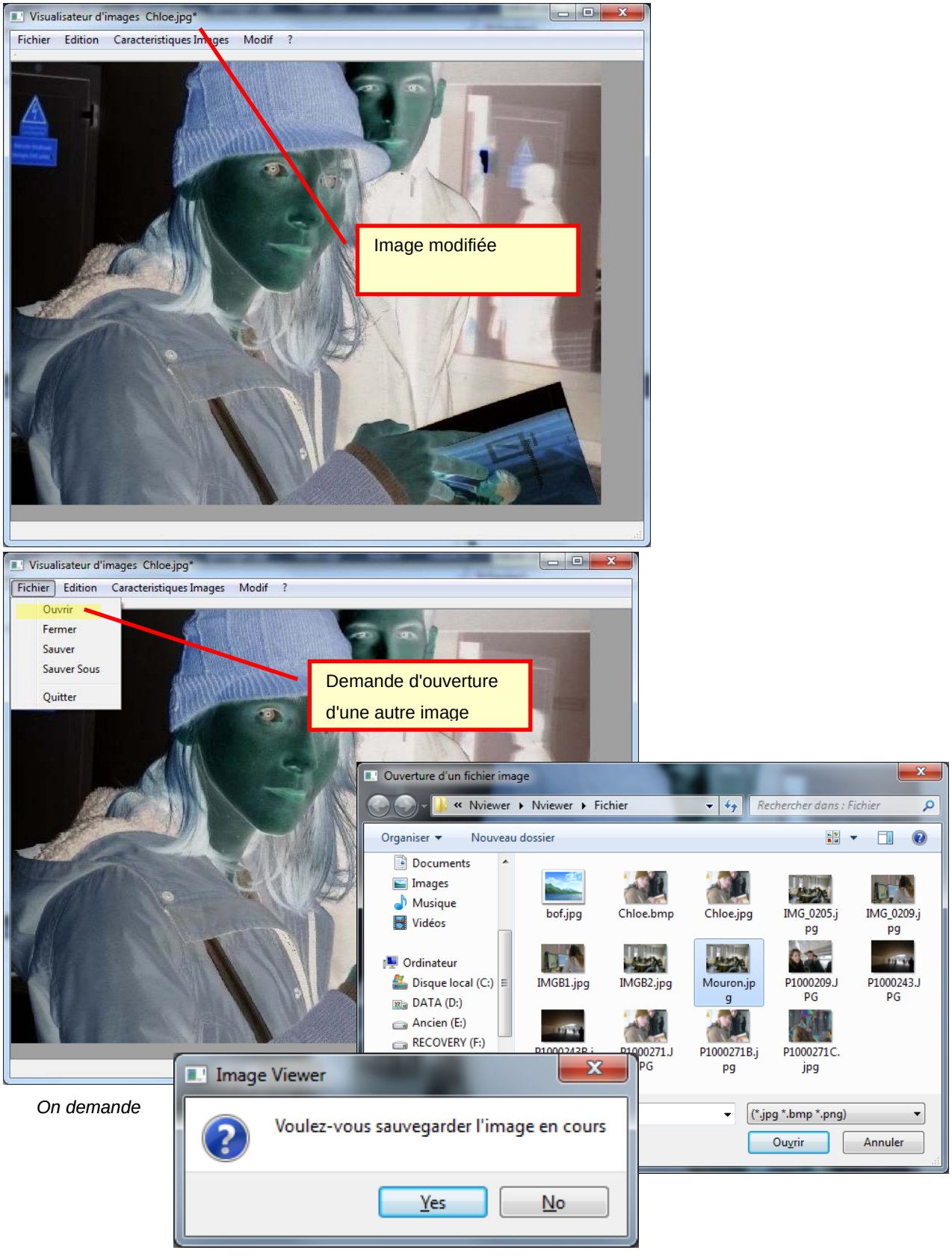
Elle reverra un tableau dynamique de 256 entiers, l'entier numéro x contenant le nombre de pixels d'intensité x dans l'image. Pour le choix de Tout, on fera une moyenne des intensités R, V et B

L'appel de l'action devra déclencher un message si aucune image n'est chargée comme ceci



7. Gestion des modifications

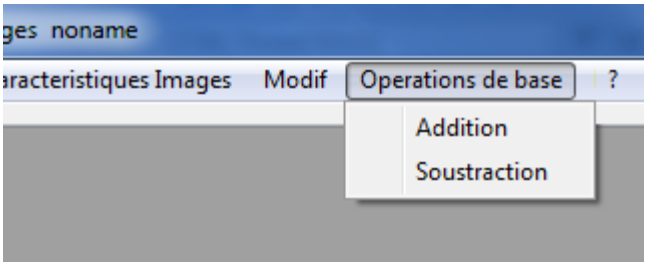
On veillera pour que, à chaque fois que l'on demandera une ouverture d'une nouvelle image, ou que l'on exercera un undo, si l'image en cours a été modifiée, on demande si ou veut la sauvegarder. Par exemple comme ceci:



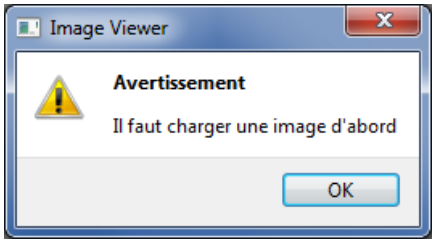
On demande

8. Le Menu Operations de base

On va créer un Menu Opération de base entre Modif et ?



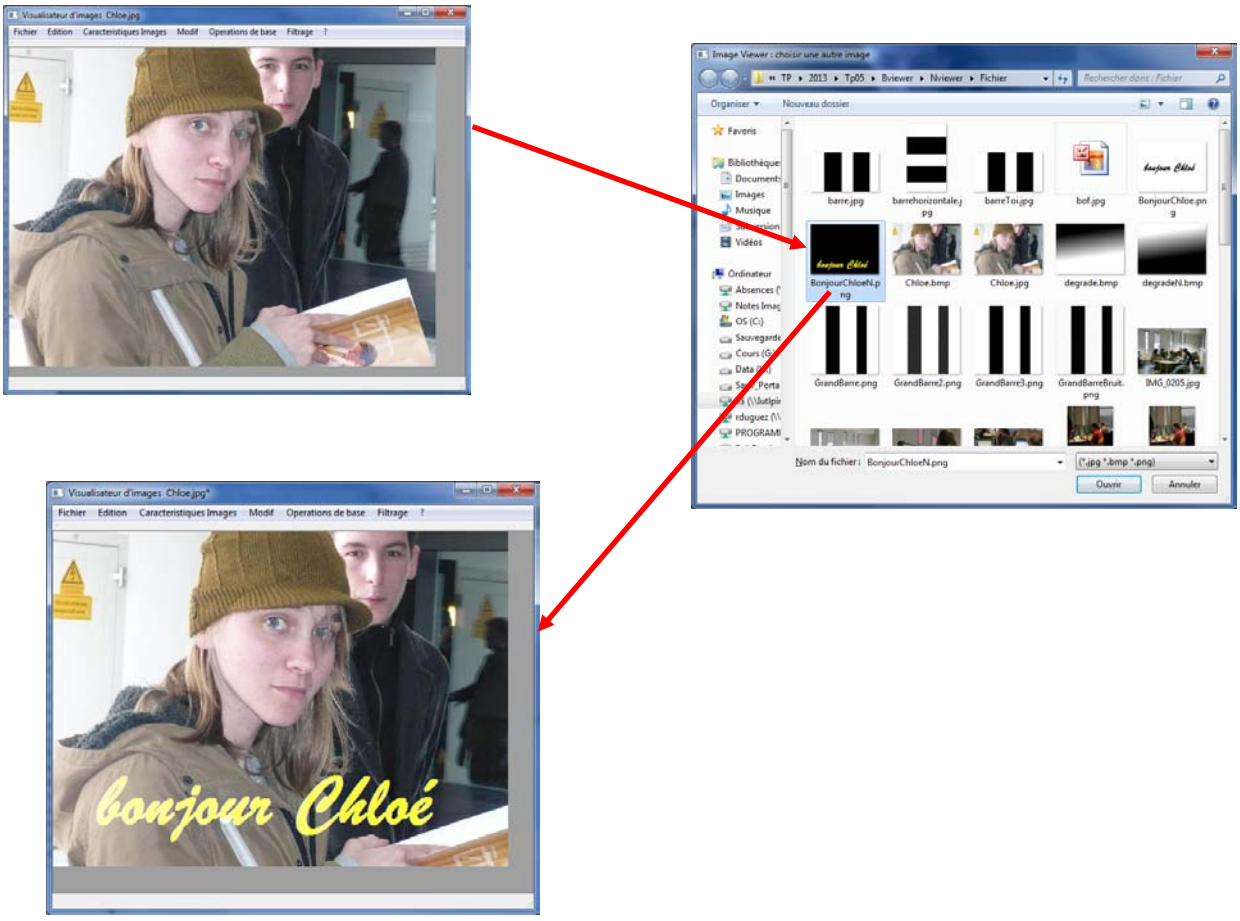
L'appel d'un sous-menu sans avoir une image chargée entrainera toujours un message :



- **Addition :**

On veut créer une image "somme" de deux Images

Pour cela un va charger une image, puis on appellera le menu addition



Pour cela on créera une méthode Somme de QXImage dont le prototype pourra être :

```
QXImage * Somme ( QXImage *tmp ) ;
```

On fera attention, la taille des deux Images ne sont pas nécessairement les mêmes . On fera pour chaque pixel concerné la somme de chaque couleur en veillant à n'avoir que des valeurs en 0 et 255

- **Soustraction:**

On fera de même pour faire la soustraction en deux images