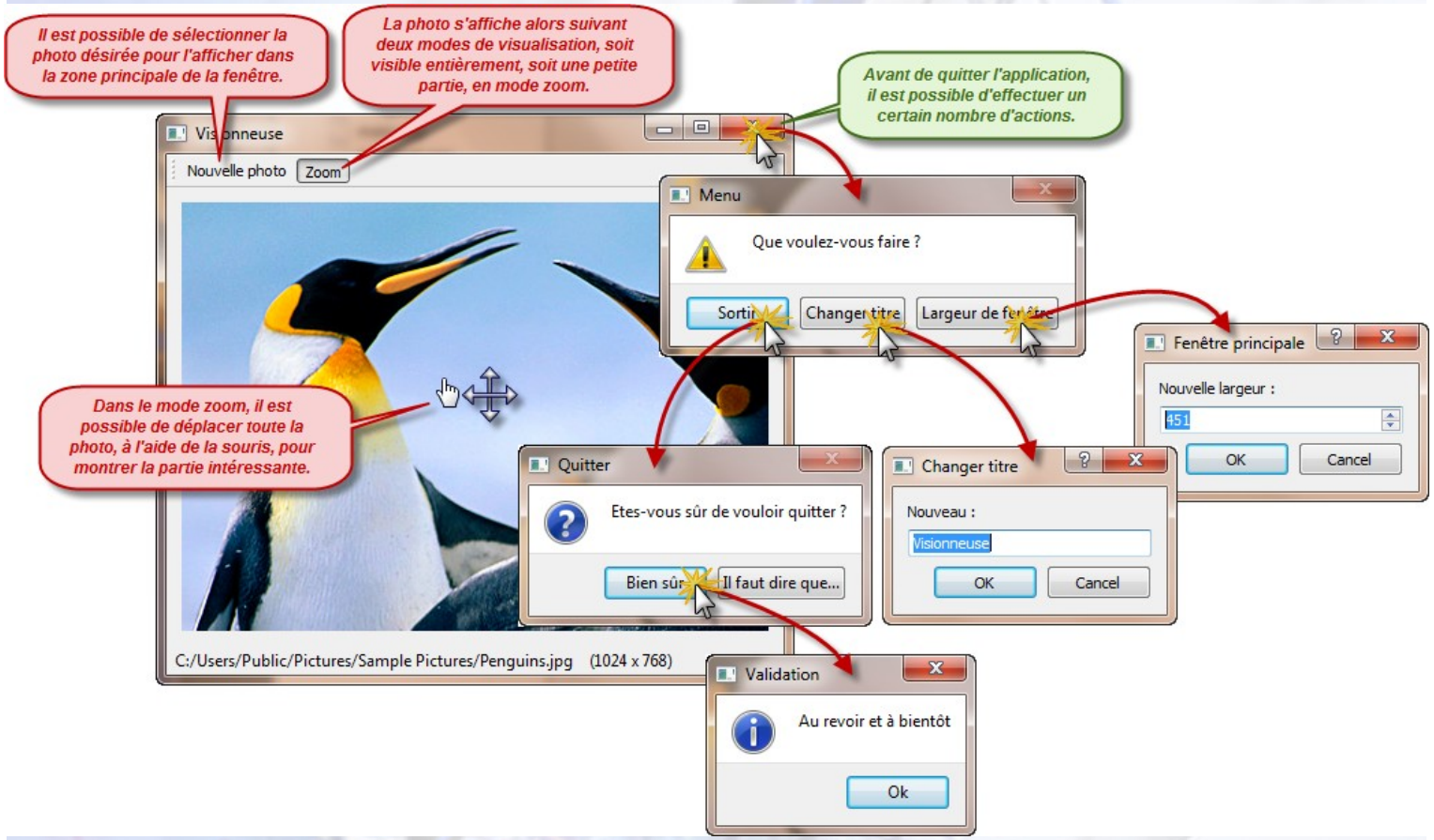
Au travers de ce projet, nous allons voir comment travailler avec les images. A cette occasion, nous en profiterons pour mettre en œuvre des boîtes de dialogue prédéfinies et de voir également comment récupérer l'ensemble des octets stockés dans un fichier.

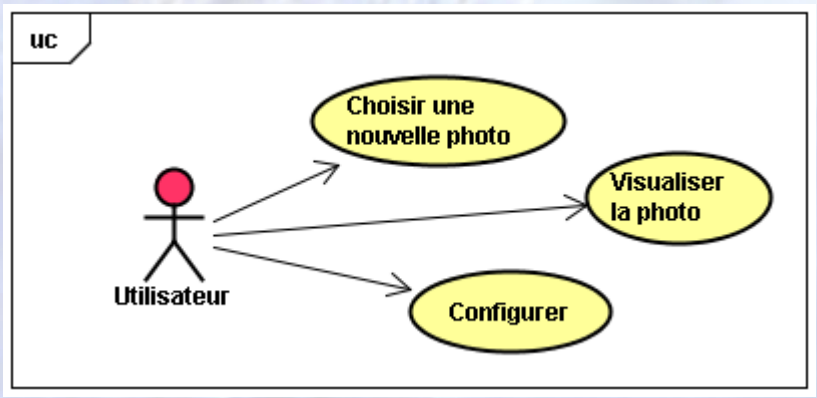
Plus précisément, dans cette application, nous devons choisir une photo présente sur le disque dur de l'ordinateur afin de la visualiser dans la zone principale de la fenêtre. A partir de là, Il est possible de présenter la photo suivant deux types d'affichage :

- *Soit nous affichons la photo complète qui s'adapte alors automatiquement à la dimension de la fenêtre principale de l'application.*
- *Soit nous proposons le mode zoom qui permet alors de voir la photo en taille réelle. Vous pouvez dès lors choisir la partie à afficher à l'aide du curseur de la souris. Dans ce cas précis, le curseur prend la forme d'une main.*

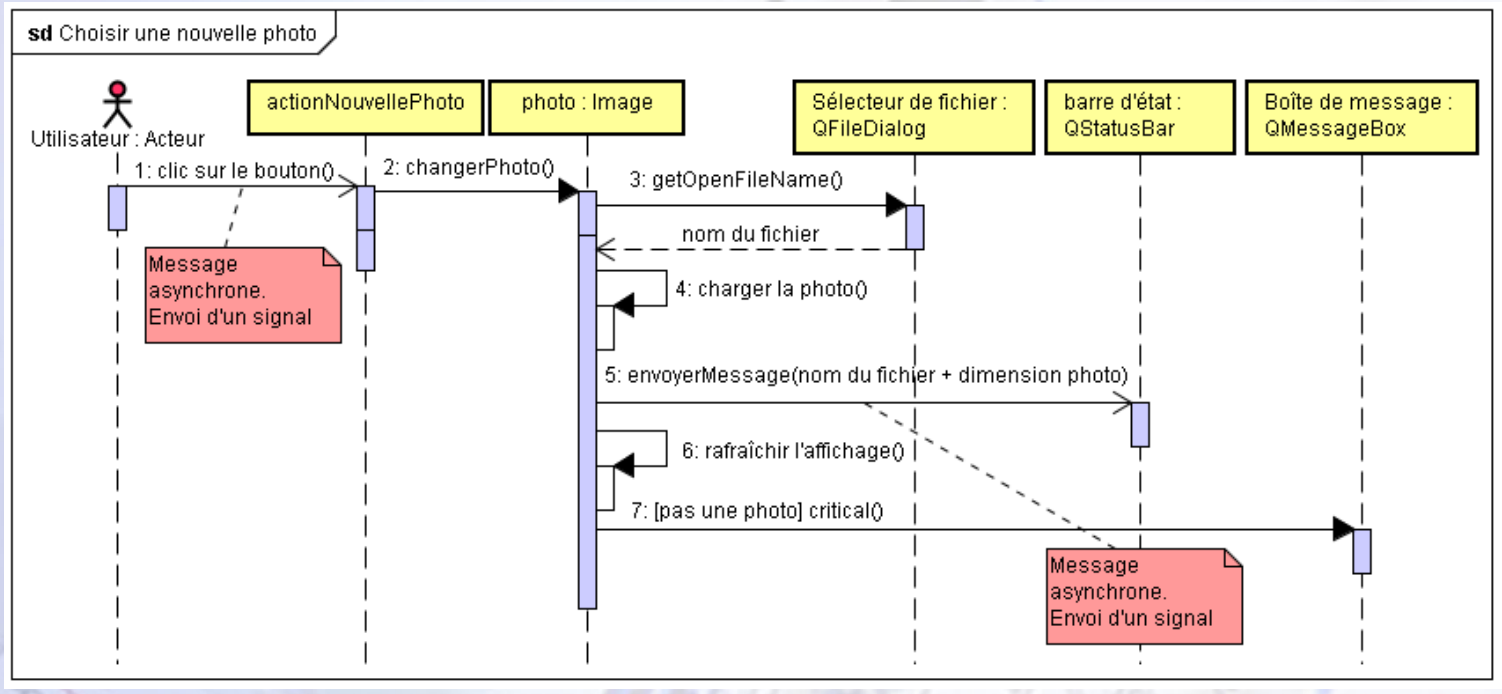


Nous profitons de l'occasion pour utiliser un certain nombre de boîtes de dialogue prédéfinies, comme le sélecteur de fichier, les boîtes de message ainsi que les boîtes de saisie.

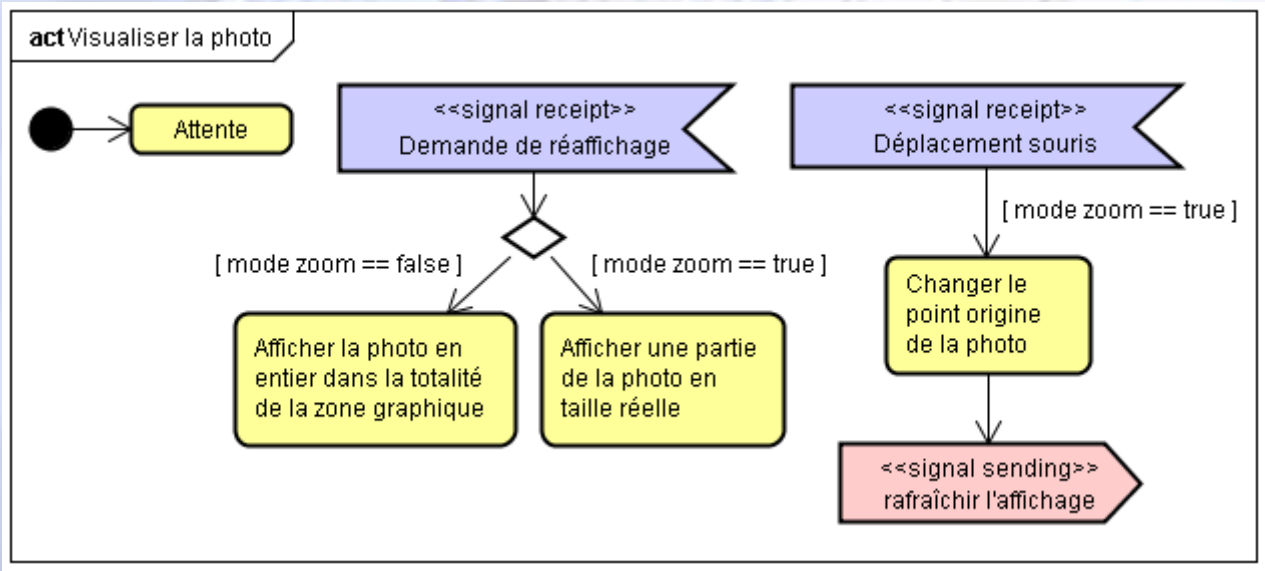
MODÉLISATION – DIAGRAMME DE CAS D'UTILISATION



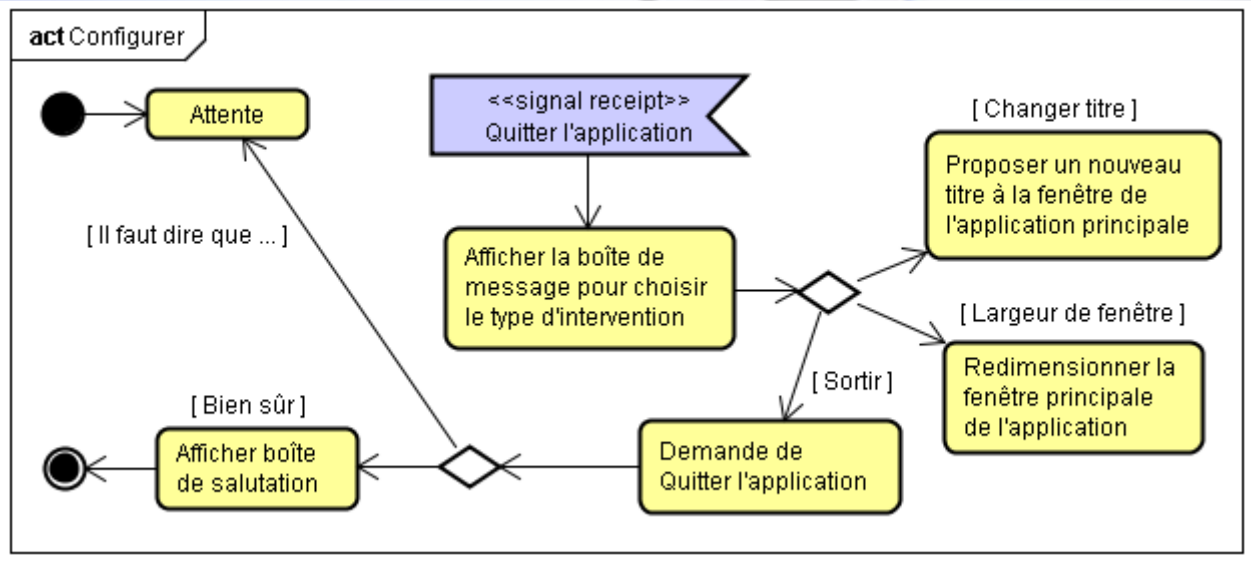
MODÉLISATION – DIAGRAMME DE SÉQUENCE CHOISIR UNE NOUVELLE PHOTO



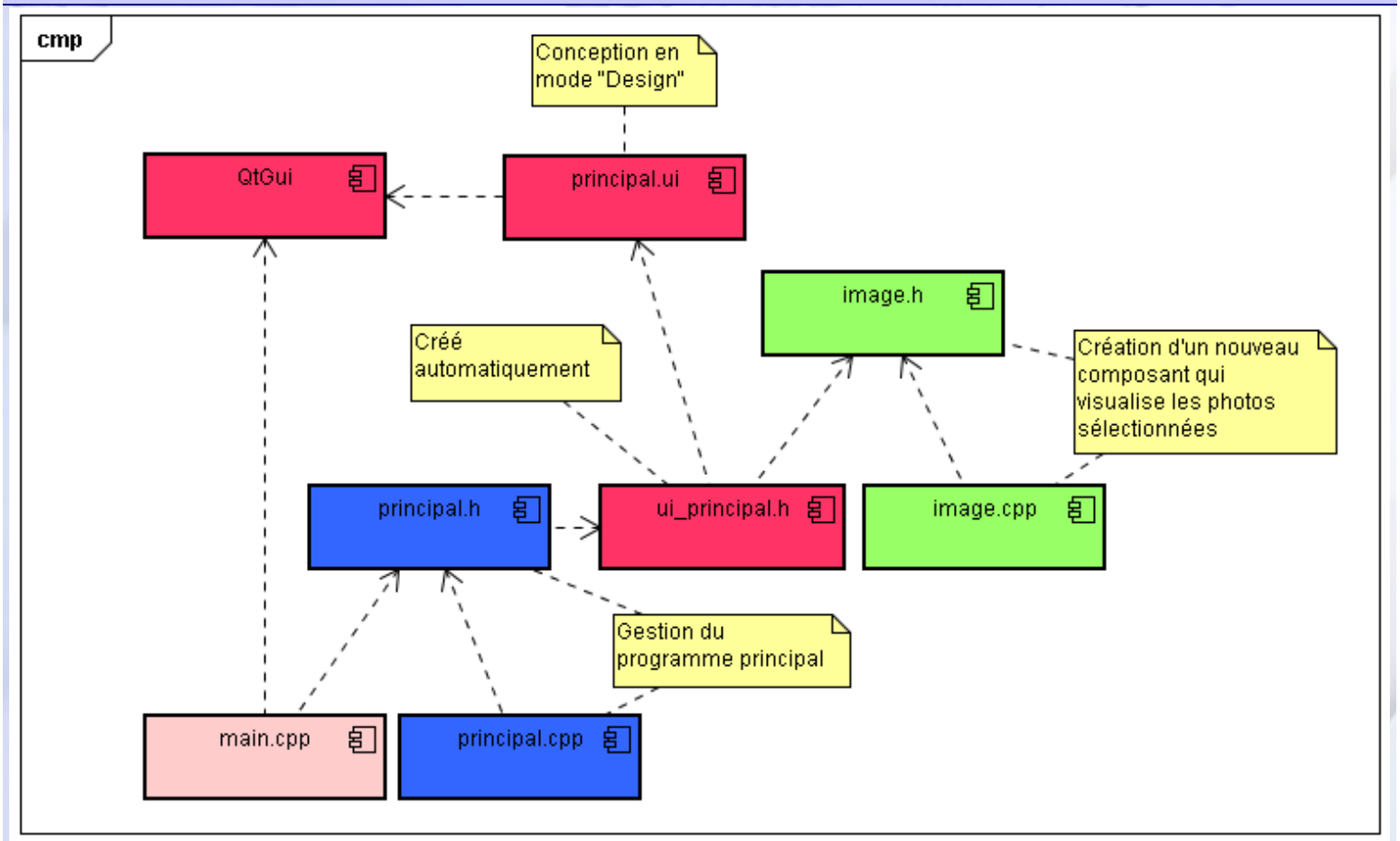
MODÉLISATION – DIAGRAMME D'ACTIVITÉ DE VISUALISÉ LA PHOTO



MODÉLISATION – DIAGRAMME D'ACTIVITÉ DE CONFIGURER



MODÉLISATION – DIAGRAMME DE COMPOSANTS



LES BOÎTES DE DIALOGUE – SÉLECTEUR DE FICHIERS

Il existe un certain nombre de boîtes de dialogue prédéfinies dans QT Designer. La première d'entre-elles, le sélecteur de fichier, permet de choisir le nom du fichier (ou d'un répertoire) qui va servir à l'ouvrir ou proposer un enregistrement avec le nom introduit.

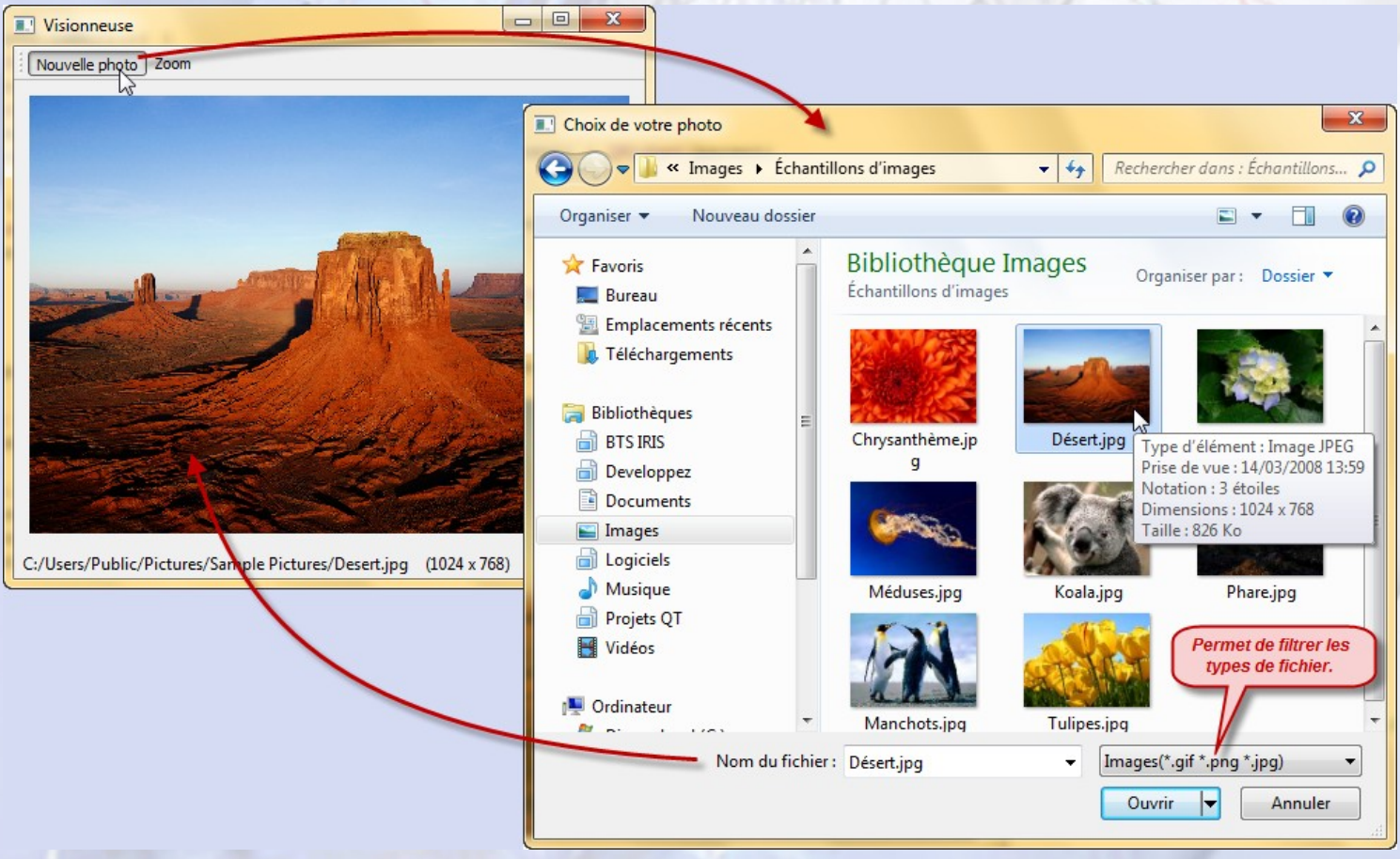
La classe correspondante s'appelle ***QFileDialog***. Elle possède un certain nombre de méthodes statiques qui permettent d'activer le sélecteur de fichier dans le mode souhaité.

- ***getOpenFileName()*** *: cette méthode permet de récupérer le nom du fichier sélectionné. L'ouverture réelle du fichier se fait par la suite. Ce n'est pas au travers de la boîte de dialogue que nous ouvrons le fichier. Cette boîte a juste pour but de nous permettre de localiser le fichier avec son nom complet (imbrication des répertoires).*
- ***getOpenFileNames()*** *: cette méthode est très similaire à la précédente, mais elle permet de faire le choix de plusieurs fichiers en même temps.*
- ***getSaveFileName()*** *: cette méthode permet d'écrire ou de sélectionner le nom du fichier souhaité à l'endroit prévu dans l'arborescence des répertoires. Là aussi, la sauvegarde ne s'effectue que dans un deuxième temps, lorsque le nom du fichier est parfaitement connu.*
- ***getExistingDirectory()*** *: cette méthode renvoi le nom du répertoire sélectionné.*

Je rappelle qu'une méthode statique permet d'évoquer une fonctionnalité sans création d'objet. Il s'agit d'une méthode de classe. A ce titre, pour lancer une telle méthode, vous devez la préfixer de la classe qui la supporte à l'aide de l'opérateur de portée « :: ».

Toutes ces méthodes renvoient une chaîne de caractères qui correspond au nom du fichier (ou du répertoire) sélectionné, bien entendu, si l'utilisateur approuve ce choix en cliquant sur le bouton de validation. Dans le cas contraire, la méthode renvoie une chaîne vide. Vous êtes donc dans l'obligation de vérifier le contenu de cette chaîne pour élaborer la stratégie correspondante.

Il est possible de proposer des filtres pour que le choix des fichiers soit orienté suivant le type de fichier à récupérer. Un filtre se désigne sous forme de chaîne de caractères à l'intérieure de laquelle vous précisez un intitulé, par exemples « Les images », suivi, entre parenthèses, de la liste des extensions de fichier attendues (.png *.gif *.jpg *.jpeg). Il est même possible de proposer plusieurs filtres en une seule fois. Dans ce cas, il faut respecter le canevas précédent pour chaque filtre chacun étant alors séparé par le double opérateur « ; », par exemple :* "Les images (*.png *.xpm *.jpg);;Fichiers texte (*.txt);;Documents XML (*.xml)"



LES BOÎTES DE DIALOGUE – BOÎTES DE MESSAGE

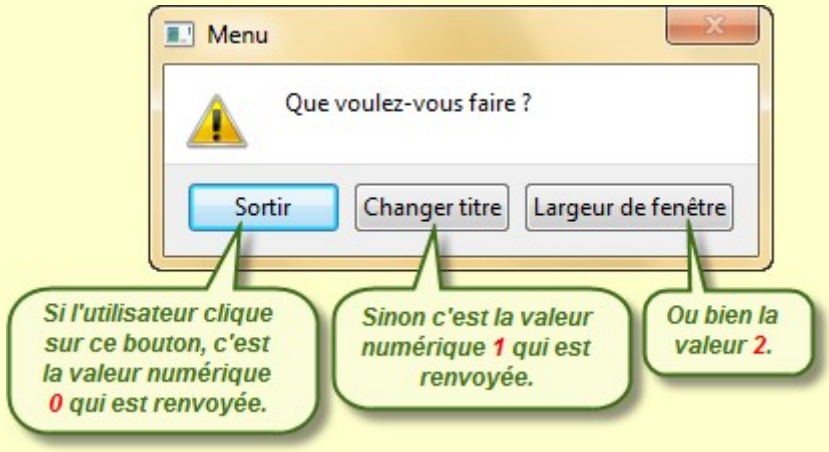
Le deuxième type de boîtes de dialogue que nous avons souvent besoin sont les boîtes de message. Ces dernières nous permettent d'élaborer une alerte modale fenêtrée. Dans ce cas de figure, comme pour toutes les boîtes de dialogue, nous devons impérativement cliquer sur l'un des boutons présents pour pouvoir évoluer dans l'application. Il peut s'agir d'un simple renseignement, mais cela peut-être un avertissement spécifique qui réclame alors un choix déterminé de la part de l'utilisateur.

La classe correspondante s'appelle ***QMessageBox***. Elle possède un certain nombre de méthodes statiques qui permettrent de choisir le type d'alerte souhaité. A chaque type d'alerte correspond une icône correspondante :

- ***information()*** *: cette méthode ouvre une boîte de dialogue avec un simple message informatif qu'il suffit juste de consulter. Nous cliquons ensuite sur le bouton de validation pour passer à la suite et pour informer que le message a bien été lu.*
- ***question()*** *: cette méthode nous donne un message qui réclame un choix de notre part. Il faut alors proposer des boutons spécifiques en conséquence.*
- ***warning()*** *: cette méthode ouvre une boîte de dialogue avec un message d'avertissement. Généralement, ce type de boîte de message nous permet de valider un choix déjà fait.*
- ***critical()*** *: cette méthode ouvre une boîte de dialogue, avec cette fois-ci un message d'alerte. L'utilisateur doit alors être très attentif à ce type de message et prendre en compte l'avertissement proposé.*

En réalité, le choix de la méthode influence uniquement l'icône qui apparaîtra dans la boîte de dialogue. Avec n'importe quelle type de boîte de dialogue, vous pouvez choisir le nombre de boutons que vous désirez avec l'intitulé souhaité.

Chacune de ces méthodes renvoie une valeur entière qui correspond au choix effectué par l'utilisateur. Si ce dernier clique sur le premier bouton de la boîte de dialogue, c'est la valeur 0 qui est retournée. Si l'utilisateur clique sur le deuxième bouton, c'est la valeur 1 qui est retournée, etc. Vérifier donc le retour de cette méthode pour élaborer la stratégie adaptée à la circonstance.



LES BOÎTES DE DIALOGUE – BOÎTES DE SAISIE

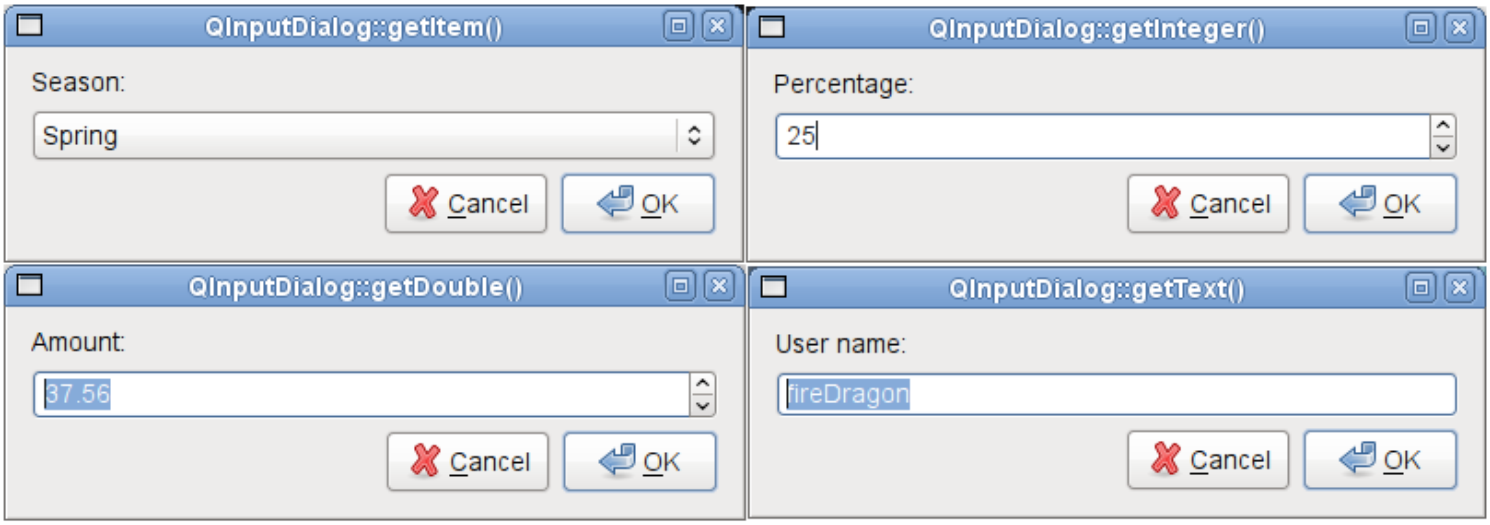
Enfin, les boîtes de dialogue également importantes, sont les boîtes de saisie qui permettent, simplement et de façon ergonomique, de récupérer une valeur à la volée suivant le besoin du moment. Là aussi, nous disposons d'un certain nombre prédéfini de boîtes de saisie adaptée à la situation requise.

La classe correspondante s'appelle ***QInputDialog***. Elle possède un certain nombre de méthodes statiques qui permettent de choisir le type de retour souhaité :

- ***getText()*** *: cette méthode ouvre une boîte de dialogue de saisie qui permet de récupérer une chaîne de caractères, si la saisie est validée.*
- ***getInteger()*** *: cette méthode ouvre une boîte de dialogue de saisie qui permet de récupérer une valeur entière, si la saisie est validée.*
- ***getDouble()*** *: cette méthode ouvre une boîte de dialogue de saisie qui permet de récupérer une valeur réelle, si la saisie est validée.*
- ***getItem()*** *: cette méthode ouvre une boîte de dialogue de saisie qui permet de choisir une chaîne parmi un ensemble d'éléments présents sous forme de boîte de liste, là aussi si la saisie est validée.*

*En réalité, ces boîtes de saisie possèdent le composant interne **QLineEdit** qui est justement spécialisé dans la saisie des valeurs. Il est tout-à-fait possible de contrôler la façon de saisir en utilisant des constantes de cette classe prévues à cet effet. Nous disposons de quatre modes.*

- ***QLineEdit::Normal*** *: mode par défaut. Tout ce que nous écrivons apparaît intégralement dans la zone de saisie.*
- ***QLineEdit::NoEcho*** *: Ici au contraire, plus rien n'apparaît, quelque soit la valeur que vous saisissez. Cela permet d'introduire des valeurs secrètes sans qu'aucune personne ne voit ce que vous tapez au clavier.*
- ***QLineEdit::Password*** *: Ce mode est plus adapté que le critère précédent. Il montre en plus un astérisque pour chacun des caractères que vous saisissez au clavier. Cela permet de contrôler le nombre de caractères que nous avons déjà introduit sans les visualiser explicitement.*
- ***QLineEdit::PasswordEchoOnEdit*** *: Lorsque vous utilisez toutes ces boîtes de saisie, il est possible de visualiser la valeur précédente, ou si vous voulez, la valeur actuellement en cours. Dans ce mode, la valeur actuelle n'est pas explicitement visible, mais des astérisques sont proposées à la place, ce qui en fait valeur secrète. Par contre, la nouvelle valeur que vous saisissez apparaît en clair.*



PRINCIPAL.H

principal.h — Principal

```
1   #ifndef PRINCIPAL_H
2   #define PRINCIPAL_H
3
4   #include "ui_principal.h"
5   #include <QMainWindow>
6   #include <QFileDialog>
7
8   class Principal : public QMainWindow, public Ui::Principal
9   {
10    Q_OBJECT
11  public:
12    Principal(QWidget *parent = 0);
13  protected:
14    void closeEvent(QCloseEvent *evt);
15  };
16
17  #endif // PRINCIPAL_H
18
```

Prise en compte de la demande de clôture du logiciel

PRINCIPAL.CPP

principal.cpp | <Select Symbol> | Line: 1, Col: 1

```
#include <QtGui>
#include "principal.h"

Principal::Principal(QWidget *parent) : QMainWindow(parent)
{
  setupUi(this);
}

void Principal::closeEvent(QCloseEvent *evt)
{
  int choix = QMessageBox::warning(this, "Menu", "Que voulez-vous faire ?", "Sortir", "Changer titre", "Largeur de fenêtre");
  switch (choix)
  {
    case 0 :
      choix = QMessageBox::question(this, "Quitter", "Etes-vous sûr de vouloir quitter ?", "Bien sûr", "Il faut dire que...");
      if (choix==0) evt->accept();
      else evt->ignore();
      QMessageBox::information(this, "Validation", "Au revoir et à bientôt", "Ok");
      break;
    case 1 :
      setWindowTitle(QInputDialog::getText(this, "Changer titre", "Nouveau : ", QLineEdit::Normal, windowTitle()));
      // Si vous ne désirez pas montrer le titre actuel, vous pouvez préciser moins d'arguments comme ci-dessous
      // setWindowTitle(QInputDialog::getText(this, "Changer titre", "Nouveau titre :");
      evt->ignore();
      break;
    case 2 :
      double ratio = (double) width() / height();
      int largeur = QInputDialog::getInt(this, "Fenêtre principale", "Nouvelle largeur :", QLineEdit::Normal, width());
      // Si vous ne désirez pas montrer la largeur actuelle, vous pouvez préciser moins d'arguments comme ci-dessous
      // int largeur = QInputDialog::getInt(this, "Fenêtre principale", "Nouvelle largeur :");
      int hauteur = largeur / ratio;
      setFixedWidth(largeur);
      setFixedHeight(hauteur);
      evt->ignore();
      break;
  }
}
```

IMAGE.H

image.h | <Select Symbol>

```
#ifndef IMAGE_H
#define IMAGE_H

#include <QWidget>

class Image : public QWidget
{
  Q_OBJECT

public:
  Image(QWidget *parent = 0);

protected:
  void paintEvent(QPaintEvent *evt);
  void mousePressEvent(QMouseEvent *evt);
  void mouseMoveEvent(QMouseEvent *evt);
  void mouseReleaseEvent(QMouseEvent *evt);
signals:
  void envoyerMessage(QString message);
private slots:
  void activerZoom(bool activer);
  void changerPhoto();
private:
  QPixmap photo;
  bool zoom;
  QPoint origine, contact, decalage;
};

#endif // IMAGE_H
```

Rafraîchissement du composant Image pour gérer l'affichage de la photo sélectionnée en mode zoom ou pas.

Mémorisation de l'attribut contact pour enregistrer l'endroit du curseur de la souris, juste au moment du clic, pour exploiter le déplacement en mode zoom.

Déplacement de la photo en mode zoom si l'utilisateur maintien l'action sur le bouton gauche de la souris.

Conserver le décalage actuel en mode zoom lorsque l'utilisateur relâche le bouton gauche de la souris.

Envoi d'un message pour la barre d'état donnant le nom du fichier de la nouvelle photo choisie.

Choisir le mode zoom ou pas.

Choisir une nouvelle photo.

Ensemble d'attributs qui mémorisent : la photo sélectionnée, le choix du mode zoom et les différents points pour le calcul du placement de l'image .

IMAGE.CPP

```
#include "image.h"
#include <QtGui>

Image::Image(QWidget *parent) : QWidget(parent)
{
  zoom = false;
  decalage = origine = rect().topLeft();
}

void Image::paintEvent(QPaintEvent *evt)
{
  if (!photo.isNull()) {
    QPainter dessin(this);
    if (zoom) {
      QRect zone = photo.rect();
      zone.setTopLeft(origine);
      dessin.drawPixmap(rect().topLeft(), photo, zone);
    }
    else {
      QPixmap image = photo.scaledToWidth(rect().width());
      dessin.drawPixmap(rect(), image, rect());
    }
  }
}

void Image::changerPhoto()
{
  QString nomFichier = QFileDialog::getOpenFileName(this, "Choix de votre photo", "", "Images (*.gif *.png *.jpg)");
  if (!nomFichier.isEmpty())
  {
    photo.load(nomFichier);
    envoyerMessage(QString("%1    (%2 x %3)").arg(nomFichier).arg(photo.width()).arg(photo.height()));
    update();
    if (photo.width() == 0) QMessageBox::critical(this, "Problème", "Ce n'est pas une photo...", "Ok");
  }
}

void Image::activerZoom(bool activer)
{
  zoom = activer;
  if (zoom) setCursor(Qt::PointingHandCursor);
  else setCursor(Qt::ArrowCursor);
  update();
}

void Image::mousePressEvent(QMouseEvent *evt)
{
  contact = evt->pos();
}

void Image::mouseMoveEvent(QMouseEvent *evt)
{
  if (zoom) {
    origine = contact - evt->pos() + decalage;
    update();
  }
}

void Image::mouseReleaseEvent(QMouseEvent *evt)
{
  decalage += contact - evt->pos();
}
```

Dès la création de l'objet, il faut connaître la position du composant par rapport à la fenêtre. Le mode zoom est désactivé par défaut.

Rafraîchissement automatique du composant.

Si la photo est présente.

En mode zoom, placement du point origine de la photo de telle sorte que la partie visible corresponde à l'attente de l'utilisateur.

Tracé de la photo suivant les critères retenus.

Lorsque le mode zoom est inactif, création d'une nouvelle photo qui correspond parfaitement à la dimension du composant. Du coup, la photo se voit entièrement.

Obtention d'une nouvelle photo à l'aide du sélecteur de fichiers.

Chargement de l'image.

Envoi d'un message pour la barre d'état avec le nom du fichier sélectionné et les dimension de la photo.

Boîte de message d'erreur au coup où le fichier sélectionné n'est pas une photo.

Méthode de traitement, lancée lorsque l'utilisateur clique sur le bouton correspondant, qui gère le choix du mode zoom. Le curseur de la souris est changé en conséquence et le rafraîchissement automatique de l'affichage est sollicité.

Mémorisation de l'attribut contact qui enregistre l'endroit du curseur de la souris, juste au moment du clic, pour exploiter correctement le déplacement en mode zoom.

En mode zoom, à chacun des déplacement de la souris, calcul de la nouvelle origine afin que la partie visible de l'image soit correctement positionnée.

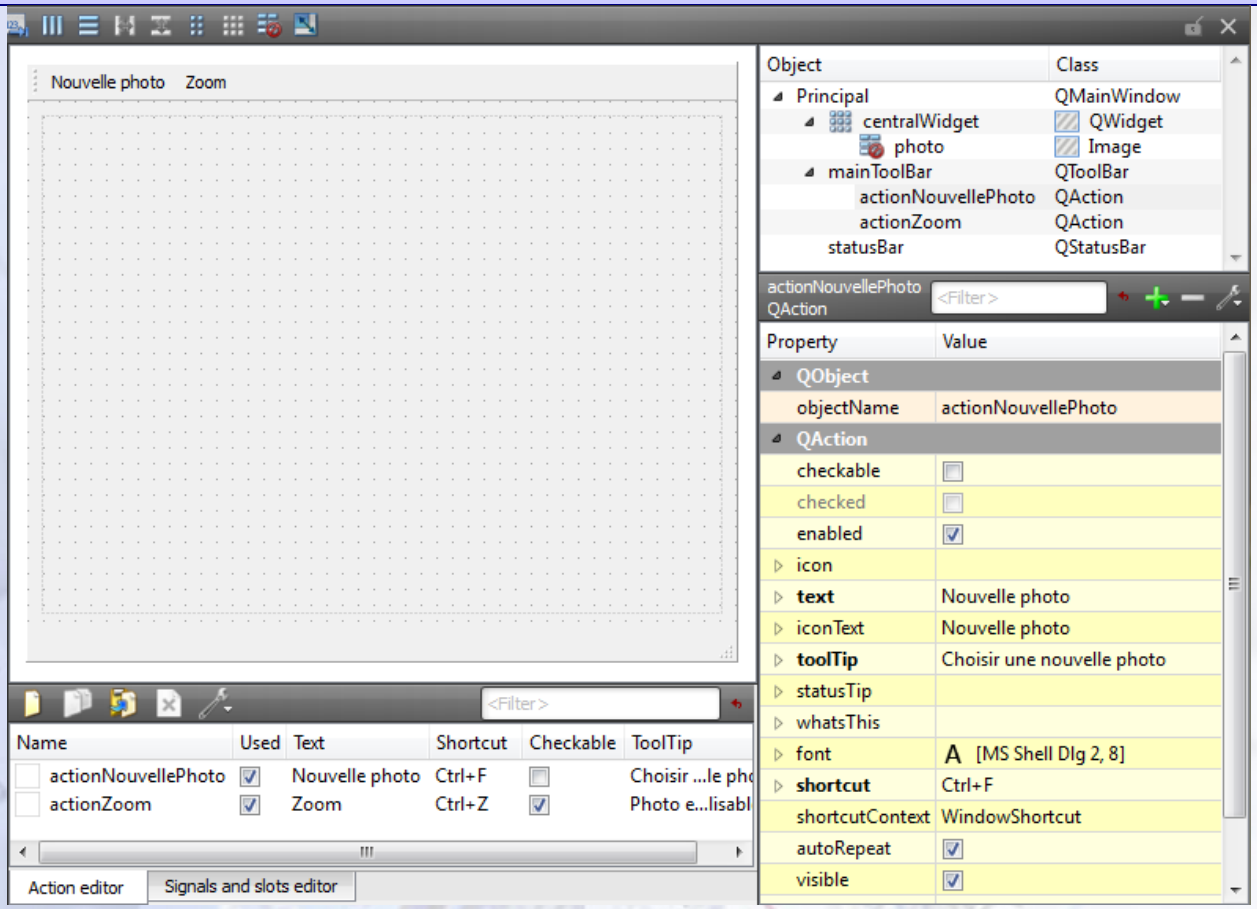
Rafraîchissement de l'image pour prendre en compte ces nouveaux calculs afin que la photo suive ainsi le mouvement de la souris.

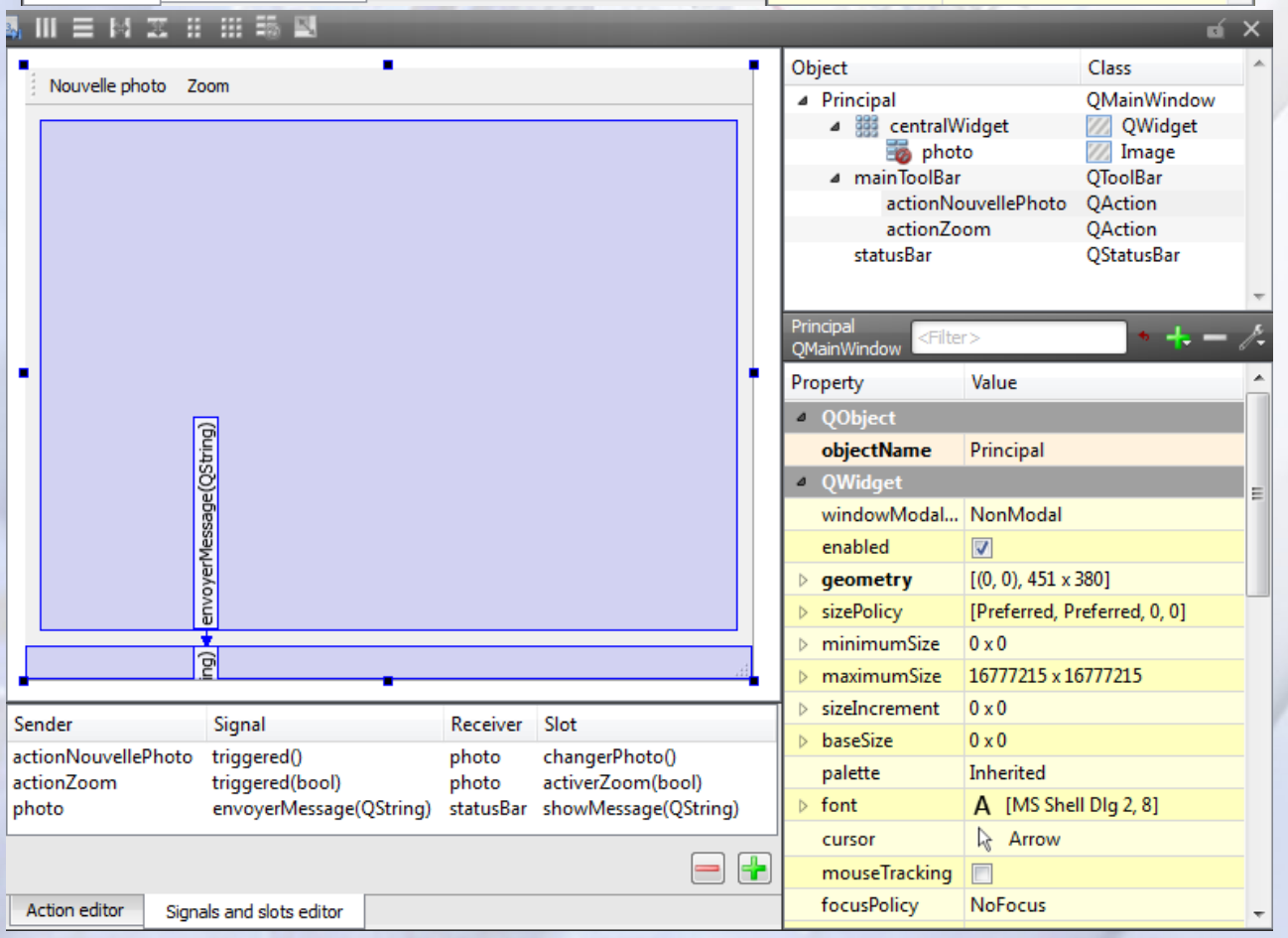
Conserver le décalage calculé lorsque l'utilisateur relâche le bouton gauche de la souris.

*Pour la gestion des images, dans les codes sources, nous avons utilisé la classe **QPixmap** qui est plus spécialisée sur le rendu de l'image dans l'IHM. Il existe une autre classe, **QImage**, qui réalise le même type de traitement mais qui se préoccupe plus des entrées-sorties et peut directement accéder à chacun des pixels en particulier.*

*Si vous désirez prendre **QImage** en lieu et place de **QPixmap**, utilisez alors la méthode drawImage**()** de **QPainter** à la place de drawPixmap().*

x PRINCIPAL.UI



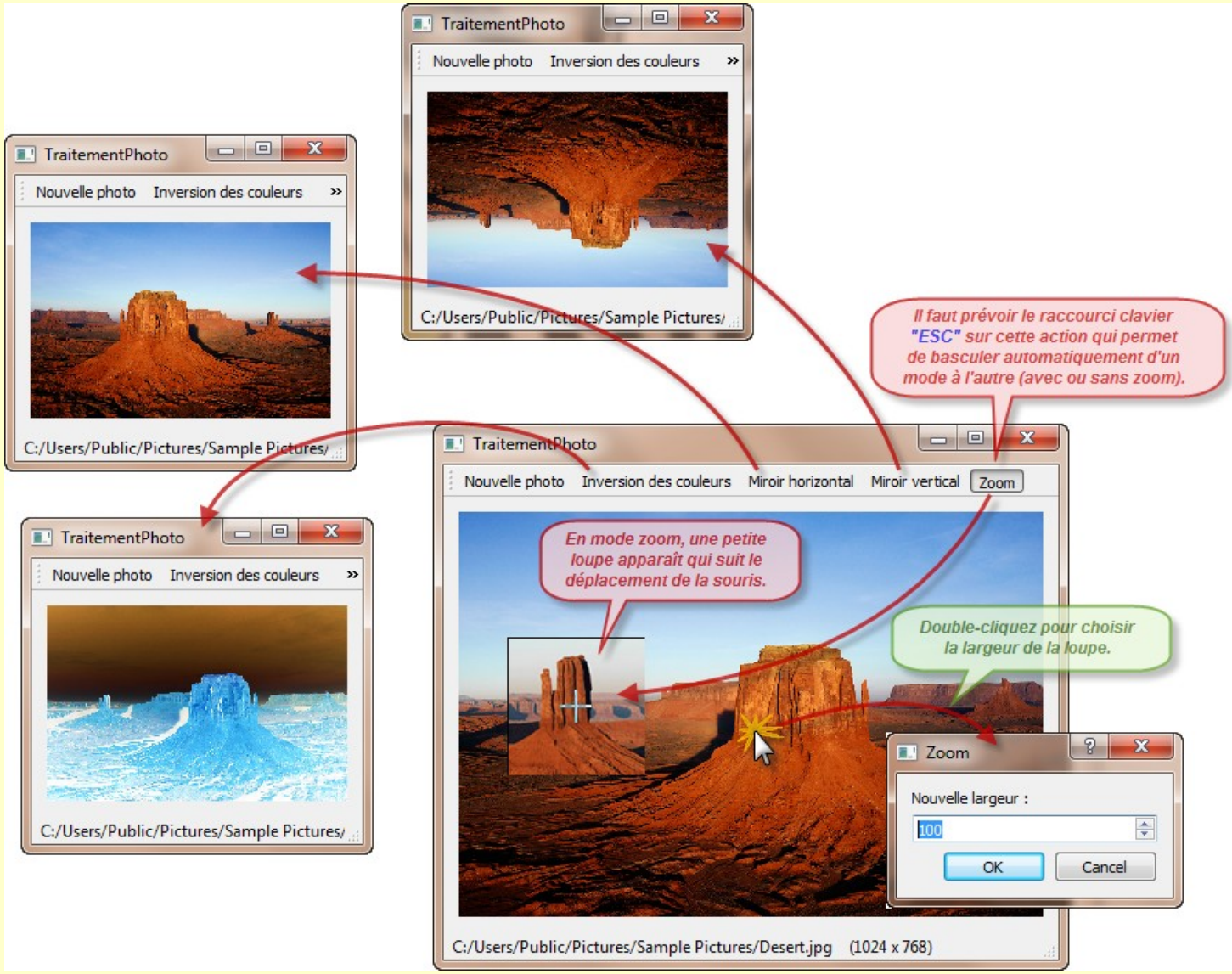


TRAVAUX PRATIQUES EN AUTONOMIE

Je vous propose de faire un nouveau projet capable également d'afficher des photos. Cette fois-ci, nous prévoyons de faire de tout petits traitements d'image, comme le mode négatif et le retournement des photos dans les sens horizontal et vertical. Par ailleurs, le mode zoom proposé est totalement différent puisqu'une loupe apparaît pour montrer un détail important de la photo à l'endroit de la souris.

Pour élaborer correctement ce projet, je vous propose de respecter les critères suivants :

- *Prenez la classe **QImage** plutôt que la classe **QPixmap** pour récupérer et visualiser les photos correspondantes.*
- *La classe **QImage** possède la méthode invertPixels() qui permet d'inverser les couleurs de tous les pixels constituant la photo. Utilisez l'aide de Qt Creator pour savoir comment l'utiliser correctement.*
- *La classe **QImage** possède également la méthode mirrored() qui permet faire un miroir dans le sens horizontal et/ou vertical de la photo en cours. Là aussi, ,utilisez l'aide de Qt Creator pour savoir comment l'utiliser correctement.*
- *Pour la loupe, il est nécessaire de prévoir une nouvelle image en taille réduite qui sera une copie partielle de la photo originale. Utilisez justement la méthode copy() de la classe **QImage** pour réaliser cette opération.*



*Pour vous aider dans votre recherche, je vous donne la déclaration de la classe **Photo** qui gère toutes ces situations, suivi de l'IHM correspondante.*

photo.h | <Selectionner un symbole>

```
1  #ifndef PHOTO_H
2  #define PHOTO_H
3
4  #include <QWidget>
5
6  class Photo : public QWidget
7  {
8    Q_OBJECT
9
10 public:
11   Photo(QWidget *parent = 0);
12 protected:
13   void paintEvent(QPaintEvent *evt);
14   void mouseMoveEvent(QMouseEvent *evt);
15   void mouseDoubleClickEvent(QMouseEvent *evt);
16 signals:
17   void envoyerMessage(QString message);
18 private slots:
19   void changerPhoto();
20   void inverserCouleur();
21   void miroirHorizontal();
22   void miroirVertical();
23   void modeZoom(bool activation);
24 private:
25   QImage photo, portion;
26   bool zoom;
27   QPoint localisation;
28   int largeur;
29 };
30
31 #endif // PHOTO_H
```