Source : T. VAIRA

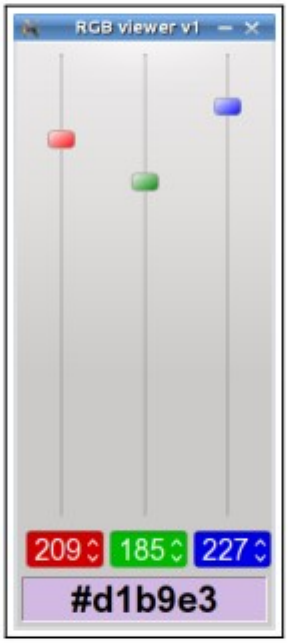
Objectifs

Les objectifs de ce TP sont :

- créer une application principale
- assurer sa mise en page ajoutant les composants QToolBars, QMenuBar et QStatusBar.
- assurer la traduction en français avc Qt Linguist
- intégrer la fonctionnalité du glisser-déposer (drag & drop)

Mise en situation

Dans ce TP, il s'agit de réaliser un programme permettant de régler les trois couleurs fondamentales afin d'obtenir une nouvelle couleur.



La version 1

On va développer l'application en 4 itérations :

- itération n°1 : réalisation de la GUI (*Graphical User Interface*)
- itération n°2 : ajout de la liste des couleurs connues du système et la conservation de 6 couleurs
- itération n°3 : ajout des fonctionnalités associées au menu, barre d'outils, ...
- itération n°4 : le glisser-déposer et la traduction

Remarque : un développement itératif s'organise en une série de développement très courts de durée fixe nommée itérations. Le résultat de chaque itération est un système partiel exécutable, testé et intégré (mais incomplet).

Travail demandé

Itération 1

Dans cette première itération, on s'attachera à créer sa propre application principale en créant une nouvelle classe **MyMainWindow** qui héritera de la classe **QMainWindow**. Une instance de cette classe représentera la fenêtre de l'application :

```
#include <QApplication>
#include <QtGui>

#include "MyMainWindow.h"

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

  MyMainWindow myMainWindow;

  myMainWindow.show();

  return app.exec();
}
```

Code 1: main.cpp

Le code source de cette classe sera réparti en deux fichiers : `MyMainWindow.h` et `MyMainWindow.cpp`.

Remarque : vous devez permettre la traduction (cf. itération 4) des chaînes affichées dans l'application en utilisant systématiquement la méthode `tr()`. On utilisera la langue anglaise pour la version de base.

On utilisera un widget personnalisé (`MyWidget`) comme **widget central**. On le définit en appelant `setCentralWidget()`.

La démarche pour la personnalisation de ce widget sera :

- Ajouter un `QSlider` vertical.
- Ajouter un `QSpinBox` au dessous. Adapter la font (Arial) et la taille des caractères (14 par exemple).
- Ajuster les valeurs maximales du slider et de la spinbox à 255.
- Placer les deux éléments dans un layout VBox.
- Pour redonner au slider sa largeur, régler dans sizePolicy le champ hSizeType sur Expanding en utilisant `setSizePolicy()`
- Répéter le même travail pour créer les trois colonnes nécessaires
- Placer les trois layouts VBox dans un layout HBox
- Créer et ajouter un label au-dessous pour la couleur
- Fixer sa police (Arial, 18, Bold), son alignement (`AlignHCenter | AlignVCenter`) et son aspect (`Panel | Sunken`)

On pourra personnaliser le style des sliders :

```
/* pour le slider rouge */
sliderR->setStyleSheet("QSlider::handle:vertical {
                background: qlineargradient(x1:0, y1:0, x2:1, y2:1, stop:0 white, stop:1 red);
                border: 1px solid #999999;
                border-radius: 5px;
            }");
```

Ensuite il vous faudra :

- Créer les 6 connections entre slider et spinbox (bidirectionnel) : le signal `valueChanged(int)` connecté sur le slot `setValue(int)`
- Créer trois slots au sein du widget : `RedAdjust(int value)`, `GreenAdjust(int value)` et `BlueAdjust(int value)`
- Associer les signaux `valueChanged(int)` des sliders avec les trois slots nouvellement créés

Créer une méthode privée nommée `RGBAdjust()`. Elle permettra d'ajuster la couleur de fond du label. Pour cela, il sera nécessaire de demander à chacun des sliders sa position (méthode `value()`). La valeur résultante de la couleur sous la forme hexadécimale précédée d'un dièse sera affichée à l'intérieur de ce label. Pour colorier un label, on pourra utiliser le code suivant :

```
QPalette palette;
labelCouleur->setAutoFillBackground(true);
/* QPalette::Window : A general background color */
palette.setColor(QPalette::Window, QColor(QColor(valueR->value(),valueG->value(),valueB->value())));
labelCouleur->setPalette(palette);
```

Terminer en créant une méthode privée `Init()` qui permet d'initialiser les valeurs (255) des 3 sliders. Elle sera appelée automatiquement par le constructeur.

L'instanciation des widgets et leur positionnement se fera dans le constructeur de la classe **`MyWidget`** (voir Code 5).

Question 1. Éditer la déclaration de la classe **`MyMainWindow`**.

```
#ifndef MY_MAIN_WINDOW_H
#define MY_MAIN_WINDOW_H

#include <QMainWindow>

class MyMainWindow : public QMainWindow
{
   public:
      MyMainWindow();
};

#endif
```

Code 2: MyMainWindow.h

Question 2. Éditer la définition de la classe `MyMainWindow`.

```
#include <QtGui>

#include "MyMainWindow.h"
#include "MyWidget.h"

MyMainWindow::MyMainWindow()
{
   MyWidget *centralWidget = new MyWidget;

   setCentralWidget(centralWidget);

   setWindowTitle("RGB viewer v1");
   setFixedSize(210, 480);
}
```

Code 3: MyMainWindow.cpp

Question 3. Compléter la déclaration de la classe `MyWidget`.

```
#ifndef MY_WIDGET_H
#define MY_WIDGET_H

#include <QWidget>
#include <QSlider>
#include <QSpinBox>
#include <QLabel>

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

   protected slots:
      /* les slots RedAdjust, GreenAdjust et BlueAdjust */

   private:
      /* les widgets et les mthodes prives */
};

#endif
```

Code 4: MyWidget.h

Question 4. Compléter la définition de la classe **MyWidget**.

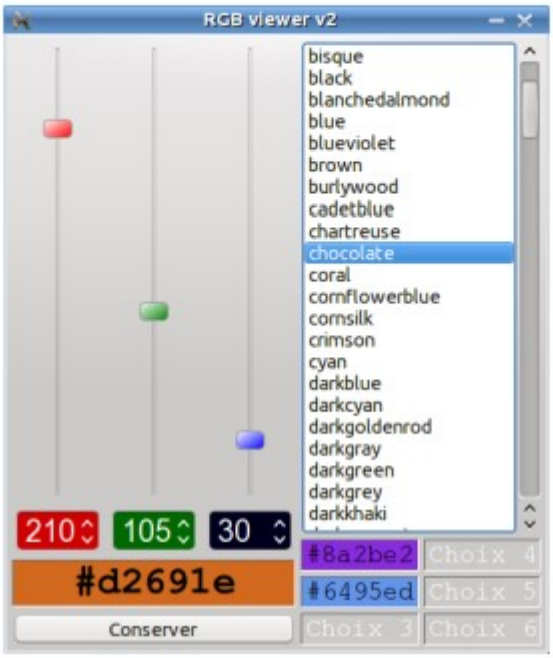
```
#include <QtGui>

#include "MyWidget.h"

MyWidget::MyWidget( QWidget *parent ) : QWidget( parent )
{
  /* TODO : cf. enonce */
  ...

  Init();
}

void MyWidget::Init()
{
   /* TODO : cf. enonce */
}

void MyWidget::RGBAdjust()
{
  /* TODO : cf. enonce */
}

void MyWidget::RedAdjust(int value)
{
   /* TODO : changer la couleur de fond du spinbox en tenant compte de la valeur de la
       composante rouge. Mettre la couleur des caracteres en blanc */

   RGBAdjust();
}

void MyWidget::GreenAdjust(int value)
{
   /* TODO : idem */

   RGBAdjust();
}

void MyWidget::BlueAdjust(int value)
{
   /* TODO : idem */

   RGBAdjust();
}
```

Code 5: MyWidget.cpp

Question 5. Fabriquer l'application version 1 et tester.

Itération 2

Cette version intègre deux nouvelles fonctionnalités : l'ajout de la liste des couleurs connues du système et la possibilité de conserver 6 couleurs.



Fonctionnalité n°1 :
On va commencer par l'ajout de la liste des couleurs connues du système.

```
#include <QtGui>

#include "MyWidget.h"

MyWidget::MyWidget( QWidget *parent ) : QWidget( parent )
{
  ...
  QStringList listeCouleurs = QColor::colorNames();
  QStringListModel *modeleCouleurs = new QStringListModel(listeCouleurs);
  QListView *vueCouleurs = new QListView;
  vueCouleurs->setModel(modeleCouleurs);
  ...
}
```

Code 6: MyWidget2.cpp

Lorsque l'utilisateur cliquera sur le nom d'une couleur, celle-ci devra s'afficher dans la partie gauche de l'application : composantes RGB et valeur en hexadécimale. Pour cela, il faudra tout d'abord connecter le signal `clicked(QModelIndex)` émis par `QListView` au slot `ColorChoice(QModelIndex)` de la classe `MyWidget`.

Dans la méthode `ColorChoice()`, on récupérera le nom de la couleur en utilisant `data()` de l'objet `QModelIndex` reçu en argument :

```
/* le slot ColorChoice */
void MyWidget::ColorChoice(QModelIndex model)
{
   QVariant nom = model.data(Qt::DisplayRole);
   ...
}
```

Code 7: ColorChoice()

Avec le nom de la couleur ainsi obtenue, on créera un objet de type `QColor`. On récupérera les composantes avec `red()`, `green()` et `blue()` pour les passer ensuite aux sliders.

Fonctionnalité n°2 :

On décide de pouvoir conserver 6 couleurs personnalisées. Pour cela, on va ajouter un bouton "Conserver" et 6 QLabel comme indiqué ici :

Lorsque l'utilisateur clique sur le bouton "Conserver", la couleur courante se place dans le premier choix disponible. On décide de gérer les 6 couleurs conservées comme un tampon circulaire : on placera la couleur dans le premier `QLabel` libre sinon on reviendra à la première position.

On va ajouter un nouveau slot à la classe `MyWidget` : `ColorKeep()`. Il sera déclenché sur le signal `clicked()` du bouton "Conserver".

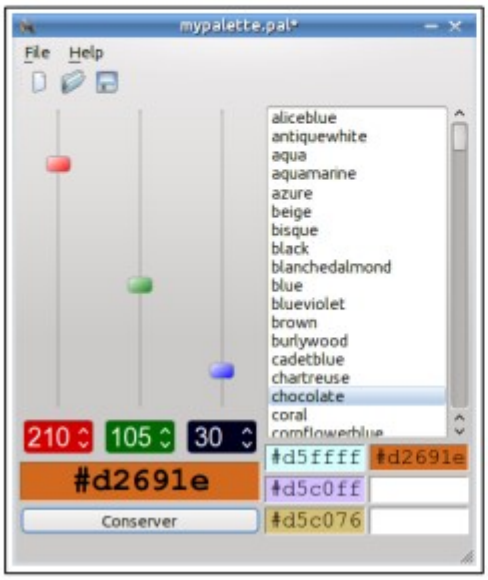
Question 6. Ajouter la liste des couleurs connues du système et assurer son positionnement et son fonctionnement dans l'application.

Question 7. Ajouter le bouton et les 6 QLabel et coder son fonctionnement pour l'application.

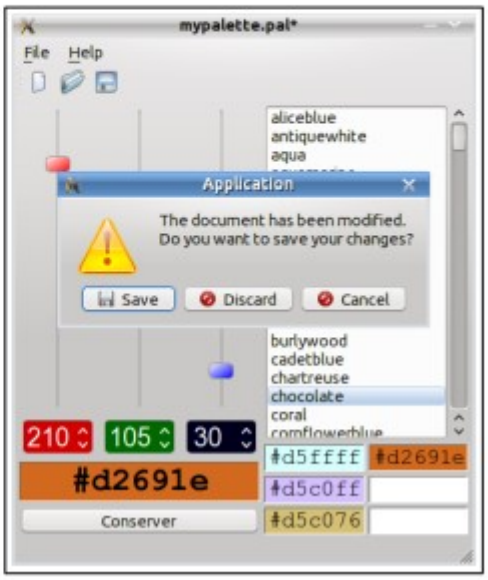
Question 8. Fabriquer l'application version 2 et tester.

Itération 3

On va maintenant doter le programme des fonctionnalités de base d'une application standard. Cela permettra à l'utilisateur de sauvegarder ses couleurs personnalisées dans des fichiers (palettes) et de les recharger afin de les modifier éventuellement.



L'utilisateur a créé quelques couleurs personnelles ...



Puis quand il essaye de quitter l'application ...

Modification n°1 : un nouveau `QLabel`

Il s'avère nécessaire de créer notre propre `QLabel`. Pour cela, on va créer une nouvelle classe **`MyLabel`** qui hérite de `QLabel`. Cette classe offrira les services dont on a besoin pour notre application : fixer la couleur de fond du `QLabel` et conserver la valeur de cette couleur en hexadécimale.


```
#ifndef MY_LABEL_H
#define MY_LABEL_H

#include <QtGui>

class MyLabel : public QLabel
{
    Q_OBJECT
    private:
        QString _color; /* la valeur de la couleur sous forme hexadecimale */
```

```
    public:
        MyLabel(QWidget * parent = 0);

        void clear(); /* efface le contenu du "QLabel" */
        QString color(); /* retourne la valeur de la couleur */
        void setColor(int r, int g, int b); /* fixe le contenu du "QLabel" */
        void setColor(QColor color); /* fixe le contenu du "QLabel" */
};

#endif
```

Code 8: MyLabel.h

L'intégration de cette classe va modifier le comportement de la classe `MyWidget`.

Question 9. Intégrer la classe `MyLabel` en assurant les modifications nécessaires dans la classe `MyWidget` et tester.

Modification n°2 : la classe `MyMainWindow`

Notre programme doit fonctionner sur l'exemple fourni par Qt (qt-project.org/doc/mainwindows-application.html). Cet exemple montre comment implémenter une application standard GUI avec menus, barres d'outils et une barre d'état. L'intégration de ces fonctionnalités va modifier en profondeur les classes `MyMainWindow` et `MyWidget`.

Par conséquence, le constructeur de la classe `MyMainWindow` devient :

```
#include <QtGui>

#include "MyMainWindow.h"
#include "MyWidget.h"

MyMainWindow::MyMainWindow()
{
   centralWidget = new MyWidget;

   setCentralWidget(centralWidget);

   createActions();
   createMenus();
   createToolBars();
   createStatusBar();

   connect(centralWidget, SIGNAL(contentsChanged()), this, SLOT(documentWasModified()));

   setCurrentFile("");
   setUnifiedTitleAndToolBarOnMac(true);

   //setWindowTitle("untitled.pal[*] - RGB viewer v3");
   setFixedSize(420, 480);
}
...
```

Code 9: MyMainWindow3.cpp

En reprenant le code fourni par Qt, il vous faudra l'adapter à vos besoins.

La méthode `createActions()` :

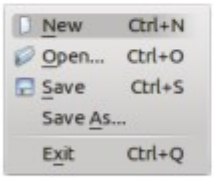
La classe `QAction` fournit une interface abstraite pour décrire une action (= commande) qui peut être insérée dans les widgets. Dans de nombreuses applications, des commandes communes peuvent être invoquées via des menus, boutons, et des raccourcis clavier. Puisque l'utilisateur s'attend à ce que chaque commande soit exécutée de la même manière, indépendamment de l'interface utilisateur utilisée, il est utile de représenter chaque commande comme une action. Les actions peuvent être ajoutés aux menus et barres d'outils, et seront automatiquement synchronisées.

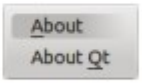
Dans cette méthode, on créera les actions pour **New**, **Open**, **Save**, **Save As**, **Exit**, **About** et **About Qt**. On utilisera la méthode `about()` de la classe `QMessageBox` pour afficher la boîte de dialogue About.

La méthode `createMenus()` :

La classe `QMenu` fournit un widget pour une utilisation dans les barres de menus et les menus contextuels. Un menu contextuel est un menu qui s'affiche lorsqu'on fait un clic droit sur un widget. Un widget menu est un menu de sélection. Il peut être soit un menu déroulant dans une barre de menu ou un menu contextuel autonome. Les menus déroulants sont indiquées par la barre de menu lorsque l'utilisateur clique sur l'élément concerné ou appuie sur la touche de raccourci spécifié. Qt implémente donc les menus avec `QMenu` et `QMainWindow` les garde dans un `QMenuBar`. On utilise la méthode `addMenu()` de `QMenuBar` pour insérer un menu dans une barre de menu. La classe `QMenuBar` fournit une barre de menu horizontale. Une barre de menu se compose d'une liste d'éléments de menu déroulant.

Dans cette méthode, on créera deux menus déroulants : **File** (pour les actions New, Open, Save, Save As et Exit) et **Help** (pour les actions About et About Qt).

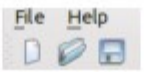




La méthode `createToolBars()` :

La classe `QToolBar` fournit une barre d'outils qui contient un ensemble de contrôles (généralement des icônes) et située sous les menus. Pour ajouter une barre d'outils, on doit tout d'abord appeler la méthode `addToolBar()` de `QMainWindow`. Avec Qt, la barre d'outils utilise des actions pour construire chacun des éléments de celle-ci. Les boutons de la barre d'outils sont donc insérés en ajoutant des actions et en utilisant `addAction()` ou `insertAction()`. Les boutons peuvent être séparés en utilisant `addSeparator()` ou `insertSeparator()`. Quand un bouton de la barre est enfoncée, il émet le signal `actionTriggered()`.

Dans cette méthode, on créera une barre d'outils pour **File** (avec les actions New, Open et Save).



La méthode createStatusBar() :

La classe `QStatusBar` fournit une barre horizontale appropriée pour la présentation des informations d'état. `QStatusBar` permet d'afficher différents types d'indicateurs. Une barre d'état peut afficher trois types de messages différents : temporaire (affiché brièvement), normal (affiché tout le temps, sauf quand un message temporaire est affiché) et permanent (jamais caché). La barre d'état peut être récupéré à l'aide de la méthode `statusBar()` de `QMainWindow` et remplacé à l'aide de `setStatusBar()`.

Dans cette méthode, on fixera le message normal 'Ready'. On utilisera aussi la barre d'état pour afficher deux messages temporaires pendant 2s : 'File loaded' (au chargement d'un fichier) et 'File saved' (pendant l'enregistrement d'un fichier).

On modifie la déclaration de la classe `MyMainWindow` qui devient :

```
#ifndef MY_MAIN_WINDOW_H
#define MY_MAIN_WINDOW_H

#include <QMainWindow>

class MyWidget;

class MyMainWindow : public QMainWindow
{
   Q_OBJECT
   public:
      MyMainWindow();

   protected:
      void closeEvent(QCloseEvent *event);

   private slots:
      void newFile();
      void open();
      bool save();
      bool saveAs();
      void about();
      void documentWasModified();

   private:
      void createActions();
      void createMenus();
      void createToolBars();
      void createStatusBar();
      bool maybeSave();
      void loadFile(const QString &fileName);
      bool saveFile(const QString &fileName);
      void setCurrentFile(const QString &fileName);
      QString strippedName(const QString &fullFileName);

      MyWidget *centralWidget;
      QString curFile;
      QMenu *fileMenu;
      QMenu *helpMenu;
      QToolBar *fileToolBar;
      QAction *newAct;
```

```
    QAction *openAct;
    QAction *saveAct;
    QAction *saveAsAct;
    QAction *exitAct;
    QAction *aboutAct;
    QAction *aboutQtAct;
};

#endif
```

Code 10: MyMainWindow3.h

Modification n°3 : la classe MyWidget

Pour la communication entre la classe `MyMainWindow` et le widget central (`MyWidget`), on conservera les mêmes noms de méthodes fournies dans l'exemple de Qt, soit :

```
centralWidget->Clear() /* permettra d'effacer le contenu de la GUI */

centralWidget->setModified() /* permettra de fixer l'etat courant de l'application */
centralWidget->isModified() /* permettra de retourner l'etat courant de l'application pour
    savoir si une modification (a enregistrer) est en cours */

centralWidget->setPlainText() /* permettra de passer le contenu du fichier ouvert */
centralWidget->toPlainText() /* permettra de recuperer les donnees a enregistrer */
```

Code 11: Communication MyMainWindow et MyWidget

La méthode `Clear()` remettra le widget central dont son état initial et donc les couleurs personnalisées seront effacées.

Pour gérer l'état courant de l'application, on ajoutera un attribut `modified` à la classe `MyWidget`. Les accesseurs de cet attribut seront donc : `isModified()` et `setModified()`.

Il reste à gérer le fichier qui stockera les couleurs créées. On utilisera des fichiers de type "texte" avec l'extension **.pal**. On enregistrera tout simplement la valeur en hexadécimale de chaque couleur conservée de la manière suivante :

```
#d5ffff
#d5c0ff
#d5c076
```

Remarque : le saut de ligne sera notre délimiteur entre chaque couleur.

Dans la méthode `setPlainText()`, on recevra les données sous la forme d'un `QString`. Pour recharger les couleurs personnalisées contenues dans ce `QString`, le plus simple est de les gérer à partir d'une `QStringList` (et de les avoir découpées grâce au saut de ligne) :

```
void MyWidget::setPlainText(QString datas)
{
   QStringList listeCouleurs;
   listeCouleurs = datas.split("\n");
   //qDebug() << "setPlainText : " << datas;
   //for (int i = 0; i < listeCouleurs.size(); ++i)
      //qDebug() << i << " : " << listeCouleurs.at(i).toLocal8Bit().constData();
```

```
    Clear(); /* on efface tout */
    ...
}
```

Code 12: setPlainText()

Pour finir, la méthode `toPlainText()` s'assurera de retourner un QString contenant la liste des couleurs personnalisées séparées par un saut de ligne ('\n').

On modifie la déclaration de la classe `MyWidget` qui devient alors :

```
#ifndef MY_WIDGET_H
#define MY_WIDGET_H

#include <QtGui>

class MyLabel;

class MyWidget : public QWidget
{
   Q_OBJECT

   public:
      MyWidget(QWidget *parent=0);

      void Clear();
      bool isModified();
      void setPlainText(QString datas);
      QString toPlainText();
      void setModified(bool modified);

   signals:
      void contentsChanged();

   protected slots:
      void RedAdjust(int value);
      void GreenAdjust(int value);
      void BlueAdjust(int value);
      void ColorChoice(QModelIndex model);
      void ColorKeep();

   private:
      /* les membres privees */

      bool modified;

      void Init();
      void RGBAdjust();
};

#endif
```

Code 13: MyWidget3.h

Question 10. Modifier les classes `MyMainWindow` et `MyWidget` en tenant compte des fonctionnalités demandées.

Question 11. Fabriquer l'application version 3 et tester.

Itération 4

Dans cette dernière itération, on va améliorer l'interactivité avec l'utilisateur. Pour cela, on va tout d'abord retirer le bouton "Conserver". Maintenant l'utilisateur pourra :

- double-cliquer sur la couleur créée pour la conserver ou
- la glisser-déposer dans l'emplacement de son choix

D'autre part, l'utilisateur pourra modifier une couleur personnalisée par un simple clic sur la couleur de son choix. Le glisser-déposer fonctionnera aussi entre couleurs personnalisées (comme un copier/coller).

L'action de glisser-déposer

L'utilisateur a cliqué sur la couleur #d2691e pour la modifier

Le widget `QLabel`, et donc notre classe `MyLabel`, ne possède pas de signal `clicked()` (comme les `QPushButton` par exemple), ni de signal `doubleClicked()`. Pour recevoir des événements de la souris dans ses propres widgets, il suffit de réimplémenter les gestionnaires d'évènements (*event handler*). On va donc émettre ces signaux à partir des gestionnaires `mousePressEvent()` et `mouseDoubleClickEvent()`.

La redéfinition de la méthode `mouseDoubleClickEvent()` pour simuler le signal `doubleClicked()` sera :

```
void MyLabel::mouseDoubleClickEvent(QMouseEvent *event)
{
    if(event->button() == Qt::LeftButton)
    {
        emit doubleClicked();
    }
}
```

Code 14: mouseDoubleClickEvent()

On opère de la même façon pour le signal `clicked()` :

```
void MyLabel::mousePressEvent(QMouseEvent *event)
{
    if(event->button() == Qt::LeftButton)
    {
       dragStartPosition = event->pos(); /* on enregistre la position de depart pour un
          eventuel glisser-deposer */
       emit clicked(_numero); /* on indique le numero du QLabel qui emet le clic */
    }
}
```

Code 15: mousePressEvent()

Le 'glisser-déposer' (drag & drop) est un mécanisme visuel simple qui permettra à l'utilisateur de transférer des informations entre objets `MyLabel` au sein de l'application. Ici, le glisser-déposer ne concerne que des objets `MyLabel` existants et seul l'échange de l'information de couleur suffit pour implanter la fonctionnalité demandée.

Dans notre situation, il suffira de redéfinir les gestionnaires d'évènements `dragEnterEvent()` et `dropEvent(`

Pour intégrer ces nouvelles fonctionnalités, on va modifier la déclaration de la classe `MyLabel` :

```
#ifndef MY_LABEL_H
#define MY_LABEL_H

#include <QtGui>

class MyLabel : public QLabel
{
   Q_OBJECT
   private:
      QString _color;
      int _numero; /* permettra d'identifier le QLabel parmi d'autres */
      QPoint dragStartPosition;

   public:
      MyLabel(int numero=0, QWidget *parent = 0);
      void clear();
      QString color();
      void setColor(int r, int g, int b);
      void setColor(QColor color);

      void mousePressEvent(QMouseEvent *event);
      void mouseDoubleClickEvent(QMouseEvent *event);
      void mouseMoveEvent(QMouseEvent *event);
      void dragEnterEvent(QDragEnterEvent *event);
      void dropEvent(QDropEvent *event);

   signals:
      void clicked(int);
      void doubleClicked();
      void changed(int);
};

#endif
```

Code 16: MyLabel4.h

Il faudra aussi réimplémenter `mouseMoveEvent()` pour déterminer si un glisser devrait commencer. Ceci est nécessaire pour distinguer l'opération de glisser-déposer d'un simple clic sur le label.

Le widget receveur `MyLabel` devra accepter le déposer (*drop*) en appelant la méthode `setAcceptDrops(true)` que l'on placera dans le constructeur de le classe `MyLabel`.

Les redéfinitions des gestionnaires d'évènement seront les suivantes :

```
void MyLabel::mouseMoveEvent(QMouseEvent *event)
{
   if (!(event->buttons() & Qt::LeftButton))
      return;
   if ((event->pos() - dragStartPosition).manhattanLength()
         < QApplication::startDragDistance())
      return;

   /* on demarre une operation de glisser-deposer */
   QDrag *drag = new QDrag(this);
   QMimeData *mimeData = new QMimeData;
   /* dessine un rectangle de couleur a afficher lors du deplacement */
   QPixmap pixmap(this->width(), this->height());
   QPainter painter;
   painter.begin(&pixmap);
   painter.fillRect(pixmap.rect(), QColor(_color));
   painter.setPen( QPen(Qt::black, 4) );
   painter.setFont(QFont("Courier New", 12, QFont::Bold));
   painter.drawText(pixmap.rect(), Qt::AlignCenter, _color);
   painter.end();
   drag->setPixmap(pixmap);

   if(!_color.isEmpty())
   {
      /* l'information trasnferee sera la valeur de la couleur en hexadecimale */
      mimeData->setText(_color);
      drag->setMimeData(mimeData);
      drag->exec(Qt::CopyAction); /* c'est parti ! */
   }
}

void MyLabel::dragEnterEvent(QDragEnterEvent *event)
{
   if (event->mimeData()->hasFormat("text/plain"))
   {
      event->acceptProposedAction();
   }
}

void MyLabel::dropEvent(QDropEvent *event)
{
   /* on modifie la couleur du MyLabel cible */
   this->setColor(QColor(event->mimeData()->text()));
   emit changed(_numero); /* il y a eu un changement : on le signale */
   event->acceptProposedAction();
}
```

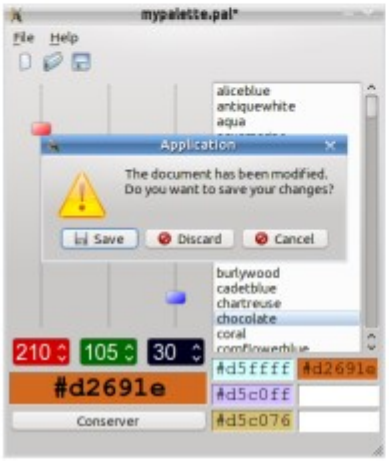
Code 17: Le glisser-déposer

Ces modifications entraînent évidemment des changements pour le `widget central`. La fonction (`ColorKeep`), assurée précédemment par le bouton "Conserver", est maintenant associée à un double clic sur la couleur créée. Il faudra donc modifier la connexion signal/slot.

D'autre part, les 6 `MyLabel` pour les couleurs personnalisées sont capables d'émettre deux signaux à connecter : `clicked(int)` et `changed(int)`. On créera deux nouveaux slots : `ColorEdit(int)` et `ColorChanged(int)`. Le paramètre `int` passé en argument permettra d'identifier chaque label par son numéro.

Question 12. Coder les nouvelles fonctionnalités d'interactivité et tester.

Pour finaliser l'application, on assurera sa traduction en français en utilisant **Qt Linguist**.



La version 3 dans sa langue initiale

La version 4 après traduction

La première modification à apporter se trouve dans le fichier de projet (`.pro`). Il faut ajouter cette ligne :

```
TRANSLATIONS = rgbviewer4_fr.ts
```

Puis vous devez créer le fichier `.ts` en utilisant l'outil `lupdate` :

```
$ lupdate -verbose rgbviewer4.pro
```

Ensuite, vous traduirez chaque chaîne avec l'outil `linguist` :

```
$ linguist rgbviewer4_fr.ts
```

Une fois la traduction réalisée, il vous faut générer le fichier binaire `.qm` en utilisant l'outil `lrelease` :

```
$ lrelease rgbviewer4_fr.ts
```

Il ne faut pas oublier d'intégrer un objet de type `QTranslator` dans le main pour charger le fichier de traduction :

```
#include <QApplication>
#include <QtGui>
#include <QTranslator>
```

```cpp
#include "MyMainWindow.h"

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

  QTranslator translator;
  translator.load("rgbviewer4_fr");
  app.installTranslator(&translator);

  MyMainWindow myMainWindow;

  myMainWindow.show();

  return app.exec();
}
```

Code 18: main4.cpp

Question 13. Assurer la traduction en français de l'application.

Question 14. Fabriquer l'application version 4 et tester.