Developpez.com
Club des développeurs

Implémentation du modèle MVC

Qt Quarterly

par Qt Quarterly (La liste complète)

Date de publication : 23 mai 2009

Dernière mise à jour :

Qt Quarterly est un journal électronique disponible exclusivement aux clients Qt. Chaque trimestre, nous envoyons un e-mail qui, nous l'espérons, ajoutera à votre expérience Qt, avec des articles de qualité écrits par des experts de Qt.

Developpez.com a reçu l'autorisation de Nokia afin de traduire ces articles.

I - Implémentation du modèle MVC.......... 3

Copyright © 2009 - Kinji1. Aucune reproduction, même partielle, ne peut être faite de ce site et de l'ensemble de son contenu : textes, documents, images, etc sans l'autorisation expresse de l'auteur. Sinon vous encourez selon la loi jusqu'à 3 ans de prison et jusqu'à 300 000 E de dommages et intérêts.

I - Implémentation du modèle MVC

par Jarek Kobus

Qt4 utilise le modèle MVC pour ses classes d'éléments visuels tels que QListView, QTable, etc. Mais MVC est plus qu'un simple modèle pour gérer les éléments visuels, il peut aussi être un moyen de synchroniser différents widgets. Dans cette article, nous montrons comment appliquer ce principe, en tirant pleinement parti du méchanisme signal-slot de Qt.

Dans l'article "Une table modèle/vue pour de grands ensembles de données" [NOTE: titre original 'A Model/View Table for Large Datasets', de plus ça force le titre de cette QQ, donc à décider], nous avons vu comment créer une classe dérivée de QTable implémentant le modèle et la vue. Nous abordons ici une approche plus générique qui peut être appliquée à n'importe lequel des widgets Qt (et à nos propres classes de widgets).

Un modèle est un ensemble de données, et une vue est un composant de l'IHM qui permet une représentation visuelle du modèle pour l'utilisateur. Si le modèle (les données) ne peuvent pas être modifiées par l'utilisateur, un modèle et une vue est suffisant. Mais si le modèle peut être modifié, il nous faut alors un controlleur, qui donne à l'utilisateur la possibilité de modifier les données présentées dans le vue, et dont les changements sont répercutés dans la source de données.

Imaginons que nous voulons permettre la gestion des couleurs dans notre application. Nous désirons fournir une palette de couleurs ue l'utilisateur peut utiliser au sein de l'application, par exemple pour spécifier la couleur du texte ou la couleur des formes déssinées. Nous pourrions vouloir présenter la palette de différentes manières, à différents endroits de l'application, mais nous voulons que les les couleurs utilisées appartiennent à une unique palette.

Dans cet exemple, le modèle est la palette de couleurs, et la vue un widget qui peut afficher ses couleurs. Le controlleur pourrait être un "widget d'édition" séparé ou pourrait être associé au widget de la vue. Dès que les données du modèle changent, par exemple parce qu'un utilisateur a édité les données dans l'une des vues, toutes les vues actives doivent en être informées pour qu'elles se mettent à jour elles-même. Pour cet exemple, nous nous contenterons de réaliser qu'un seul widget, mais nous pourrions en faire autant que l'on désire, chacun présentant les données à sa façon.

Notre modèle utilise le modèle singleton, puisque nous ne voulons qu'une unique palette qui sera utilisée dans l'application entière. Regardons le définition de la classe modèle de notre palette.

```
class PaletteModelManager : public QObject
    {
        Q_OBJECT

    public:
         PaletteModelManager();

        static PaletteModelManager *getInstance();

        QMap<QString, QColor> getPalette() const { return thePalette; }
        QColor getColor(const QString &id) const;

    public slots:
        QString addColor(const QString &id, const QColor &color);
        void changeColor(const QString &id, const QColor &newColor);
        void removeColor(const QString &id);

    signals:
        void colorAdded(const QString &id);
        void colorChanged(const QString &id, const QColor &color);
        void colorRemoved(const QString &id);

    private:
        PaletteModelManager(QObject *parent = 0, const char *name = 0)
            : QObject(parent, name) {}
```


Copyright © 2009 - Kinji1. Aucune reproduction, même partielle, ne peut être faite de ce site et de l'ensemble de son contenu : textes, documents, images, etc sans l'autorisation expresse de l'auteur. Sinon vous encourez selon la loi jusqu'à 3 ans de prison et jusqu'à 300 000 E de dommages et intérêts.

```
        QMap<QString, QColor> thePalette;
        static PaletteModelManager *theManager;
    };
```

La classe PaletteModelManager est assez inhabituelle. Tout d'abord elle fournit une fonction statique getInstance() qui retourne un pointer vers le seul et unique objet PaletteModelManager qui peut exister dans l'application. Ensuite, elle a un constructeur privé, ce qui assure que les utilisateur ne pourront pas instancier la classe elle-même. Cette deux fonctionnalités sont utilisées pour implémenter un singleton en C++.

La palette en elle-même est une simple association de chaînes (ID) vers des couleurs. Le slot addColor() possède quant à lui une valeur de retour qui n'est pas de type void afin d'être utilisé à la fois comme slot mais aussi comme fonction.

La classe fournit plusieurs interfaces clés. Une interface d'accès (getInstance()) qui nous donne un pointeur à partir duquel nous pouvons interagir avec le modèle. Une interface de lecture (getPalette()) avec laquelle nous pouvons lire l'état actuel du modèle. Une interface de modification (via les slots) qui permet de modifier le modèle. Et une interface d'information (via les signaux) qui notifie les changement du modèle.

```
PaletteModelManager *PaletteModelManager::theManager = 0;

    PaletteModelManager *PaletteModelManager::getInstance()
    {
        if (!theManager)
            theManager = new PaletteModelManager();
        return theManager;
    }

    PaletteModelManager:: PaletteModelManager()
    {
        if (theManager == this)
            theManager = 0;
    }
```

Le pointeur global PaletteModelManager est initailisé statiquement à 0. Il pourrait être tentant d'utiliser un obket statique plutôt qu'un pointeur, mais certains compilateurs n'appelent pas les constructeurs des objets statiques, surtout dans les bibliothèques, donc notre approche est plus robuste. Dans getInstance() nous construisons l'unique instance si elle n'existe pas encore.

Nous avons omis l'implémentation de tous les slots excepté changeColor() :

```
void PaletteModelManager::changeColor(const QString &id, const QColor &newColor)
    {
        if (!thePalette.contains(id) || thePalette[id] == newColor)
            return;
        emit colorChanged(id, newColor);
        thePalette[id] = newColor;
    }
```

Ici nous émettons le signal colorChanged() *avant* de réaliser la modification afin que la palette soit dans son état original dans tous les slots connectés au signal colorChanged(), avec l'ID et la couleur qui vont devenir actifs en paramètres. Ceci est utile si vous voulez suivre les changement, par exemple pour supporter un historique des actions. Parfois il peut être plus approprié d'émettre le signal *après* la modification (comme nous le faisons dans PaletteModelManager::addColor()), mais dans de tels cas l'état précédent ne peut pas être obtenus directement du gestionnaire de palette, donc si celui-ci est nécessaire il doit être passé en paramètre du signalqui notifie la modification. Une dernière stratgie serait d'émettre des signaux avant et après un changement d'état.

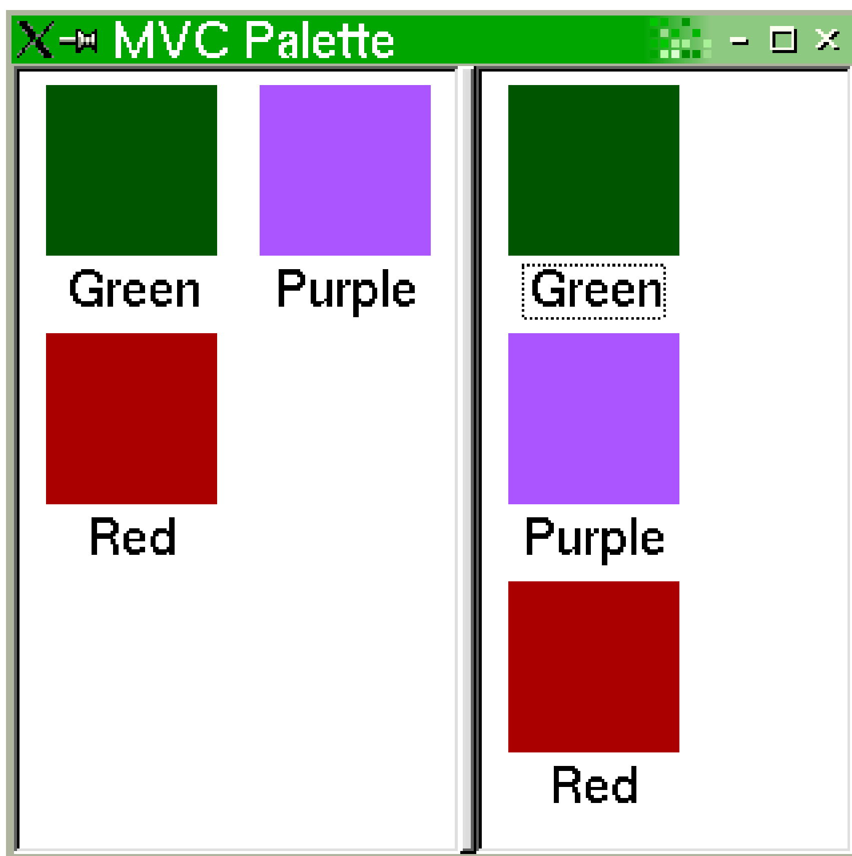
Maintenant que nous vu comment implémenter notre modèle, voyons comment s'en servir. Nous allons créer une vue d'icones personnalisée qui présentera les couleurs et leurs ID. Voici sa définition :

```
class PaletteIconView : public QIconView
    {
```


Copyright © 2009 - Kinji1. Aucune reproduction, même partielle, ne peut être faite de ce site et de l'ensemble de son contenu : textes, documents, images, etc sans l'autorisation expresse de l'auteur. Sinon vous encourez selon la loi jusqu'à 3 ans de prison et jusqu'à 300 000 E de dommages et intérêts.

```
    Q_OBJECT

public:
    PaletteIconView(QWidget *parent = 0, const char *name = 0);
    PaletteIconView() {}

    void setPaletteModelManager(PaletteModelManager *manager);

private slots:
    void colorAdded(const QString &id);
    void colorChanged(const QString &id, const QColor &newColor);
    void colorRemoved(const QString &id);

    void contextMenuRequested(QIconViewItem *item, const QPoint &pos);

private:
    void clearOld();
    void fillNew();
    QPixmap getColorPixmap(const QColor &color) const;

    PaletteModelManager *theManager;
    QMap<QString, QIconViewItem *> itemFromColorId;
    QMap<QIconViewItem *, QString> colorIdFromItem;
};
```

Notre vue personnalisée stocke un pointeur sur le PaletteModelManager. Puisque celui-ci est unique, nous aurions pu simplement utiliser la fonction statique getInstance(), mais nous avons préféré une approche plus générale, puisque la plupart des modèles ne sont pas implémentés comme des singletons. Les slots privés sont utilisés pour mettre à jour la vue et le gestionnaire de palette. Notre controlleur est embarqué avec notre vue, ici en tant que menu contextuel.



Nous allons maintenant regarder les fonctions principales de PaletteIconView

```
PaletteIconView::PaletteIconView(QWidget *parent, const char *name)
        : QIconView(parent, name), theManager(0)
    {
```


Copyright © 2009 - Kinji1. Aucune reproduction, même partielle, ne peut être faite de ce site et de l'ensemble de son contenu : textes, documents, images, etc sans l'autorisation expresse de l'auteur. Sinon vous encourez selon la loi jusqu'à 3 ans de prison et jusqu'à 300 000 E de dommages et intérêts.

```
        setPaletteModelManager(PaletteModelManager::getInstance());
        connect(this, SIGNAL(contextMenuRequested(QIconViewItem*, const QPoint&)),
                this, SLOT(contextMenuRequested(QIconViewItem*, const QPoint&)));
    }
```

Le constructeur est assez limpide, nous sauvegardons le pointeur sur le PaletteModelManager et nous connectons lemenu contextuel.

```
void PaletteIconView::setPaletteModelManager(PaletteModelManager *manager)
    {
        if (theManager == manager)
            return;
        if (theManager) {
            disconnect(theManager, SIGNAL(colorAdded(const QString&)),
                       this, SLOT(colorAdded(const QString&)));
            disconnect(theManager, SIGNAL(colorChanged(const QString&, const QColor&)),
                       this, SLOT(colorChanged(const QString&, const QColor&)));
            disconnect(theManager, SIGNAL(colorRemoved(const QString&)),
                       this, SLOT(colorRemoved(const QString&)));
            clearOld();
        }
        theManager = manager;
        if (theManager) {
            fillNew();
            connect(theManager, SIGNAL(colorAdded(const QString&)),
                    this, SLOT(colorAdded(const QString&)));
            connect(theManager, SIGNAL(colorChanged(const QString&, const QColor&)),
                    this, SLOT(colorChanged(const QString&, const QColor&)));
            connect(theManager, SIGNAL(colorRemoved(const QString&)),
                    this, SLOT(colorRemoved(const QString&)));
        }
    }
```

Lorsqu'un nouveau gestionnaire de palette est utilisé, les connexions vers l'ancien sont rompues (s'il y en avait) et de nouvelles sont établies. Nous n'avons pas montré la fonction clearOld(), elle nettoie les associations item - couleur et nettoie aussi la vue.

```
void PaletteIconView::fillNew()
    {
        QMap<QString, QColor> palette = theManager->getPalette();
        QMap<QString, QColor>::const_iterator i = palette.constBegin();
        while (i != palette.constEnd()) {
            colorAdded(i.key());
            ++i;
        }
    }
```

La fonction fillNew() construit les associations avec les ID et les couleurs de la palette et ajoute chaque couleur dans la vue.

```
void PaletteIconView::colorAdded(const QString &id)
    {
        QIconViewItem *item = new QIconViewItem(this, id,
                                      getColorPixmap(theManager->getColor(id)));
        itemFromColorId[id] = item;
        colorIdFromItem[item] = id;
    }
```

Lorsque l'utilisateur ajoute une nouvelle couleur via le menu contextuel, nous créons un nouvel item dans notre vue et mettons à jour l'association item - ID.

```
void PaletteIconView::contextMenuRequested(QIconViewItem *item, const QPoint &pos)
    {
        if (!theManager)
            return;
        QPopupMenu menu(this);
```


Copyright © 2009 - Kinji1. Aucune reproduction, même partielle, ne peut être faite de ce site et de l'ensemble de son contenu : textes, documents, images, etc sans l'autorisation expresse de l'auteur. Sinon vous encourez selon la loi jusqu'à 3 ans de prison et jusqu'à 300 000 E de dommages et intérêts.

```
        int idAdd = menu.insertItem(tr("Add Color"));
        int idChange = menu.insertItem(tr("Change Color"));
        int idRemove = menu.insertItem(tr("Remove Color"));
        if (!item) {
            menu.setItemEnabled(idChange, false);
            menu.setItemEnabled(idRemove, false);
        }
        int result = menu.exec(pos);
        if (result == idAdd) {
            QColor newColor = QColorDialog::getColor();
            if (newColor.isValid()) {
                QString name = QInputDialog::getText(tr("MVC Palette"), tr("Color Name"));
                if (!name.isEmpty())
                    theManager->addColor(name, newColor);
            }
        }
        else if (result == idChange) {
            QString colorId = colorIdFromItem[item];
            QColor old = theManager->getColor(colorId);
            QColor newColor = QColorDialog::getColor(old);
            if (newColor.isValid())
                theManager->changeColor(colorId, newColor);
        }
        else if (result == idRemove) {
            QString colorId = colorIdFromItem[item];
            theManager->removeColor(colorId);
        }
    }
```

Le menu contextuel est très simple. Tout d'abord nous vérifions qu'il y a bien un gestionnaire de palette, sans lequel nous ne pouvons rien faire. Ensuite nous créons les éléments du menu, mais désactivons ceux qui ne s'appliquent seulement qu'aux éléments si l'utilisateur n'a pas invoquer le menu depuis l'un d'eux (si item == 0). Si l'utilisateur choisit "Add", nous affichons une boite de dialogue et s'il choisit une couleur, nous affichons une autre boîte pour obtenir l'ID de la couleur. S'il choisit "Change", nous affichons une boîte qui propose de choisir la nouvelle couleur, et s'il choisit "Remove" nous retirons la couleur.

Notez que l'ajout, la modification et la suppression sont appliqués au gestionnaire de palette, /pas/ à la vue car c'est le gestionnaire qui responsable des données de couleur et il émettra les signaux du changement d'état à toutes les vues associées, y compris celle-ci, pour qu'elles se mettent à jour. C'est une méthode plus sûre que de mettre directement la vue à jour, puisque cela assure que toutes les vues sont mises à jour via le modèle en utilisant le même code.

Voici une fonction main() qui créé les deux vus de la palette.

```
int main(int argc, char **argv)
    {
        QApplication app(argc, argv);
        QSplitter splitter;
        splitter.setCaption(splitter.tr("MVC Palette"));
        PaletteIconView view1(&splitter);
        PaletteIconView view2(&splitter);
        PaletteModelManager *manager = PaletteModelManager::getInstance();
        manager->addColor(splitter.tr("Red"), Qt::red);
        manager->addColor(splitter.tr("Green"), Qt::green);
        manager->addColor(splitter.tr("Blue"), Qt::blue);
        app.setMainWidget(&splitter);
        splitter.show();
        return app.exec();
    }
```

Une fois que nous avons créer nos vues, nous ajoutons quelques couleurs. L'utilisateur paut ajouter, changer et retirer des couleurs en utilisant le menu contextuel que chaque vue fournit. Et quelque soit ce que fait l'utilisateur sur l'une des vues, cela est appliqué aux deux.

Conclusion

Copyright © 2009 - Kinji1. Aucune reproduction, même partielle, ne peut être faite de ce site et de l'ensemble de son contenu : textes, documents, images, etc sans l'autorisation expresse de l'auteur. Sinon vous encourez selon la loi jusqu'à 3 ans de prison et jusqu'à 300 000 E de dommages et intérêts.

Grâce aux méchanisme de signal-slot de Qt, l'implémentation de composants Modèle/Vue/Controlleur est simple. Une regard minutieux doit être donné quant au choix du moment de l'émission des signaux notifiant les modifications au modèle : avant ou après l'application des changements. Il est plus simple et plus sûr de mettre à jour les vues indirectement en laissant leur controlleur appeler le modèle plutôt que directement par leur controlleur. Notez également que pour une implémentation robuste vous devez vous assurer que les tentatives de mise à jour du modèle en réponse à un signal sont prises en charge de manière adaptée, par exemple nous ne voudrions pas que removeColor() soit appelée depuis un slot connecté à colorAdded().

Les classes présentées ici pourrient être améliorées de nombreuses manières, par exemple en créant un plugin pour PaletteIconView utilisable avec Qt Designer, ou en fournissant des signaux et des slots pour mettre à jour et notifier les chanagement de l'ID d'une couleur. La classe PaletteIconView pourrait être améliorée en apportant le support du "drag and drop", tandis que la classe PaletteModelManager pourrait être capable de sauvegarder et charger des palettes. Des vues additionnelles pourraient aussi être créées, par exemple avec des listes déroulantes. Une extension plus ambitieuse serait l'implémentation d'un méchanisme pour annuler et refaire les actions. Le code source complet de cet articles est disponible ici : Sources **qq10-mvc.zip (8K)**

Copyright © 2009 - Kinji1. Aucune reproduction, même partielle, ne peut être faite de ce site et de l'ensemble de son contenu : textes, documents, images, etc sans l'autorisation expresse de l'auteur. Sinon vous encourez selon la loi jusqu'à 3 ans de prison et jusqu'à 300 000 E de dommages et intérêts.