

Utilisation d'Undo/Redo avec les vues

Qt by Nokia

par Witold Wysota traducteur : Denys Bulant Qt Quarterly

Date de publication : 26/05/2009


Dernière mise à jour : 08/07/2010

Le module Undo/Redo de Qt, introduit avec Qt 4.2, fournit la possibilité de doter les utilisateurs de vos applications avec des moyens d'annuler les changements apportés aux documents, tout en fournissant aux développeurs une API facile à utiliser, basée sur le patron de conception **Command**.

Cet article est une traduction autorisée de **Undo/Redo with Item Views**, par Witold Wysota.

I - L'article original.....	3
II - Introduction.....	3
III - Choix d'une approche.....	3
IV - Annulation/Restauration avec des modèles personnalisés.....	4
V - Rendre annulables les modèles standards.....	7
VI - Prêt à restaurer.....	11
VII - Divers.....	11

I - L'article original

Qt Quarterly est une revue trimestrielle électronique proposée par Nokia à destination des développeurs et utilisateurs de Qt. Vous pouvez trouver les  **versions originales**.

Nokia, Qt, Qt Quarterly et leurs logos sont des marques déposées de *Nokia Corporation* en Finlande et/ou dans les autres pays. Les autres marques déposées sont détenues par leurs propriétaires respectifs.

Cet article est la traduction de l'article **Undo/Redo with Item Views** de Witold Wysota paru dans la Qt Quarterly Issue 25.

Cet article est une traduction d'un des tutoriels écrits par **Nokia Corporation and/or its subsidiary(-ies)** incluse dans la documentation de Qt, en anglais. Les éventuels problèmes résultant d'une mauvaise traduction ne sont pas imputables à Nokia.

II - Introduction

De prime abord, combiner les fonctionnalités d'annulation/restauration avec le module modèle/vue peut sembler une tâche dantesque. Dans cet article, nous espérons transformer une tâche apparemment difficile en une tâche facile, qui peut être appliquée sur plusieurs modèles.

III - Choix d'une approche

Il y a deux approches que nous pouvons prendre pour combiner le module modèle/vue avec celui d'annulation :

- Nous pouvons établir l'infrastructure d'annulation/restauration en nous basant sur le modèle : les commandes d'annulation et de restauration utilisent l'API de **QAbstractItemModel** afin de modifier les données sous-jacentes ; ou,
- Nous pouvons écrire le modèle de façon à instancier des commandes et les pousser sur la pile d'annulations.

La première approche peut sembler la plus simple, puisqu'elle fonctionne avec n'importe quel modèle, et fournit au module d'annulation un contrôle total. Vous pouvez créer autant de types de commandes différentes que désiré, et chacune d'elle peut appeler plusieurs méthodes de l'API du modèle dans leurs méthodes `undo()` et `redo()`.

Par exemple, une classe dérivant de **QUndoCommand** qui modifie l'état de coche des objets peut fournir cette implémentation :

```
class ChStateCmd : public QUndoCommand
{
public:
    ChStateCmd(const QModelIndex &index, Qt::CheckState s,
               QString text, QUndoCommand *parent = 0)
        : QUndoCommand(text, parent), ind(index) {
        Old = qvariant_cast<Qt::CheckState>(
            index.data(Qt::CheckStateRole));

        New = s;
    }
    void redo() {
        ind.model()->setData(ind, New, Qt::CheckStateRole);
        ind.model()->setData(ind, New==Qt::Checked
            ? Qt::green : Qt::red, Qt::BackgroundRole);
    }
    void undo() {
        ind.model()->setData(ind, Old, Qt::CheckStateRole);
        ind.model()->setData(ind, Old==Qt::Checked
            ? Qt::green : Qt::red, Qt::BackgroundRole);
    }
private:
```

```
QPersistentModelIndex ind;
Qt::CheckState Old, New;
};
```

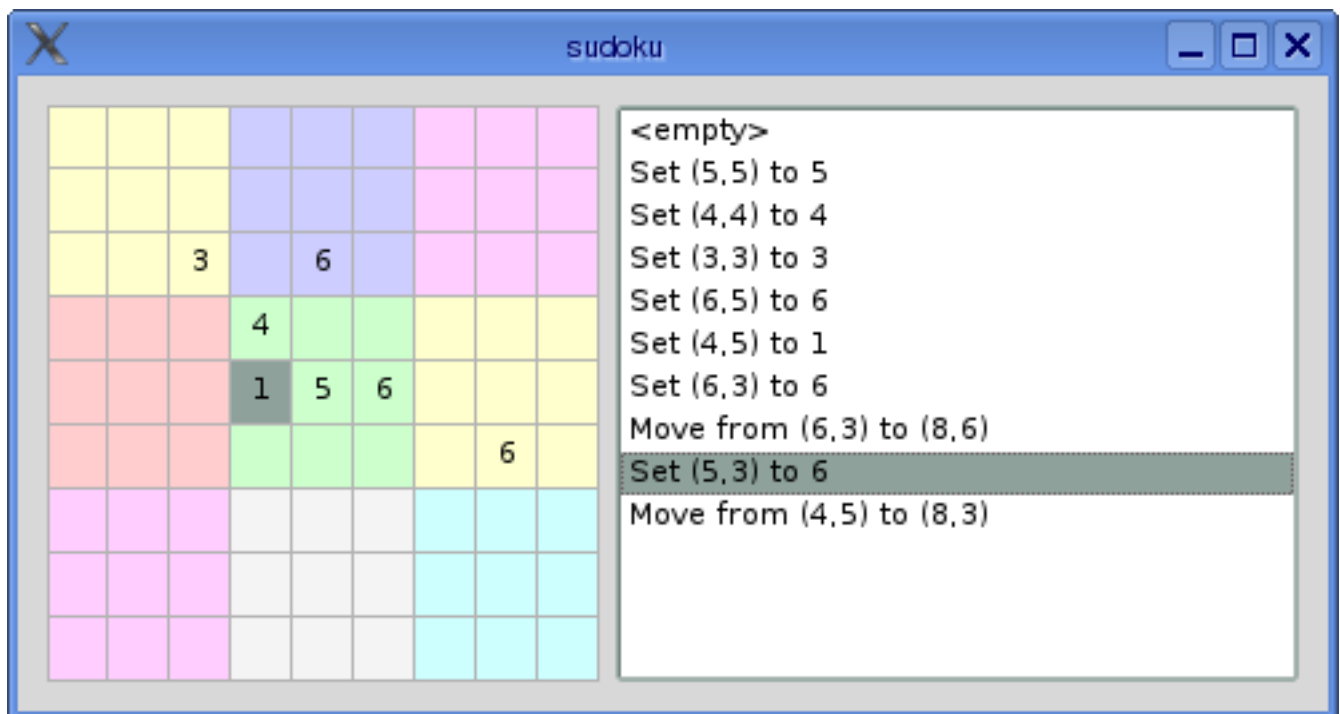
Implémenter une telle structure représente un effort minime dans la plupart des cas. Cependant, son principal inconvénient est visible lorsque quelque chose appelle votre modèle directement, sans s'interfacer par le module d'annulation/restauration, la pile d'annulations perdra sa cohérence et ne sera plus capable de ramener le modèle à l'état désiré ; certains changements seront ignorés alors que d'autres seront écrasés par ceux stockés dans la pile d'annulations.

Un exemple de ce cas de figure est le glissé/déposé. Dans cette situation, c'est la vue qui appelle **QAbstractItemModel::dropMimeData()**, il n'y a donc aucun moyen de contrôler l'appel ni de savoir comment l'annuler. Vous pourriez dériver de la vue et essayer de réimplémenter certaines méthodes pour essayer d'englober l'appel à dropMimeData() dans une commande, mais ceci casserait la modularité du code et, par-dessus tout, c'est fastidieux et lourd.

Maintenant, il devient évident qu'embarquer la fonctionnalité d'annulation dans le modèle lui-même est une meilleure idée.

IV - Annulation/Restauration avec des modèles personnalisés

C'est souvent plus simple et facile de définir l'infrastructure d'annulation dans vos propres modèles. Ils fournissent un contrôle total sur les données et c'est exactement ce que vous voulez - il n'y a aucun risque de manquer des emplacements vulnérables dans lesquels des données peuvent être injectées dans le modèle à votre insu.



Dans cette section, nous implémenterons un modèle simple gérant un Sudoku avec la possibilité d'annuler et restaurer des mouvements effectués par le joueur. Une planche de sudoku consiste en neuf sous-champs carrés, chacun doté de neuf cases. Le but est, pour le joueur, de remplir la grille avec des nombres allant d'un à neuf, de façon à ce que tous les nombres au sein d'une rangée, d'une colonne et d'un sous-champ soient uniques. Nous allons donc avoir besoin d'un modèle en tableau de taille constante.

Une version abrégée de la classe du modèle ressemble à ceci :

```
class SudokuBoard : public QAbstractTableModel
{
    Q_OBJECT

    friend class ChangeValueCommand;
    friend class MoveDataCommand;

public:
    SudokuBoard(QObject *parent = 0);
    ...
    bool setData(const QModelIndex &index, ...);
    QStringList mimeTypes() const;
    QMimeData *mimeData(const QModelIndexList &) const;
    bool dropMimeData(const QMimeData *data, ...);
    QUndoStack *undoStack() const;
    Qt::DropActions supportedDropActions () const;

protected:
    void emitDataChanged(const QModelIndex &index)
    {
        emit dataChanged(index, index);
    }

    QUndoStack *m_stack;
    int m_data[9][9];
};
```

Comme vous pouvez le constater, nous gardons la pile d'annulations dans le modèle de façon à pouvoir la modifier directement. De plus, notez la déclaration d'amitié avec les deux classes commandes. Ceci nous permet de manipuler les données privées du modèle. Alternativement, nous pourrions fournir une classe assistante qui aurait l'accès aux données privées et lui faire émettre des signaux au nom du modèle. Cette approche est particulièrement pratique si vous utilisez une implémentation privée ou si vous possédez un grand nom de types de commandes et voulez les cacher au monde extérieur. Ici, nous utilisons seulement deux commandes, il est donc plus aisé de n'utiliser qu'une seule classe.

```
QUndoStack *SudokuBoard::undoStack() const
{
    return m_stack;
}

Qt::DropActions SudokuBoard::supportedDropActions() const
{
    return Qt::CopyAction | Qt::MoveAction;
}
```

Nous passerons le code du constructeur (où toutes les cellules du tableau sont mises à zéro et où nous construisons une pile d'annulations), le destructeur, ainsi que la fonction data(). Puisque nous voulons permettre à l'utilisateur de déposer des données sur la cellule, le modèle doit informer les autres composants du type de données qu'il peut gérer. Pour ce faire, nous devons réimplémenter mimeTypes() afin de renvoyer un type MIME personnalisé :

```
QStringList SudokuBoard::mimeTypes()
{
    return QStringList() << "application/x-sudokucell";
}
```

Bien sûr, nous devons être capables de créer des objets destinés au glissé/déposé (par souci de simplicité, nous autorisons la dépose d'un seul élément à la fois) :

```
QMimeData *SudokuBoard::mimeData(
    const QModelIndexList &list) const
{
    if (list.isEmpty()) return 0;
    QMimeData *mime = new QMimeData;
    QModelIndex index = list.at(0);
    QString cell = QString("%1x%2=%3")
        .arg(index.column())
```

```
        .arg(index.row())
        .arg(index.data(Qt::DisplayRole));
    QByteArray ba = cell.toAscii();
    mime->setData("application/x-sudokucell", ba);
    return mime;
}
```

Jusqu'ici, il n'y a rien de nouveau pour le créateur de modèle expérimenté. Tout ce qu'il nous reste à faire est d'écrire les deux méthodes qui vont générer des changements sur les données du modèle. C'est ici que la magie commence.

```
bool SudokuBoard::setData(const QModelIndex &index,
    const QVariant &val, int role)
{
    ...
    if (role != Qt::EditRole)
        return false;
    m_stack->push(new ChangeValueCommand(index, val, this)); // [NdT] Opérateur de déréférencement
    utilisé par souci de cohérence.
    return true;
}
```

L'implémentation de setData() est plutôt simple - nous autorisons seulement les modifications du rôle Qt::EditRole et c'est réalisé en empilant un objet ChangeValueCommand sur la pile d'annulations. Notez que nous n'émettons pas le signal dataChanged() - c'est une responsabilité de la commande que de le faire.

La prochaine étape est d'implémenter le dépôt de données d'une façon similaire :

```
bool SudokuBoard::dropMimeData(const QMimeData *data,
    Qt::DropAction action, int row, int column,
    const QModelIndex &par)
{
    QString str(data->data("application/x-sudokucell"));
    if (str.isEmpty()) return false;
    int c = str[0].toAscii()-'0';
    int r = str[2].toAscii()-'0';
    int v = str[4].toAscii()-'0';
    if (par.data().toInt() == v) return false;

    switch (action) {
    case Qt::CopyAction:
        m_stack->push(new ChangeValueCommand(index(par.row(), // Opérateur de déréférencement utilisé par
        par.column()), v, this));
        return true;
    case Qt::MoveAction:
        m_stack->push(new MoveDataCommand(index(r, c), // Opérateur de déréférencement utilisé
        par, this));
        return true;
    default:
        return false;
    }
}
```

Tout ce qu'il nous reste à faire est d'implémenter les commandes nous-mêmes.

```
class ChangeValueCommand : public QUndoCommand
{
public:
    ChangeValueCommand(const QModelIndex &index,
        const QVariant &value, SudokuBoard *model)
        : QUndoCommand(), m_model(model)
    {
        m_old = index.data(Qt::DisplayRole);
        m_new = value;
        m_row = index.row();
        m_col = index.column();
    }
}
```

```

    setText(QApplication::translate("ChangeValueCommand",
        "Set (%1,%2) to %3").arg(m_col+1)
        .arg(m_row+1).arg(m_new.toInt()));
}
void redo()
{
    QModelIndex index = m_model->index(m_row, m_col);
    m_model->m_data[m_col][m_row] = m_new.toInt();
    m_model->emitDataChanged(index);
}
void undo()
{
    QModelIndex index = m_model->index(m_row, m_col);
    m_model->m_data[m_col][m_row] = m_old.toInt();
    m_model->emitDataChanged(index);
}
private:
    SudokuBoard *m_model;
    QVariant m_new, m_old;
    int m_row, m_col;
};

```

Dans le constructeur, nous stockons toutes les données nécessaires, et, dans redo() et undo(), nous modifions les données et émettons un signal en utilisant une fonction membre protégée du modèle. MoveDataCommand est similaire - la différence est que nous modifions la cellule source ainsi que celle de destination.

Notre modèle de Sudoku est réellement simple - il est plat et n'autorise pas l'ajout (ou suppression) de lignes ni de colonnes. Pour un modèle plus complexe, plus de commandes seront nécessaires, en considérant toutes les fonctionnalités du modèle ainsi que toutes méthodes de manipulation de données que vous voudriez ajouter à l'API du modèle. De plus, prenez en considération le fait de pouvoir créer des commandes personnalisées gratuitement de l'extérieur du modèle en utilisant **QUndoStack::beginMacro()** - utilisez-le afin de créer des commandes plus complexes.

V - Rendre annulables les modèles standards

Très souvent, nous utilisons les modèles par défaut livrés avec Qt au lieu d'utiliser les nôtres. En de tels cas, ce n'est pas possible d'embarquer la fonctionnalité d'annulation/restauration au sein du modèle puisque nous n'avons pas accès à son implémentation.

Heureusement, nous pouvons tout de même avoir l'infrastructure d'annulation/restauration dans le modèle sans même toucher au modèle original - nous pouvons utiliser un modèle proxy afin de manipuler le modèle original à la demande.

Puisque nous devons uniquement fournir une correspondance un à un entre les indices, il est plus simple de partir d'un QSortFilterProxyModèle. Nous n'avons même pas besoin de régler le filtre ni quoi que ce soit d'autre.

```

class UndoRedoProxy : public QSortFilterProxyModel
{
    friend class UndoRedoProxyHelper;
public:
    UndoRedoProxy(QObject *parent=0);
    bool setData( ... );
    ...
    QUndoStack *undoStack() const { return m_stack; }

protected:
    virtual QString setDataText(const QModelIndex &,
                                const QVariant &, int) const;

private:
    QUndoStack *m_stack;
    UndoRedoProxyHelper *m_helper;
    bool m_cachedResult;
    void setData_helper(const QModelIndex &index,
                        const QVariant &value, int role);
};

```

```
...  
};
```

À nouveau, nous fournissons un accès à la pile d'annulations et réimplémentons les méthodes qui génèrent des changements sur le modèle. Ce coup-ci, au lieu de déclarer chacune des commandes d'annulation comme amies du proxy, nous n'aurons qu'un seul objet de la classe UndoRedoProxyHelper qui appellera les méthodes privées *_helper() du proxy au nom des commandes.

```
struct UndoRedoProxyHelper  
{  
    UndoRedoProxy *proxy;  
    void setData(const QModelIndex &index,  
                 const QVariant &value, int role){  
        proxy->setData_helper(index, value, role);  
    }  
    ...  
};
```

Les méthodes assistantes sont simplement des appels aux méthodes du modèle sous-jacent pour réaliser la tâche nécessaire et émettre les bons signaux au besoin.

```
void UndoRedoProxy::setData_helper(const QModelIndex &i,  
                                   const QVariant &value, int role){  
    m_cachedResult = QSortFilterProxyModel::setData(i,  
                                                     value, role);  
    emit dataChanged(i, i);  
}
```

La variable booléenne m_cachedResult nous permet de renvoyer une valeur à partir des méthodes qui doivent en renvoyer. Son usage est visible dans la méthode publique du proxy setData().

```
bool UndoRedoProxy::setData( ... ){  
    if (!index.isValid() || index.data(role) == value  
        || value.isNull()) return false;  
    SetDataCommand *cmd = new SetDataCommand(index, value,  
                                              role, m_helper);  
    cmd->setText(setDataText(index, value, role));  
    m_stack->push(cmd);  
    if(!m_cachedResult) delete m_stack->command(0);  
    return m_cachedResult;  
}  
  
QString UndoRedoProxy::setDataText(const QModelIndex &  
                                   const QVariant &, int) const{  
    return tr("Set Data");  
}
```

Lorsque nous poussons une commande sur la pile, redo() sera appelée, ce qui entraînera l'appel de la commande assistante qui stockera le résultat de l'opération du modèle sous-jacent dans une variable booléenne. Ensuite, nous pouvons retourner la valeur ainsi obtenue à notre environnement. Notez que nous supprimons la commande si setData() échoue dans le modèle source et la façon dont nous récupérons le texte pour la commande en utilisant la méthode virtuelle protégée setDataText(); ceci nous permet d'ajouter nos propres descriptions en dérivant de ce modèle et en réimplémentant les méthodes respectives.

Passons maintenant aux classes de commandes elles-mêmes. Premièrement, nous allons créer une classe de base pour toutes nos commandes. Cela rendra possible, et de façon aisée, d'opérer sur le proxy en utilisant l'objet subsidiaire.

```
class UndoProxyCommand : public QUndoCommand  
{  
public:  
    UndoProxyCommand(UndoRedoProxyHelper *h,  
                     QUndoCommand *par=0):QUndoCommand(par){ m_helper = h;}
```



```
UndoRedoProxyHelper *helper() const{return m_helper;}
UndoRedoProxy *proxy() const{return m_helper->proxy;}
private:
    UndoRedoProxyHelper *m_helper;
};
```

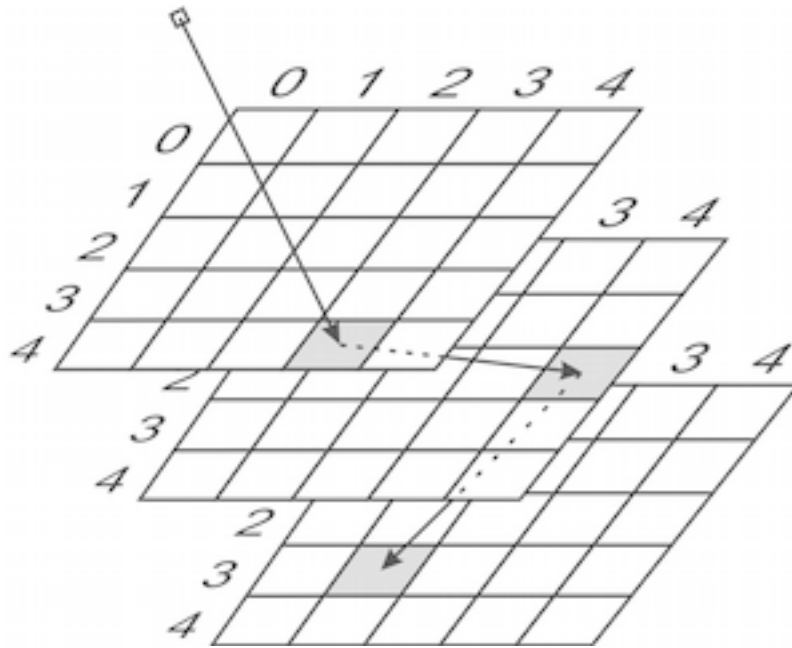
Vous trouverez toutes les classes de commandes implémentées dans le code accompagnant cet article, donc nous ne montrerons que le plus simple : celui opérant sur setData().

```
class SetDataCommand : public UndoProxyCommand
{
public:
    SetDataCommand(const QModelIndex &index,
                   const QVariant &value, int role,
                   UndoRedoProxyHelper *helper)
        : UndoProxyCommand(helper)
    {
        m_index = pathFromIndex(index);
        m_value = value;
        m_role = role;
    }
    void undo()
    {
        QModelIndex index = pathToIndex(m_index, proxy());
        QVariant old = index.data(m_role);
        helper()->setData(index, m_value, m_role);
        m_value = old;
    }
    void redo()
    {
        QModelIndex index = pathToIndex(m_index, proxy());
        QVariant old = index.data(m_role);
        helper()->setData(index, m_value, m_role);
        m_value = old;
    }
private:
    Path m_index;
    QVariant m_value;
    int m_role;
};
```

L'objet Path et les deux fonctions l'utilisant, pathToIndex() et pathFromIndex(), sont utilisés pour nous aider à agir sur des modèles hiérarchiques et qui changent rapidement.

Le chemin est utilisé pour obtenir un index convenable dans le modèle - comme nous le savons, ces objets sont volatiles et nous ne devrions les stocker nulle part. Ce pourrait être tentant d'utiliser **QPersistentModelIndex** à la place, mais il apparaît qu'il s'agit de la mauvaise approche puisque l'index va devenir invalide si nous enlevons l'objet vers lequel il pointe. Dans tous les cas, il ne serait pas possible d'agir sur une séquence de commandes qui contient au moins une opération qui insère ou supprime des objets. De ce fait, nous devons trouver un moyen de stocker la position de l'objet sans utiliser d'index du modèle.

C'est ici qu'interviennent les fonctions mentionnées plus haut - elles convertissent un **QModelIndex** en une entité qui nous permet de retrouver l'objet lorsque nous en aurons besoin ultérieurement. Comment cacheriez-vous le trésor de pirate de façon à être certain de pouvoir le retrouver plus tard ? Vous enterrez le butin sous un arbre, trouvez un objet facile à localiser (telle une pierre sur la plage), et écrivez des pas à effectuer de la pierre jusqu'à l'arbre ! "108 pas à gauche, 207 à droite, 42 à gauche, creuser sous l'arbre qui ressemble à une bouteille de rhum."



Nous pouvons procéder à l'identique avec notre modèle. Regardez le diagramme - nous pouvons stocker une liste de numéros de lignes et de colonnes que nous devons suivre à partir de l'index racine du modèle pour atteindre l'objet que nous recherchons. Path est une liste de paires de la forme [ligne, colonne] qui nous montre le chemin à travers le modèle.

```
typedef QPair<int, int> PathItem;
typedef QList<PathItem> Path;

Path pathFromIndex(const QModelIndex &index){
    QModelIndex iter = index;
    Path path;
    while(iter.isValid()){
        path.prepend(PathItem(iter.row(), iter.column()));
        iter = iter.parent();
    }
    return path;
}

QModelIndex pathToIndex(const Path &path,
                        const QAbstractItemModel *model){
    QModelIndex iter;
    for(int i=0; i<path.size(); i++){
        iter = model->index(path[i].first,
                           path[i].second, iter);
    }
    return iter;
}
```

Ceci nous permet de stocker des chemins plutôt que des indices de façon à pouvoir accéder à l'objet quelque soit le nombre d'insertions ou suppressions effectuées entre-temps. L'utilisation de cette approche est facile à implémenter dans les commandes restantes.

Il y a deux choses à se rappeler. La première est que, si vous supprimez des objets, rangées ou colonnes, vous devriez sauver les données de tout les objets supprimés (y compris les enfants) quelque part de façon à pouvoir annuler la suppression. La seconde est qu'il pratique de pouvoir donner des descriptions significatives aux commandes poussées sur la pile. Pour ce faire, vous pouvez fournir des méthodes au modèle qui sont appelées chaque fois qu'une commande est poussée sur la pile, qui renverrait à son tour une description basée sur les paramètres de la commande exécutée.

VI - Prêt à restaurer

Vous pouvez tester le proxy en modifiant l'exemple "Editable Tree Model" fourni avec Qt. Après avoir joué avec, pourquoi ne pas ajouter à vos modèles précédemment implémentés les capacités d'annulation/restauration ? Amusez-vous !

VII - Divers

Source Les fichiers source

Au nom de toute l'équipe Qt, j'aimerais adresser le plus grand remerciement à Nokia pour nous avoir autorisé la traduction de cet article !

J'aimerais aussi adresser un immense merci à **dourouc05** pour ses relectures et corrections !