Vues en QML et redéfinition de modèles en C++.

François Delobel

Master TechMed et Setsys, Université Clermont-Auvergne

Plan

1 Vues Qt
2 Modèle / Vues
- Principes
- Modèle en CPP: principes
- Vues: principes
- Modèle: programmation
- Exemple: les données
- main.qml
- Exemple: le modèle
- Le modèle - 0 Les rôles
- Le modèle - 1 Lecture
- Le modèle - 2 Écriture
- Le modèle - 3 Ajout simple
- Le modèle - 3 Ajout multilignes
- Le modèle - 4 Suppression
- Le main

iut Université d'Auvergne

Les vues en Qt

Vue: affichage de données multiples / complexes.

- Éléménts similaires, données d'une table, résultat appel API web...
- La vue est uniquement la partie graphique.
- Typiquement scrollable.
- Liste: vue à une dimension (très utilisé sur mobile)
- Tableau: vue à deux dimensions
- Arbre: arborescence (fichiers, paramètres...)

Flickable: le “machin scrollant”

- Élément “flickable”, qu’on peut déplacer au doigt ou à la souris.
- Sous types possibles (non limitatif)
  - `ListView`
  - `GridView`
  - `TableView`
  -

Pourquoi faire des vues et modèles?

Buts

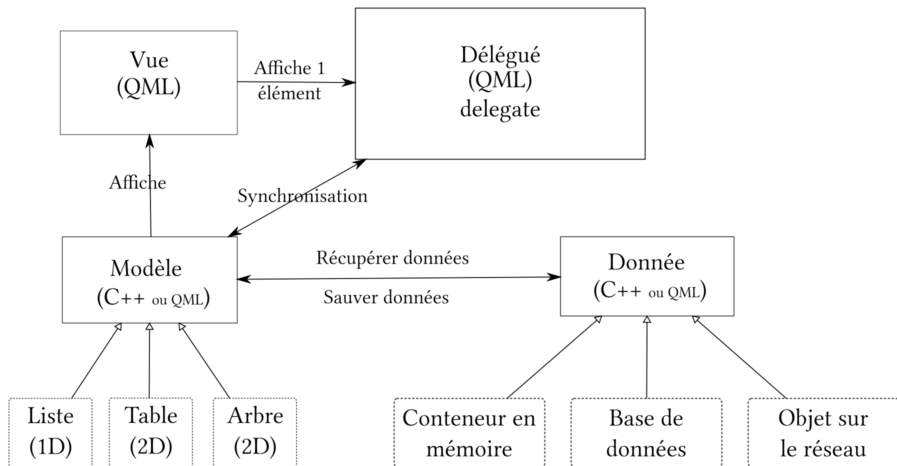
- Afficher / éditer des données multiples
- Mieux séparer l'affichage (UI) des données pour pouvoir changer les sources de données.

Quatre partenaires pour ce faire

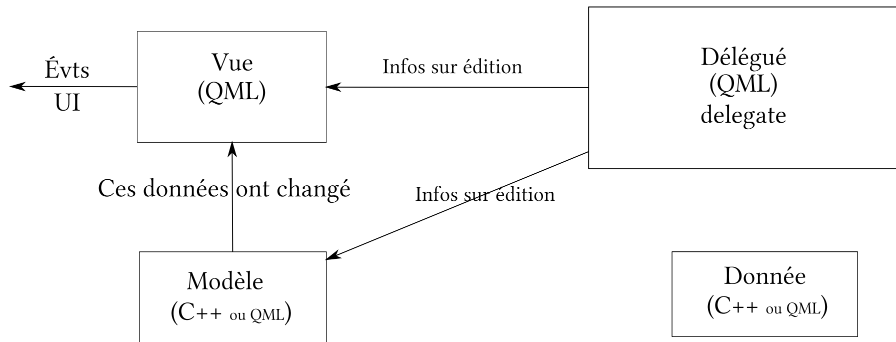
- *Vue*: Affiche et permet à l'utilisateur d'agir sur les éléments.
- *Délégué*: Permet d'afficher ou d'éditer *finement* un élément.
- *Modèle*: Fournit à la vue les informations à afficher en cherchant les informations dans la partie métier.
- *Données*: Les données, au sens métier. Elles sont totalement masquées par le modèle.

iut Université d'Auvergne

Vue d'ensemble



Rôle des signaux



ListView avec ListModel QML

```
ListView {
    model: ListModel {    // Les données
        ListElement { name: "Fourme" }
        ListElement { name: "Saint Nectaire" }
    }

    delegate: Rectangle {
        width: ListView.view.width
        color: "lightgray"
        border.color: "black"

        Text {
            anchors.centerIn: parent
            text: name
            font.pixelSize: 16
        }
    }
}
```

Mais où vivent les données?

- Démo: exemple *view*.
- Les données sont déclarées depuis le QML.
- Ok dans certaines situations, pas dans toutes.
- Comment faire un modèle en CPP ?

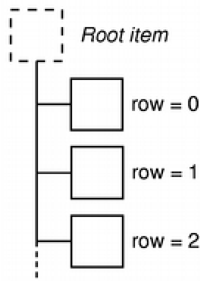
iut
Université d'Auvergne

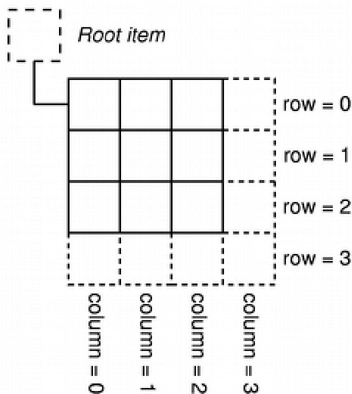
Le modèle

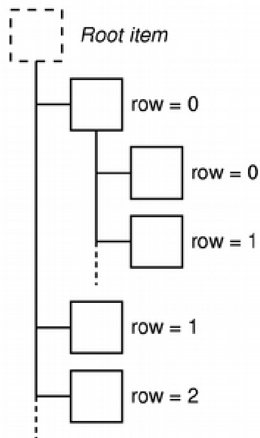
- Hérite de `QAbstractItemModel`.
- Masque l'accès aux données:
  - Sait quelle donnée se trouver à quelle position dans le modèle
  - Position? Besoin d'un *index* (à une ou deux dimensions).
  - Notion de *rôle*: à quelle facette d'un élément on s'adresse
    - → Souvent correspondant à une *propriété* d'un objet.
- Donner accès à des éléments (consultation, insertion, suppression...)
- Masquer le stockage des éléments (listes, SQL, fichier...)
- Prévenir la vue de tout ce qui se passe (signaux).
- Gère des `header`
- Gérer les sélections multiples
- Gérer les tri, les ordonnancement par l'utilisateur

les trois organisations standard des modèles

Image volée à la doc de Qt







Plusieurs organisations des données

- Un modèle *organise* des données...
  - Plusieurs types d'organisations possible.
- ... en les *masquant*
  - Certaines données impliquent des organisations (Système de fichier, tables SQL...)

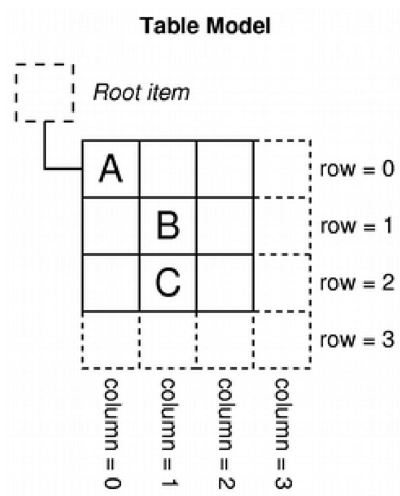
Modèles prédéfinis en C++

- `QAbstractItemModel`: le modèle abstrait de base.
- `QStringListModel`: listes de `QString`. Simple mais limité.
- `QObjectList`: Liste de `QObject`, mais des défauts
  - La liste ne sait pas prévenir la vue qu'elle a changé.
  - Par contre la MAJ des éléments fonctionne grâce aux propriétés.
- `QStandardItemModel`: Arbres ou listes d'éléments ajoutés un par un.
  - Pratique si utilisé dés le début, mais pénible si la couche métier comporte déjà des conteneurs.
- `QAbstractListModel`: redéfinir vos listes (1 dimension).
- `QAbstractTableModel`: redéfinir des tables (2 dimensions).
- `QFileSystemModel`: Accès aux fichiers
- `QSqlQueryModel`, `QSqlTableModel`, `QSqlRelationalTableMode`, pour l'accès aux BD relationnelles.

Model Index

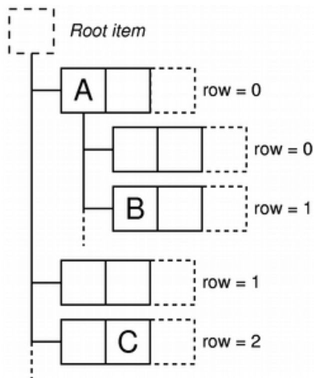
- Chaque type de modèle a son organisation
- Comment identifier un élément?
- ... Via une position, mais elle varie (Position à une ou deux dimensions par exemple).
- Donc abstrait par un `QModelIndex`
- Basé sur des lignes (*row*) et des colonnes (*column*).
- Et éventuellement sur des *parents* pour les arbres.

L'index du modèle *table*




```
QModelIndex indexC = model->index(2, 1, QModelIndex());
```

L'index du modèle *Tree* (Arbre)




```
// Index absolu
QModelIndex indexA = model->index(0, 0, QModelIndex());
QModelIndex indexC = model->index(2, 1, QModelIndex());
// Index relatif à un élément
QModelIndex indexB = model->index(1, 0, indexA);
```

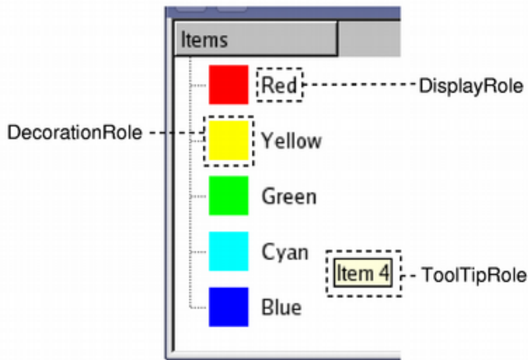
D'autres index existent!

- Redéfinissable par l'utilisateur pour s'adapter au modèle
- Exemple: les système de fichier

```
QFileSystemModel *model = new QFileSystemModel;
QModelIndex parentIndex = model->index(QDir::currentPath());
```

Les rôles

- Chaîne de caractère utilisée par la vue pour demander une information au modèle.
- Sorte de facette (façon de voir) un objet.
- Souvent lié à une propriété mais pas toujours.



Modèles de QML

- QML fournit quelques modèles simples qui peuvent éviter de redéfinir un modèle en C++.
  - `ListModel`, composée de `ListElement`
  - `XmlListModel`, qui lit du XML (y compris sur des URL contenant des flux XML ou des WebServices)
  - `VisualItemModel`, qui contient juste des éléments graphiques QML.

Vues

- Présentent les données, reçoivent les interactions de l'utilisateur.
- S'appuient sur un modèle (attribut `model`), récupèrent les infos grâce à des index et des rôles.
- S'enregistrent auprès du modèle pour être averti de ses modification (via les signaux).
- Donnent les informations au délégué afin que celui ci affiche un élément (l'élément, mais aussi l'`index`).
- Demandent au modèle d'effectuer des ajouts, des suppressions (via des méthodes).

Vues proposées

- `ListView` (ou `SilicaListView`): affichage en 1 dimension
- `GridView`(ou `SilicaGridView`): 2D
- `SlideshowView`: défilement cyclique
- `QAbstractItemView`: la base de tout ça (en C++) et d'autres...

Responsabilités des méthodes d'un modèle

Méthodes à redéfinir impérativement:

- Lecture
  - `data()`: accès aux données à une position
  - `rowCount()`: nombre d'éléments
  - Si accès depuis QML: `roleNames()`: renvoie un tableau de rôles.
- Modifier un élément
  - `setData()` : modifie un élément à une position donnée.
- Ajout et supressions d'éléments
  - `insertRows()`: ajoute des éléments à une position
  - `removeRows()`: supprime des éléments

Difficultés et pièges

- index: permet aussi de gérer des tableaux, des sélections multiples...
- Utilisation du `QVariant` pour faire transiter les données
- Nécessité d'envoyer des signaux ou d'appeler des fonctions *hooks* pour prévenir des changements

```
class Game {
public:
    Game(const QString &name, int mark);

    QString name() const;
    int mark() const;

private:
    QString m_name;
    int m_mark;
};
```

```
Game::Game(const QString &name, int mark)
    : m_name(name), m_mark(mark) {}

QString Game::name() const {
    return m_name;
}

int Game::mark() const {
    return m_mark;
}
```

```
ListView {
    anchors.fill: parent
    model: gameModel  //  C++ model

    delegate: Rectangle {
        width: ListView.view.width
        height: 50
        color: "lightgray"
        border.color: "black"

        Row {
            anchors.fill: parent
            Text { text: name  }
            Text { text: mark  }
        }

        MouseArea {
            anchors.fill: parent
            onClicked: console.log("Selected:", name, "Mark:", mark)
        }
    }
}
```

```cpp
class GameModel : public QAbstractListModel {
    Q_OBJECT

public:
    explicit GameModel(QObject *parent = nullptr);

    enum Roles {
        NameRole = Qt::UserRole + 1,
        MarkRole
    };

    // Lecture
    int rowCount(const QModelIndex &parent = QModelIndex()) const override;
    QVariant data(const QModelIndex &index, int role = Qt::DisplayRole) const overr
    QHash<int, QByteArray> roleNames() const override;

    //
    bool setData(const QModelIndex &index, const QVariant &value, int role) overrid
    bool insertRows(int row, int count, const QModelIndex &parent = QModelIndex()) 
    bool removeRows(int row, int count, const QModelIndex &parent = QModelIndex()) 

private:
    QVector<Game> m_games;
};
```

```
QHash<int, QByteArray> GameModel::roleNames() const {
    return {
        { NameRole, "name" },
        { MarkRole, "mark" }
    };
}
```

```
int GameModel::rowCount(const QModelIndex &parent) const {
    if (parent.isValid()) return 0;
    return m_games.size();
}

QVariant GameModel::data(const QModelIndex &index, int role) const {
    if (!index.isValid() || index.row() >= m_games.size())
        return QVariant();

    const Game &game = m_games.at(index.row());
    if (role == NameRole)
        return game.name();
    else if (role == MarkRole)
        return game.mark();

    return QVariant();
}
```

```
bool GameModel::setData(const QModelIndex &index, const QVariant &value, int role) {
    if (!index.isValid() || index.row() >= m_games.size())
        return false;

    Game &game = m_games[index.row()];

    if (role == NameRole) {
        game = Game(value.toString(), game.mark());
    } else if (role == MarkRole) {
        game = Game(game.name(), value.toInt());
    } else {
        return false;
    }

    emit dataChanged(index, index, {role});
    return true;
}
```

```
bool GameModel::insertRows(int row, int count, const QModelIndex &parent) {
    if (row < 0 || row > m_games.size())
        return false;

    beginInsertRows(parent, row, row + count - 1);
    for (int i = 0; i < count; ++i) {
        m_games.insert(row, Game("New Game", 0));
    }
    endInsertRows();

    return true;
}
```

```
bool GameModel::removeRows(int row, int count, const QModelIndex &parent) {
    if (row < 0 || row >= m_games.size() || (row + count) > m_games.size())
        return false;

    beginRemoveRows(parent, row, row + count - 1);
    for (int i = 0; i < count; ++i) {
        m_games.removeAt(row);
    }
    endRemoveRows();

    return true;
}
```

```
int main(int argc, char *argv[]) {
    QGuiApplication app(argc, argv);
    QQmlApplicationEngine engine;

    GameModel gameModel;
    engine.rootContext()->setContextProperty("gameModel", &gameModel);

    engine.load(QUrl(QStringLiteral("qrc:/main.qml")));
    if (engine.rootObjects().isEmpty())
        return -1;

    return app.exec();
}
```