

Reference: <https://qt.developpez.com/tutoriels/mvc/apostille-delegates-mvd/>

The architecture *MVC (Model View Controller)* is a design pattern well known and widely used when creating GUIs.

# I. Some booster shots

## IA. The MVC architecture

Originally from the Smalltalk language, this model makes it possible to divide a user interface into three entities:

- the model contains the data. It deals with data processing and interaction with, for example, a database;
- the view is the graphical representation. It corresponds to the display part of the data, from what the user can see;
- the controller supports interaction with the user, receiving all the events triggered by the user (click, selection ...) and subsequently updating the data.

This architecture makes it possible to **separate** the different entities of an application, resulting in a flexible, clear and maintainable architecture.

## IB. Implementation with Qt

In version 4, Qt introduced a set of new views implementing a *model / view* architecture.

### IB-1. Models

All templates provided by Qt are based on the `QAbstractItemModel` class. This class is the most abstract and the highest in the class hierarchy of the different models. This class provides an interface that all models must follow in order to be used correctly with a view.

**Basic, Qt provides a set of models that can be directly used as:**

- `QStringListModel`: Storage of a `QString` list;
- `QFileSystemModel`: a set of information about a file or directory on the local file system (formerly `QDirModel`);
- `QStandardItemModel`: management of any type of structure, complex or not;
- `QSqlTableModel`, `QSqlRelationalTableModel`: access to a database (SQLite, MySQL ...).

However, sometimes (or often) these models are not suitable for use and you want to create your own templates. For that, nothing really complex. It suffices to create a class derived from one of the following classes:

- `QAbstractItemModel`: As mentioned above, this is the most abstract class;
- `QAbstractListModel`: abstract class providing an interface for a list type model (associated with a `QListView`);
- `QAbstractTableModel`: Abstract class providing an interface for an array type model (associated with a `QTableView`).

Note that there is no separate class for a hierarchical model (QTreeView). It is simply QAbstractItemModel.

You will notice that all these classes use the same root, which is QAbstractXModel.

Once we have chosen the best base class for our use, it only remains to redefine some virtual methods such as:

- `int QAbstractItemModel :: rowCount (const QModelIndex & parent = QModelIndex ()) const`: function that returns the number of lines in the model;
- `QVariant QAbstractItemModel :: data (const QModelIndex & index, int role = Qt :: DisplayRole) const`: A function that returns the data associated with the role role for the index index .

For modifiable models, it is also necessary to redefine the `bool` function `QAbstractItemModel :: setData (const QModelIndex & index, const QVariant & value, int role = Qt :: EditRole) .`

However, creating custom templates is not the focus of this tutorial. Although we will briefly discuss it through an example, I refer you to [the documentation](#) dealing with the subject in depth.

## IB-2. The views

**Once you have a model (already existing or customized), you have three types of views:**

- QListView: list of elements;
- QTableView: array of elements;
- QTreeView: representation of elements in hierarchical form (element tree).

Once the view is chosen, the model is assigned to it using the `void` function `QAbstractItemView :: setModel (QAbstractItemModel * model) .`

For more details, you are free to refer to [the documentation](#) .

# II. Let's talk about the controller

As you will have probably noticed, we have not yet mentioned the last aspect, namely the controller. Here's the truth: Qt does not use quite an MVC architecture. This is not a C but a D. For those who have not made the connection, there is a D delegate ( *delegate* ). Yes, now you know everything: Qt actually uses a model-view architecture surrounded by a delegate.

## II-A. Presentation of the concept

But what does this famous delegate serve for?

Unlike a traditional MVC architecture, Qt's model-view architecture does not provide a real component for managing user interactions. As a result, this is handled by the view itself.

**However, for reasons of flexibility, the interaction and the user inputs are not taken into account by a completely separate component, namely the controller, but by an internal component to the view: the delegate. This component is responsible for two things:**

- Customize editing of items using an editor
- customize the rendering of elements within a view.

Thus, thanks to a delegate, you can customize how user input will be managed, as well as the rendering of elements. For example, when using a `QTableView` with an editable template or clicking on a cell in your table, it is possible to modify the value of the cell with a `QLineEdit`. This is possible thanks to the delegate who provided the view with a way to edit the data through a `QLineEdit` component.

## II-B. Use an existing delegate

You will probably have noticed that, by default, even if you have not set any delegate, your view is responsible for providing a way to edit your data. This is because the views already have a default delegate: `QStyledItemDelegate`. Indeed, basic, Qt provides two delegates:

- `QItemDelegate`;
- `QStyledItemDelegate`.

The only difference between these two delegates is that `QStyledItemDelegate` uses the current style for rendering data. These two delegates, however, inherit the same class, `QAbstractItemDelegate`, providing a generic base interface for all delegates. Qt also provides `QSqlRelationalDelegate`, which allows you to edit and display data from a `QSqlRelationalTableModel`.

However, even if the views have a default delegate, it is of course possible to modify and set it with the `void` function `QAbstractItemView :: setItemDelegate (QAbstractItemDelegate * delegate)`. It is also possible to retrieve the current delegate thanks to the `QAbstractItemDelegate * QAbstractItemView :: itemDelegate () const` function.

## II-C. Let's take a look at the inner workings

We said that the delegate was partly responsible for providing the user with a way to edit the data within a view by providing a publisher, simply a `QWidget` component. Even if, previously, we have evoked the example of a `QLineEdit`, the default delegate does not only create `QLineEdit`. Indeed, it creates the component adapted to the data type of the cell. For example, if the cell contains an `int`, the default delegate will create a `QSpinBox` component. For a `dual-` type component, it will be a `QDoubleSpinBox`.

Let's look at the `% QTDIR% / src / gui / itemviews / qstyleditemdelegate.cpp` code, including the `createEditor ()` method :

```
QVariant::Type t =
static_cast<QVariant::Type>(index.data(Qt::EditRole).userType());
return d->editorFactory()->createEditor(t, parent);
```

What are these two lines doing? First, the type of data contained in the cell is stored in the variable `t`. Remember, the `data ()` function of `QAbstractItemModel` returns a variable of type `QVariant`, this one is able to handle many types of data. The `userType ()` function returns this type as an `int` (which justifies the use of `static_cast`). If the variable is of type `QString`, `t` would be equal to `QVariant :: String`, `QVariant :: Date` for a variable of type `QDate`, etc.

The second line uses `QItemEditorFactory`'s `createEditor ()` method, with `editorFactory ()` defined as:

```
const QItemEditorFactory *editorFactory() const
{
    return factory ? factory : QItemEditorFactory::defaultFactory();
}
```

We understand that this method returns the default factory if none has been defined. But what is this famous factory? `QItemEditorFactory` is a so-called class factory (*factory*), allowing inter alia to create the publisher corresponding to the data type which was described above.

By default, here are the components created according to the type:

Type	widget
bool	QComboBox
int / unsigned	QSpinBox
double	QDoubleSpinBox
QDate	QDateEdit
QDateTime	QDateTimeEdit
QPixmap	QLabel
QString	QLineEdit
QTime	QTimeEdit

The default delegate is able to handle many of the data types. However, we may need behavior other than the default one. For that, it is enough to create his own delegate, that is what we will see in the following chapter.

## III. Create your own delegate

As seen in the previous part, we have three basic classes already provided by Qt:

- `QAbstractItemDelegate`;
- `QItemDelegate`;
- `QStyledItemDelegate`.

Since Qt 4.4, it is recommended to use `QStyledItemDelegate`. It will therefore be used as the basic class of our delegates.

When we only want to customize the editing of elements in our view and not the rendering, we must redefine four methods.

### III-A. `createEditor ()`

```
QWidget * QStyledItemDelegate::createEditor ( QWidget * parent, const
QStyleOptionViewItem & option,
                                         const QModelIndex & index )
const
```

This function returns the widget (editor) to edit the item at the `index index`. The `option` parameter controls how the widget appears.

This function will for example be called when the user will double-click on the cell of a `QTableView`. The editor returned by the function will then be presented to the user.

### III-B. `setEditorData ()`

```
void QStyledItemDelegate::setEditorData ( QWidget * editor, const QModelIndex
& index ) const
```

This function is used to pass to the editor `editor` the data to be displayed from the model at the `index index`.

Once the editor is created and opened, this function will be called in order to fill it with the data concerned from the model. The data will then be extracted from the model and displayed.

### III-C. `setModelData ()`

```
void QStyledItemDelegate::setModelData ( QWidget * editor, QAbstractItemModel
* model, const QModelIndex & index ) const
```

This function is, in a way, the opposite of the previous one. Its role is to retrieve the data from the editor and store it inside the model, at the index identified by the `index` parameter .

### III-D. updateEditorGeometry ()

```
void QStyledItemDelegate::updateEditorGeometry ( QWidget * editor, const
QStyleOptionViewItem & option,
const QModelIndex & index )
const
```

When the size of the view changes, this function resizes the editor to the correct size.

In general, the implementation will contain the following line of code:

```
editor->setGeometry(option.rect);
```

`option.rect` delimits the editor area.

If you want to customize the rendering of elements, you will have to redefine a fifth method.

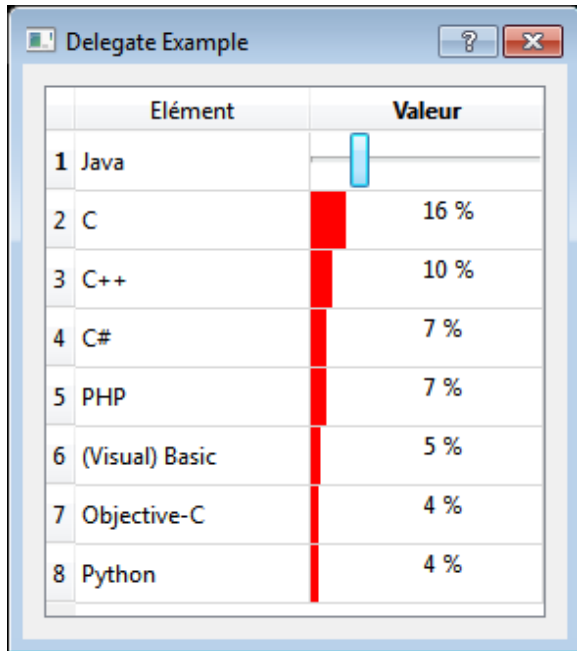
### III-E. paint ()

```
void QStyledItemDelegate::paint ( QPainter * painter, const
QStyleOptionViewItem & option,
const QModelIndex & index ) const
```

As the name suggests, this function renders a certain element using `painter` . Items are rendered using the current style of the view.

## IV. Let's go to practice

Now that we know how to create our own delegate, we will put what we have learned using an example. To avoid splitting up in a long speech, here's a screenshot of the application to make:



	Elément	Valeur
1	Java	
2	C	16 %
3	C++	10 %
4	C#	7 %
5	PHP	7 %
6	(Visual) Basic	5 %
7	Objective-C	4 %
8	Python	4 %

This is a table (QTableView) with two columns, showing an item on the left and an associated value on the right (a percentage). When you are not in edit mode, you draw a bar, representing the percentage. In addition, we use a QSlider to vary the percentage as we see fit.

## IV-A. The model

The first step will be to create your own model. We will not use QStandardItemModel here. The first thing to do is to think about what basic class we will inherit. Since the model must be presented as a table, it becomes obvious that we will inherit the QAbstractTableModel class.

### IV-A-1. The class

```
class TableModel : public QAbstractTableModel
{
    Q_OBJECT
public:
    enum Columns { Element = 0, Value, Count = Value + 1 };

    explicit TableModel(QObject *parent = 0);

    Qt::ItemFlags flags(const QModelIndex &index) const;
    int rowCount(const QModelIndex &parent = QModelIndex()) const;
    int columnCount(const QModelIndex &parent = QModelIndex()) const;

    QVariant data(const QModelIndex &index, int role = Qt::DisplayRole)
const;
```

```

    QVariant headerData(int section, Qt::Orientation orientation, int role =
Qt::DisplayRole) const;
    bool setData(const QModelIndex &index, const QVariant &value, int role =
Qt::DisplayRole);

    void addElement(const QString &element, int value);

private:
    QList<QPair<QString, int> > mElements;
};

```

For reasons of readability and flexibility, we define an enumeration `Columns` to list the different columns. `Count` is a small trick to get the number of elements and will serve for the `columnCount()` method.

The `Qt::ItemFlags flags method (const QModelIndex & index) const;` will be useful to decide which parts of our model will be editable or not (the column of elements will be read-only).

Items will be stored inside a `QPair` list. As a reminder, a pair is a set of two elements. The first element will be the `QString` text and the second will be the associated value (`int`). The `void addElement method (const QString & element, int value);` will add an element inside the model.

## IV-A-2. flags ()

```

Qt::ItemFlags TableModel::flags(const QModelIndex &index) const
{
    if (index.column() == Element)
    {
        return Qt::ItemIsEnabled | Qt::ItemIsSelectable;
    }
    else if (index.column() == Value)
    {
        return Qt::ItemIsEnabled | Qt::ItemIsSelectable | Qt::ItemIsEditable;
    }

    return QAbstractTableModel::flags(index);
}

```

Inside the function, the target column is checked using the `index column ()` method. If the column is the `Element` column, we return the `Qt::ItemIsEnabled | Qt::ItemIsSelectable`. This means our item will be enabled and selectable but not editable. If the column is the `Value` column, then the `Qt::ItemIsEditable` flag is added, allowing the user to edit the item.



If we are not in any of these configurations (which should not be the case here), we leave the responsibility to the base class to return the necessary flags.

### IV-A-3. rowCount () and columnCount ()

```
int TableModel::columnCount(const QModelIndex &parent) const
{
    return parent.isValid() ? 0 : Count;
}
```

If the `parent` index is valid (this model is not hierarchical), we return 0, otherwise `Count` (enumeration previously defined).

```
int TableModel::rowCount(const QModelIndex &parent) const
{
    return parent.isValid() ? 0 : mElements.count();
}
```

As before, we return 0 if parent is valid. Otherwise, the number of lines in the model is the number of elements in `mElements`: `mElements.count ()`.

### IV-A-4. data (), headerData () and setData ()

```
QVariant TableModel::data(const QModelIndex &index, int role) const
{
    if (!index.isValid() || index.row() < 0 || index.row() >=
mElements.count())
    {
        return QVariant();
    }

    switch (role)
    {
        case Qt::DisplayRole:
        case Qt::EditRole:
            if (index.column() == Element)
            {
                return mElements[index.row()].first;
            }
            else if (index.column() == Value)
            {
                return mElements[index.row()].second;
            }
            break;
    }
}
```

```

    }

    return QVariant();
}

```

We start by checking the validity of the index. If this is not the case, we return an invalid data: `QVariant ()`.

Then, depending on the role `role`, the corresponding data is returned. In this model, we take into account the `Qt::DisplayRole` and `Qt::EditRole` roles. If it is one of these two roles, return the first element of the pair corresponding to the index `index` if the column is `Element` or the second for the *Value* column.

For all the remaining roles, we return an invalid data.

```

QVariant TableModel::headerData(int section, Qt::Orientation orientation, int
role) const
{
    if (orientation == Qt::Horizontal && role == Qt::DisplayRole)
    {
        switch (section)
        {
            case Element:
                return trUtf8("Elément");
                break;
            case Value:
                return trUtf8("Valeur");
                break;
        }
    }

    return QAbstractTableModel::headerData(section, orientation, role);
}

```

This code should not need additional descriptions. If you are in the `Horizontal` orientation with the `DisplayRole` role, you return the text corresponding to the section (column).

```

bool TableModel::setData(const QModelIndex &index, const QVariant &value, int
role)
{
    if (!index.isValid() || index.row() < 0 || index.row() >=
mElements.count())
    {
        return false;
    }
}

```

```

switch (role)
{
case Qt::DisplayRole:
case Qt::EditRole:
    if (index.column() == Element)
    {
        mElements[index.row()].first = value.toString();
    }
    else if (index.column() == Value)
    {
        mElements[index.row()].second = value.toInt();
    }
    emit dataChanged(index, index);
    return true;
    break;
}

return false;
}

```

See the `data ()` function . It is also important to take care to emit the void signal `QAbstractItemModel :: dataChanged (const QModelIndex & topLeft, const QModelIndex & bottomRight)` , in order to notify the view that the data has changed.

#### IV-A-5. addElement ()

```

void TableModel::addElement(const QString &element, int value)
{
    const int count = mElements.count();
    if (value < 0)
    {
        value = 0;
    }
    else if (value > 100)
    {
        value = 100;
    }
    beginInsertRows(QModelIndex(), count, count);
    mElements << qMakePair(element, value);
    endInsertRows();
}

```

We start by retrieving the number of current elements and then we make sure that `value` is between 0 and 100. The void function `QAbstractItemModel :: beginInsertRows (const QModelIndex & parent, int first, int last)` allows to start an operation of insertion of line in

a model (it is necessary to call it). After adding the element inside the list using the `QPair` utility function `<T1, T2> qMakePair (const T1 & value1, const T2 & value2)`, we end the insert operation with `endInsertRows ()`.

## IV-A-6. Exercise

Using the documentation, edit the template so that the items in the first column are aligned centrally.

First clue: the function to modify is `data ()`.

Second clue: the role to manage is `Qt :: TextAlignmentRole`.

The Solution:

```
case Qt::TextAlignmentRole:
    if (index.column() == Element)
    {
        return Qt::AlignCenter;
    }
    break;
```

## IV-B. The delegate

The model is now finished and ready to use, we will tackle the delegate, which will allow us to edit the values with a cursor and draw a bar outside the edit mode.

### IV-B-1. The class

```
class TableDelegate : public QStyledItemDelegate
{
    Q_OBJECT
public:
    explicit TableDelegate(QObject *parent = 0);
    QWidget *createEditor(QWidget *parent, const QStyleOptionViewItem
&option, const QModelIndex &index) const;
    void setEditorData(QWidget *editor, const QModelIndex &index) const;
    void setModelData(QWidget *editor, QAbstractItemModel *model, const
QModelIndex &index) const;

    void paint(QPainter *painter, const QStyleOptionViewItem &option, const
QModelIndex &index) const;
    void updateEditorGeometry(QWidget *editor, const QStyleOptionViewItem
&option, const QModelIndex &index) const;
};
```

No surprises here. We create a `TableDelegate` class inheriting from `QStyledItemDelegate` and we redefine the necessary methods (see above).

#### IV-B-2. `createEditor ()`

```
QWidget *TableDelegate::createEditor(QWidget *parent, const
QStyleOptionViewItem &option,
                                   const QModelIndex &index) const
{
    if (index.column() == TableModel::Value)
    {
        QSlider *editor = new QSlider(Qt::Horizontal, parent);
        editor->setRange(0, 100);
        editor->setAutoFillBackground(true);
        return editor;
    }

    return QStyledItemDelegate::createEditor(parent, option, index);
}
```

As usual, we act according to the column ( `column ()` ). In the case of the Value column, we create a horizontal `QSlider`. It is assigned a range of values from 0 to 100 and sets the `autoFillBackground` property to `true` to get an opaque background. Otherwise, the parent class is left with the responsibility of creating the publisher.

#### IV-B-3. `setEditorData ()` and `setModelData ()`

```
void TableDelegate::setEditorData(QWidget *editor, const QModelIndex &index)
const
{
    if (index.column() == TableModel::Value)
    {
        QSlider *slider = qobject_cast<QSlider *>(editor);
        if (slider)
        {
            const int value = index.model()->data(index).toInt();
            slider->setValue(value);
        }
    }
    else
    {
        QStyledItemDelegate::setEditorData(editor, index);
    }
}
```

In order to correctly initialize the cursor, we retrieve the corresponding value in the model at the index `index` (which we do not forget to convert to `int`, the `data ()` function returning a `QVariant`) and then we assign it to the cursor by calling the `setValue ()` function. Since the `editor` parameter is of type `QWidget`, we do not forget to convert it to `QSlider` using `qobject_cast`.

```
void TableDelegate::setModelData(QWidget *editor, QAbstractItemModel *model,
const QModelIndex &index) const
{
    if (index.column() == TableModel::Value)
    {
        QSlider *slider = qobject_cast<QSlider *>(editor);
        if (editor)
        {
            model->setData(index, slider->value());
        }
    }
    else
    {
        QStyledItemDelegate::setModelData(editor, model, index);
    }
}
```

Here the reverse operation is carried out. At the end of editing, we retrieve the value of the cursor and assign it to our model index `index`. This makes it possible to update the model once the edition is finished and the editor closed.

#### IV-B-4. updateEditorGeometry ()

```
void TableDelegate::updateEditorGeometry(QWidget *editor, const
QStyleOptionViewItem &option,
                                         const QModelIndex &index) const
{
    Q_UNUSED(index);
    editor->setGeometry(option.rect);
}
```

As agreed, we update the geometry (position and size) of the editor via the information of the `option` parameter. The `Q_UNUSED` macro makes it possible to mark the `index` parameter as unused, so as not to get a warning during the compilation (`unused parameter`).

#### IV-B-5. paint ()

```

void TableDelegate::paint(QPainter *painter, const QStyleOptionViewItem
&option, const QModelIndex &index) const
{
    if (index.column() == TableModel::Value)
    {
        painter->save();
        const int value = index.model()->data(index).toInt();
        QRect rect(option.rect);
        const int width = (value * rect.width()) / 100;
        rect.setWidth(width);
        QColor c;
        if (value <= 20)
        {
            c = Qt::red;
        }
        else if (value <= 50)
        {
            c = QColor(240, 96, 0);
        }
        else
        {
            c = Qt::green;
        }

        painter->fillRect(rect, c);
        QTextOption o;
        o.setAlignment(Qt::AlignCenter);
        painter->drawText(option.rect, QString("%1 %").arg(value), o);

        painter->restore();
    }
    else
    {
        QStyledItemDelegate::paint(painter, option, index);
    }
}

```

Finally, the part returned. Wishing only to act on the value column, we first check if we are in this case. Otherwise, the parent class is left with the responsibility of rendering.

We start by registering (saving) the current state of the painter. We recover the value located at the corresponding index to calculate the effective width of the bar with a simple cross product. We thus obtain a QRect rectangle. We take the opportunity to assign a different color according to the corresponding value:

- **red** if the value is less than or equal to 20;
- **orange** if the value is less than or equal to 50;

- `green` otherwise.

With the rectangle and its color, use the `fillRect ()` function to draw and fill the rectangle with the corresponding color. Finally, we draw the text corresponding to the value followed by the percentage, centrally, before restoring the state of the previously saved painter.

This concludes the program. It is left to the reader to complete it by creating the view, the model and the delegate before filling the model with the desired values, in order to have a complete and operational program.

## V. Conclusion

At the end of this tutorial, you will have learned:

- Delegates: Components to customize the editing and rendering of elements within a view
- how to create your own delegate.

On the other hand, we have seen, through a simple example, how to create from scratch his own model. For another example of use, a little more concrete, the documentation also provides a [Star Delegate Example](#) .