

## **Curs 9 - 10**

- **QtDesigner – creare de interfete grafice folosind drag&drop**
- **Componente grafice QT cu model**
- **Interfete grafice utilizator**
  - **Prezentare de volume mari de date**
  - **Sincronizare model prezentare**
  - **Vederi multiple**

## QtDesigner din Eclipse

### Proiect Eclipse Qt GUI

File ->New->Qt GUI Project

- generează structura proiectului qt (setează modulele incluse, directoare , etc)
- .ui – fisier ce contine descrierea interfeței grafice
  - UIC (user interface copiller) utilitar ce transformă fișierul .ui in fișier c++ care construiește interfața grafică (ui\_<name>.h)
- crează o componenta GUI component, o clasă (.h, .cpp) - extinde QWidget sau altă clasă derivată din Qwidget (QDialog, QMainWindow). Aici putem adăuga sloturi și semnale noi
- main.cpp

```
int main(int argc, char *argv[])
{
    QApplication a(argc, argv);
    ProductRep w;
    w.show();
    return a.exec();
}
```

Important – Versiunea curenta de Qt eclipse plugin are probleme când folosim anumite clase din biblioteca standard c++ (cout,iostream, etc). Compilatorul interactiv raportează erori în mod greșit.

Rezolvare - Deactivați analizorul din timpul editarii - Code Analysis (Project->Properties->c/c++ general->Code Analyses deselectați toate).

Erorile sunt raportate corect după ce compilați proiectul (nu în timp ce scrieți codul)

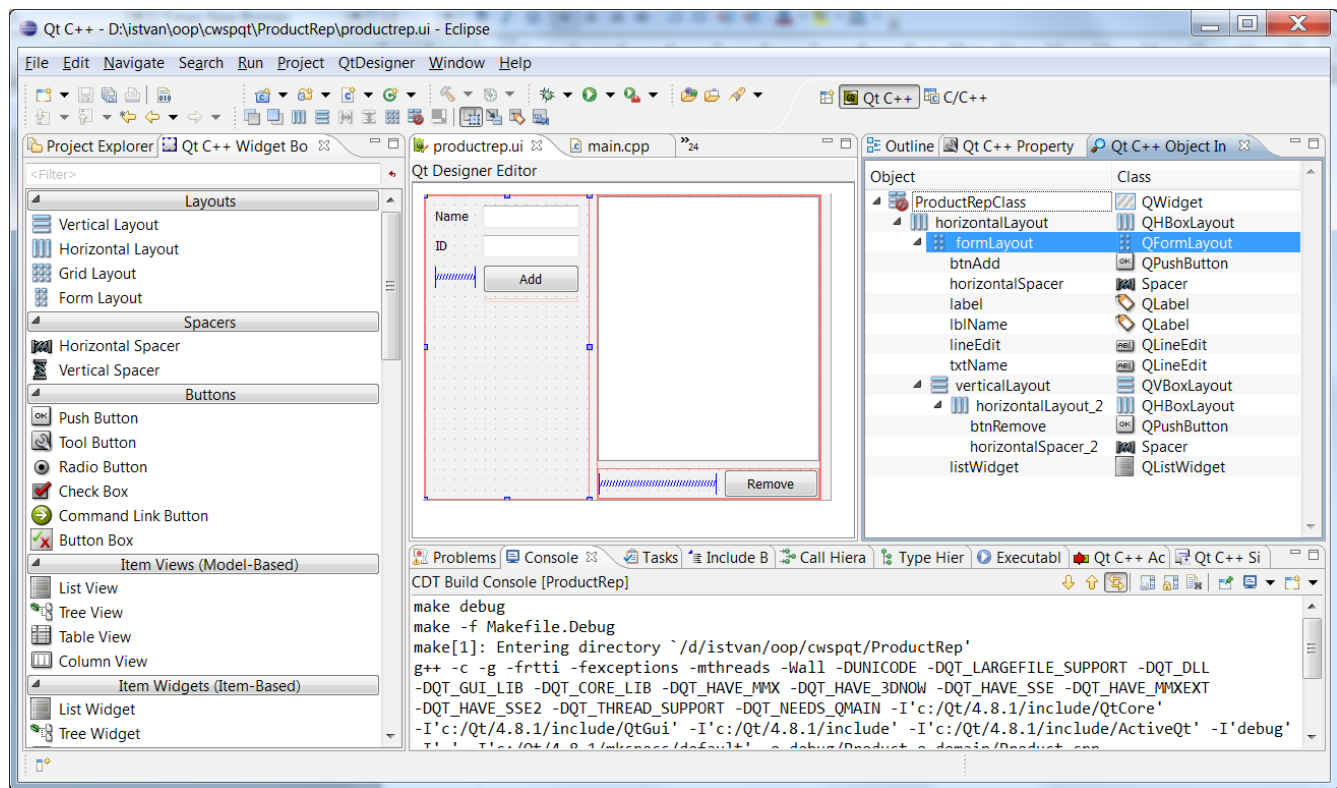
## Creare de interfețe grafice vizual (folosind drag & drop)

Pluginul Eclipse pentru Qt permite crearea de interfețe grafice în mod vizual (fără să scriem cod)

- nu este neobișnuit pentru un programator Qt să creeze aplicații exclusiv scriind cod
- dar, varianta vizuală poate fi mai rapidă în anumite situații
- permite experimentarea rapidă cu diferite variante de interfață grafică

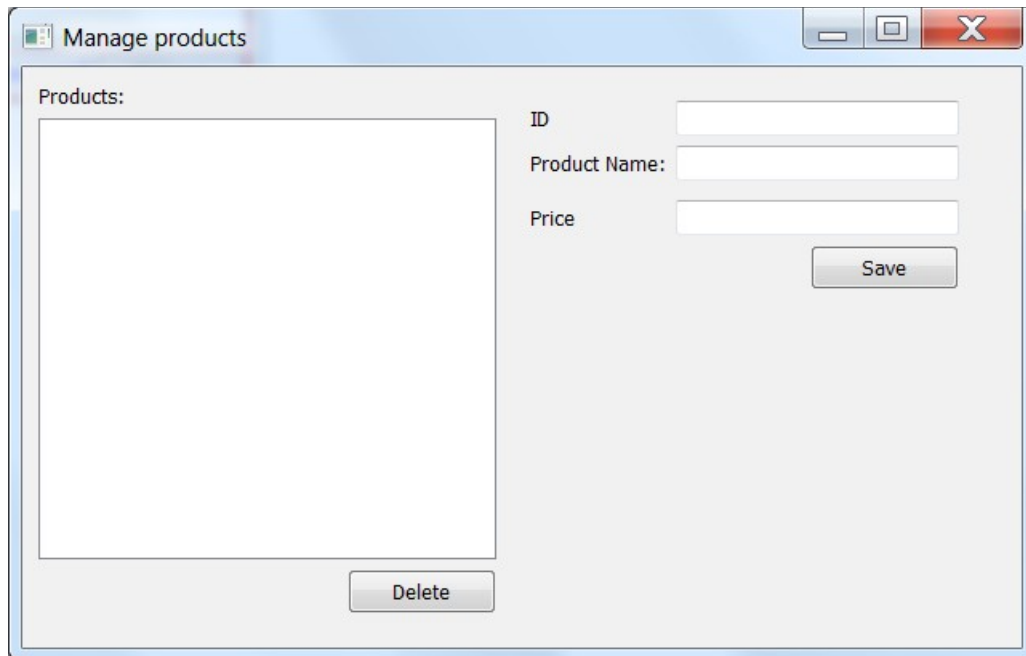
Eclipse Qt editor/views:

- Qt Designer editor – permite crearea de GUI (aranjare componente grafice)
- Qt C++ Widget Box - expune componente Qt care se pot adăuga pe fereastră
- Qt C++ Object Inspector – prezintă organizarea componentelor (componente fii)
- Qt C++ Property Editor – editare de proprietăți pentru componentele adăugate pe fereastră



## Master detail – Product

CRUD (Create Read Update Delete) pentru Product



The screenshot shows a window titled "Manage products" with standard Windows window controls (minimize, maximize, close). The window is divided into two main sections. On the left, under the label "Products:", there is a large, empty rectangular box intended for displaying a list of products. Below this box is a "Delete" button. On the right side of the window, there are three input fields for product details: "ID", "Product Name:", and "Price". Each field is represented by a white rectangular box. Below these input fields is a "Save" button. The overall layout is a typical master-detail form used for managing data in a database application.

## Sloturi definite de utilizator

<pre>class testSlots: public QWidget { Q_OBJECT  public:     testSlots(Warehouse* wh, QWidget *parent = 0);     ~testSlots();  private:     Ui::testSlotsClass ui;     Warehouse* wh;     void connectSS();     void reloadList();  private slots:     void save();     void productSelected(); };</pre>	<pre>/**  * Save products  */ void testSlots::save() {     int id = ui.txtID-&gt;text().toInt();     double price = ui.txtPrice-&gt;text().toDouble();     string desc = ui.txtName-&gt;text().toStdString();     try {         wh-&gt;addProduct(id, desc, price);         reloadList();         QMessageBox::information(this, "Info", "Product saved...");     } catch (WarehouseException ex) {         QMessageBox::critical(this, "Error", QString::fromStdString(ex.getMsg()));     } }</pre>
--	--

## QString

- Clasă pentru șiruri de caractere (Unicode ), similar cu string din biblioteca standard c++
- apare frecvent când lucrăm cu componente din Qt

### Create QString

```
QString s1 = "Hello";  
QString s2("World");
```

### QString and string (STL)

```
string str = "Hello";  
QString qStr = QString::fromStdString(str);  
string str2 = qStr.toStdString();
```

### Numbers and QString

```
QString s3 = QString::number(2);  
QString s4 = QString::number(2.5);  
QString s5 = "2";  
int i = s5.toInt();  
double d = s5.toDouble();
```

## QListWidget – populare listă, semnalul itemSelectionChanged()

```
private slots:
    void save();
    void productSelected();

/**
 * Save product
 */
void testSlots::save() {
    int id = ui.txtID->text().toInt();
    double price = ui.txtPrice->text().toDouble();
    try {
        wh->addProduct(id, ui.txtName->text().toStdString(), price);
        reloadList();
        QMessageBox::information(this, "Info", "Product saved...");
    } catch (WarehouseException ex) {
        QMessageBox::critical(this, "Error",
                               QString::fromStdString(ex.getMsg()));
    }
}

/**
 * Load the products into the list
 */
void testSlots::reloadList() {
    ui.lstProducts->clear();
    DynamicArray<Product*> all = wh->getAll();
    for (int i = 0; i < wh->getNrProducts(); i++) {
        string desc = all.get(i)->getDescription();
        QListWidgetItem *item = new QListWidgetItem(
            QString::fromStdString(desc), ui.lstProducts);
        item->setData(Qt::UserRole, QVariant::fromValue(all.get(i)-
>getCode()));
    }
}
```

## **Clasele Qt ItemView**

QListWidget, QTableWidget , QTreeWidget

Componentele se populează, adaugând toate elementele de la început (items: QListWidgetItem, QTableWidgetItem, QTreeWidgetItem).

Afișarea, cautarea, editarea sunt efectuate direct asupra datelor cu care este populat componenta

Datele care se modifică trebuie sincronizate, actualizat sursa de unde au fost încărcate (fișier, bază de date, rețea, etc)

Avantaje:

- simplu de înțeles
- simplu de folosit

Dezavantaje:

- nu poate fi folosit dacă avem volume mari de date
- este greu de lucrat cu multiple vederi asupra aceluiași date
- necesită duplicare de date



## **Model-View-Controller**

Abordare flexibilă pentru vizualizare de volume mari de date

**model:** reprezintă setul de date responsabil cu:

- încarcă datele necesare pentru vizualizare
- scrie modificările înapoi la sursă

**view:** prezintă datele utilizatorului.

- Chiar dacă avem un volum mare de date, doar o porțiune mică este vizibilă la un moment dat. View este responsabil să ceară doar datele care sunt necesare pentru vizualizare (nu toate datele)

the **controller:** mediează între model și view

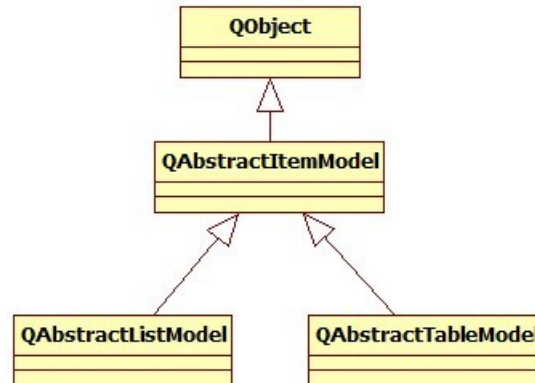
- transformă acțiunile utilizator în cereri (de navigare, de editare date)
- diferit de GRASP Controller

## Model/View în Qt

- Separarea datelor de prezentare (views)
- permite vizualizarea de volume mari de date, date complexe , are integrat lucrul cu baze de date , vederi multiple asupra datelor
- Qt 4 > oferă un set de clase model/view (list, table, tree)
- Arhitectura Model/View din Qt este inspirat din șablonul MVC (Model-View-Controller), but dar în loc de controller, Qt folosește o altă abstractizare numită **delegate**
- **delegate** – oferă control asupra modului de prezentare a datelor și asupra editării
- Qt oferă implementari default pentru delegate pentru toate tipurile de vederi (listă, tabel, tree,etc.) - în general este suficient
- Qt Item Views : QListView, QTableView, QTreeView și clase model asociate

## Creare de modele noi

- Se crează o nouă clasă pentru model (model de listă, model de tabel)
- se extinde o clasă existentă din Qt



QAbstractItemModel – clasă model pentru orice clasă Qt Item View .

Poate conține orice fel de date tabelare (row, columns) sau ierarhice (structură de tree )

Datele sunt expuse ca și un tree unde nodurile sunt tabele

Fiecare item are atasat un numar de elemente cu roluri diferite (DisplayRole, BackgroundRole, UserRole, etc)

```
void testSlots::reloadList() {
    ui.lstProducts->clear();
    DynamicArray<Product*> all = wh->getAll();
    for (int i = 0; i < wh->getNrProducts(); i++) {
        string desc = all.get(i)->getDescription();
        QListWidgetItem *item = new QListWidgetItem(
            QString::fromStdString(desc), ui.lstProducts);
        item->setData(Qt::UserRole, QVariant::fromValue(all.get(i)->getCode()));
    }
}

/**
 * Load the selected product into the detail panel
 */
void testSlots::productSelected() {
    QList<QListWidgetItem*> sel = ui.lstProducts->selectedItems();
    if (sel.size() == 0) {
        return;
    }
    QVariant idV = sel.first()->data(Qt::UserRole);
    int id = idV.toInt();
    const Product *p = wh->getByCode(id);
    ui.txtID->setText(QString::number(id));
    ui.txtName->setText(QString::fromStdString(p->getDescription()));
    ui.txtPrice->setText(QString::number(p->getPrice()));
    QMessageBox::information(this, "Info", "Product selected...");
}
```

## Creare de modele noi

```
class MyTableModel: public QAbstractTableModel {
public:
    MyTableModel(QObject *parent);
    /**
     * number of rows
     */
    int rowCount(const QModelIndex &parent = QModelIndex()) const;
    /**
     * number of columns
     */
    int columnCount(const QModelIndex &parent = QModelIndex()) const;
    /**
     * Value at a given position
     */
    QVariant data(const QModelIndex &index, int role = Qt::DisplayRole) const;
};

MyTableModel::MyTableModel(QObject *parent) :
    QAbstractTableModel(parent) {
}

int MyTableModel::rowCount(const QModelIndex & /*parent*/) const {
    return 100;
}

int MyTableModel::columnCount(const QModelIndex & /*parent*/) const {
    return 2;
}

QVariant MyTableModel::data(const QModelIndex &index, int role) const {
    if (role == Qt::DisplayRole) {
        return QString("Row%1, Column%2").arg(index.row() + 1).arg(
            index.column() + 1);
    }
    return QVariant();
}
```

Putem crea modele care încarcă doar datele care sunt efectiv necesare (sunt vizibile)

## Modele predefinite

Qt oferă modele predefinite:

- **QStringListModel** – Lucrează cu o listă de stringuri
- **StandardItemModel** - Date ierarhice
- **QDirModel** - System de fişiere
- **SqlQueryModel** - SQL result set
- **SqlTableModel** - SQL table
- **SqlRelationalTableModel** - SQL table cu chei străine
- **SortFilterProxyModel** - oferă sortare/filtrare

```
void createTree() {  
    QTreeView *tV = new QTreeView();  
    QDirModel *model = new QDirModel();  
    tV->setModel(model);  
    tV->show();  
}
```

## Modificare attribute legate de prezentarea datelor

enum Qt::ItemDataRole	Meaning	Type
Qt::DisplayRole	text	QString
Qt::FontRole	font	QFont
<u>Qt::BackgroundRole</u>	brush for the background of the cell	QBrush
Qt::TextAlignmentRole	text alignment	enum Qt::AlignmentFlag
Qt::CheckStateRole	suppresses checkboxes with QVariant(),  sets checkboxes with Qt::Checked or Qt::Unchecked	enum Qt::ItemDataRole

```
QVariant MyTableModel::data(const QModelIndex &index, int role) const {
    int row = index.row();
    int column = index.column();
    if (role == Qt::DisplayRole) {
        return QString("Row%1, Column%2").arg(row + 1).arg(column + 1);
    }
    if (role == Qt::FontRole) {
        QFont f;
        f.setItalic(row % 4 == 1);
        f.setBold(row % 2 == 1);
        return f;
    }
    if (role == Qt::BackgroundRole) {
        if (column == 1 && row % 2 == 0) {
            QBrush bg(Qt::red);
            return bg;
        }
    }
    return QVariant();
}
```

## Cap de tabel (Table headers)

- Modelul controlează și capul de tabel ( header de coloane, rânduri) pentru tabel
- Suprascrim metoda `QVariant headerData(int section, Qt::Orientation orientation, int role)`

```
QVariant MyTableModel::headerData(int section, Qt::Orientation
orientation,
    int role) const {
    if (role == Qt::DisplayRole) {
        if (orientation == Qt::Horizontal) {
            return QString("col %1").arg(section);
        } else {
            return QString("row %1").arg(section);
        }
    }
    return QVariant();
}
```

## Sincronizare model și prezentare

Dacă se schimbă datele (modelul) trebuie să se schimbe și prezentarea (view)

View este conectat (automat, în metoda view.setModel) la semnalul **dataChanged** .

Dacă se schimbă ceva în model trebuie să emit semnalul dataChanged și să se actualizeze interfața grafică

```
/**
 * Slot invoked by the timer
 */
void MyTableModel::timerTikTak() {
    QModelIndex topLeft = createIndex(0, 0);
    QModelIndex bottomRight = createIndex(rowCount(), columnCount());
    emit dataChanged(topLeft, bottomRight);
}
```



## Vederi multiple pentru același date

Putem avea multiple vederi asupra acelorași date, astfel permițând diferite tipuri de interacțiuni cu data

Folosind mecanismul de semnale și sloturi modificările în model se vor reflecta în toate vederile asociate

```
QTableView* tV = new QTableView();
MyTableModel *model = new MyTableModel(tV);
tV->setModel(model);
tV->show();

QListView *tVT = new QListView();
tVT->setModel(model);
tVT->show();
```

## Editare/modificare valori

Se suprascrie metodele:

```
bool MyTableModel::setData(const QModelIndex & index, const QVariant & value, int role)
```

```
Qt::ItemFlags MyTableModel::flags(const QModelIndex & /*index*/)
```

```
/**
 * Invoked on edit
 */
bool MyTableModel::setData(const QModelIndex & index, const QVariant & value,
    int role) {
    if (role == Qt::EditRole) {
        int row = index.row();
        int column = index.column();
        //save value from editor to member m_gridData
        m_gridData[index.row()][index.column()] = value.toString();
        //make sure the dataChange signal is emitted so all the views will be
notified
        QModelIndex topLeft = createIndex(row, column);
        emit dataChanged(topLeft, topLeft);
    }
    return true;
}

Qt::ItemFlags MyTableModel::flags(const QModelIndex & /*index*/) const {
    return Qt::ItemIsSelectable | Qt::ItemIsEditable | Qt::ItemIsEnabled;
}
```

Când schimbam modelul trebuie să emitem semnalul dataChanged (să ne asigurăm că vederile se actualizează)