Curs 11

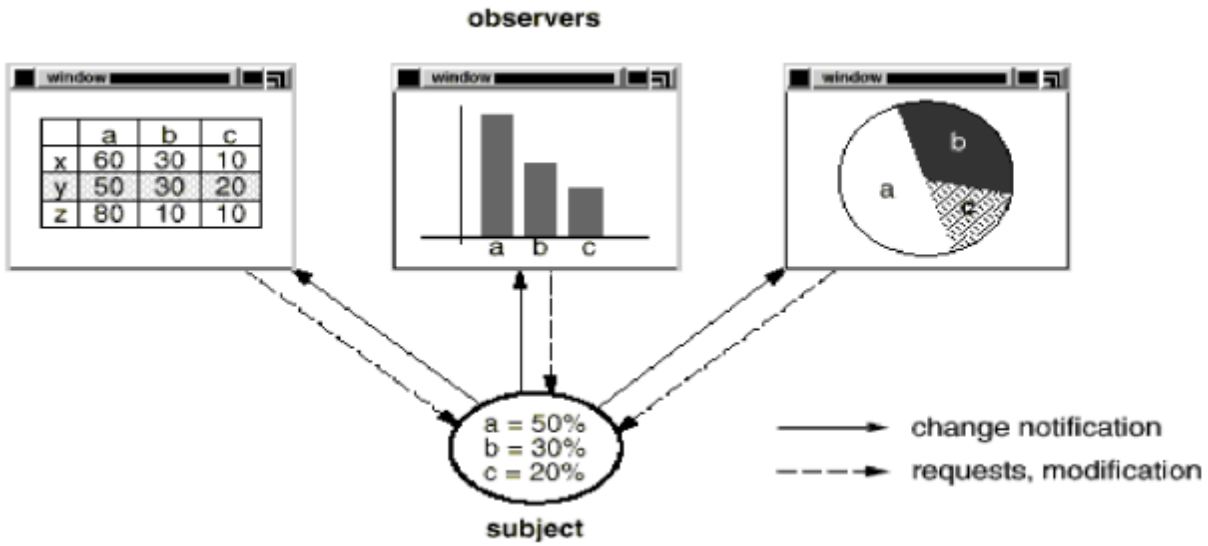
- **Interfețe grafice utilizator**
  - **Șablonul observer**
  - **Prezentare de volume mari de date**
  - **Model View Controller**

Curs 9-10 – Interfețe grafice utilizator

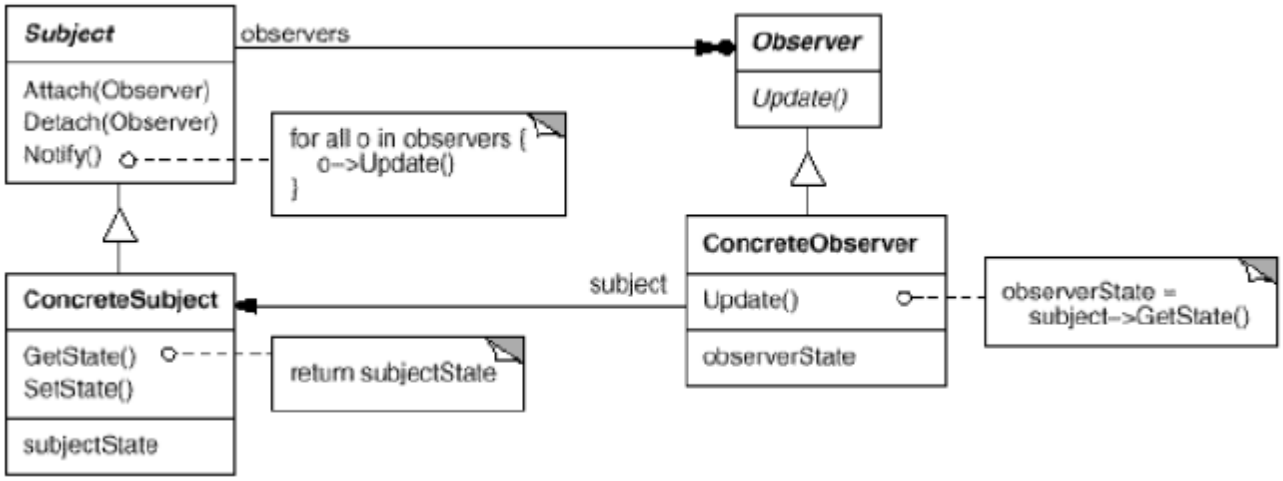
- **Semnale și sloturi**
- **Componente definite de utilizator**
- **Callback/Observer**
- **Evenimente de mouse/tastatura**
- **Graphics View Framework**

Sablonul Observer (Observer Design pattern)

Intent : Definește o relație de dependenta one-to-many intre obiecte astfel încât in momentul in care obiectul schimba starea toate obiectele dependente sunt notificate automat
Also Known As: Publish-Subscribe*
Motivation: O consecința a partiționării sistemului in clase care cooperează este ca apare nevoia de a menține consistenta intre obiecte. Scopul este sa menținem consistenta dar in același timp sa evitam cuplarea intre obiecte (cuplarea reduce reutilizabilitatea).



Patten class structure



Observer – cod c++

```
/* Update method needs to be implemented by observers
  Alternative names: Listener */
class Observer {
public:
      /* Invoked when Observable change
      Alternative names:properyChanged      */
      virtual void update() = 0;
};
/* Derive from this class if you want to provide notifications
  Alternative names: Subject, ChangeNotifier */
class Observable {
private:
      /*Non owning pointers to observer objects*/
      std::vector<Observer*> observers;
public:
      /* Observers use this method to register for notification
        Alternative names: attach, register, subscribe, addListener */
      void addObserver(Observer *obs) {
            observers.push_back(obs);
      }
      /* Observers use this to cancel the registration
      !!!Before an observer is destroyed need to cancel the registration
     Alternative names: detach, unregister, removeListener */
      void removeObserver(Observer *obs) {
        observers.erase(std::remove(begin(observers), end(observers),obs),
                        observers.end());
      }
protected:
      /* Invoked by the observable object
         in order to notify interested observer */
      void notify() {
            for (auto obs : observers) {
                  obs->update();
            }
      }
};
```

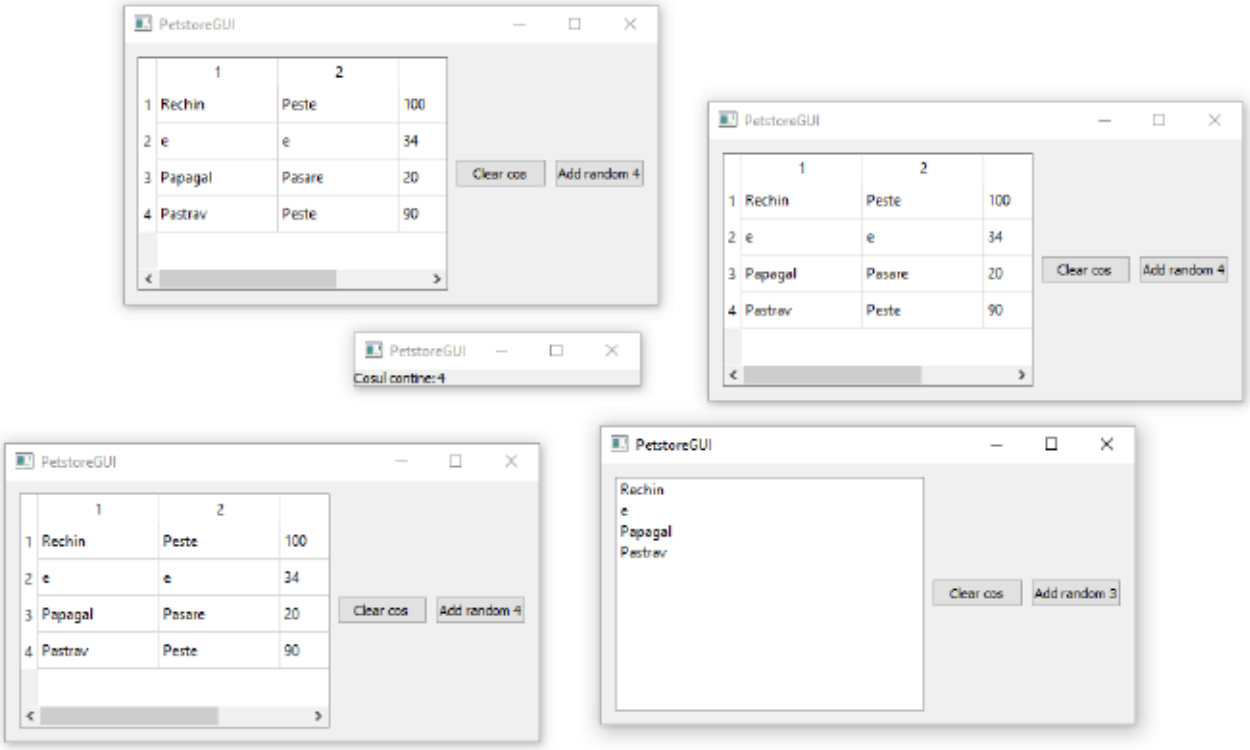
```
class InterestedObj : public Observer {
public:
  void update() override{
    std::cout << "Notified\n";
  }
};
```

```
class ConcreteSubject: public Observable {
public:
      void doStuff() {
            //...
            notify();//inherited
      }
};
```

Exemplu: cos de cumpărături

Problema: sa avem multiple ferestre (de diferite tipuri) care prezinta același cos cu animale. Fiecare fereastra trebuie actualizata in momentul in care se schimba conținutul coșului.

Scop: gestiunea dependentelor – sa decuplam clasele intre ele



Clasa Cos extinde **Observable** – conține lista de Pet din cos, notifica orice modificare in lista

Clasele CosTableGUI, CosListGUI, CosLabelGUI - extind Observer si se înscriu pentru notificare (addObserver)

Folosind șablonul Observer obținem:

- Clasa Cos nu este dependent de clasele GUI
- Clasele GUI nu depind una de cealaltă
- Se pot adăuga cu ușurința noi clase GUI

Șablonul Observer – variante

Pull vs Push

`class Observer {`
`public:`
`  virtual void update() = 0;`
 	
`};`
	`class Observer {`
	`public:`
	`virtual void update(int changedState)=0;`
	`};`

Pull: cel care este notificat trebuie sa obțină datele. In general obiectul interesat are referința la subiect (așa obține informațiile dorite)	Push: cel care este notificat primește informații despre ce s-a schimbat. Obiectul interesat nu are nevoie de referință la subiect. Exista si varianta in care se primeste subiectul ca parametru la update

Notificări diferite in funcție de ce s-a schimbat in obiectul subiect

`class Observer {`
`public:`
`virtual void itemAdded()=0;`
`virtual void itemRemoved()=0;`
`virtual void itemUpdated()=0;`
`};`
	`class Observer {`
	`public:`
	`virtual void itemAdded(Item item)=0;`
	`virtual void itemRemoved(Item item)=0;`
	`virtual void itemUpdated(Item item)=0;`
	`};`

Obiectul interesat trebuie sa implementeze toate metodele pure. Poate reacționa diferit in funcție de ce schimbări au apărut	

Observer Varianta Qt – semnale, sloturi/lambda, QObject::connect, emit

addObserver -> `QObject::connect` **removeObserver** -> `QObject::disconnect`
Elimina nevoia de a implementa o anume interfața (extinde Observer, Observable) Funcționează atât cu varianta Pull cat si Push (daca un semnal trimite si valori acestea se vor primi ca si parametru la slot sau lambda care s-a connectat la semnal Sursa semnalului si slotul (codul care se executa când apare semnalul) sunt total independente Se bazează pe generare de cod C++ (moc compiler)

Publisher/Subscriber - Varianta folosita in general in cazul in care dorim notificări intre procese (nu in același aplicație).

Clasele Qt ItemView

QListWidget, QTableWidget , QTreeWidget

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

Afișarea, căutarea, 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 daca 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 sa ceară doar datele care sunt necesare pentru vizualizare (nu toate datele)

controller: mediază 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), dar în loc de controller, Qt folosește o altă abstractizare numită **delegate**

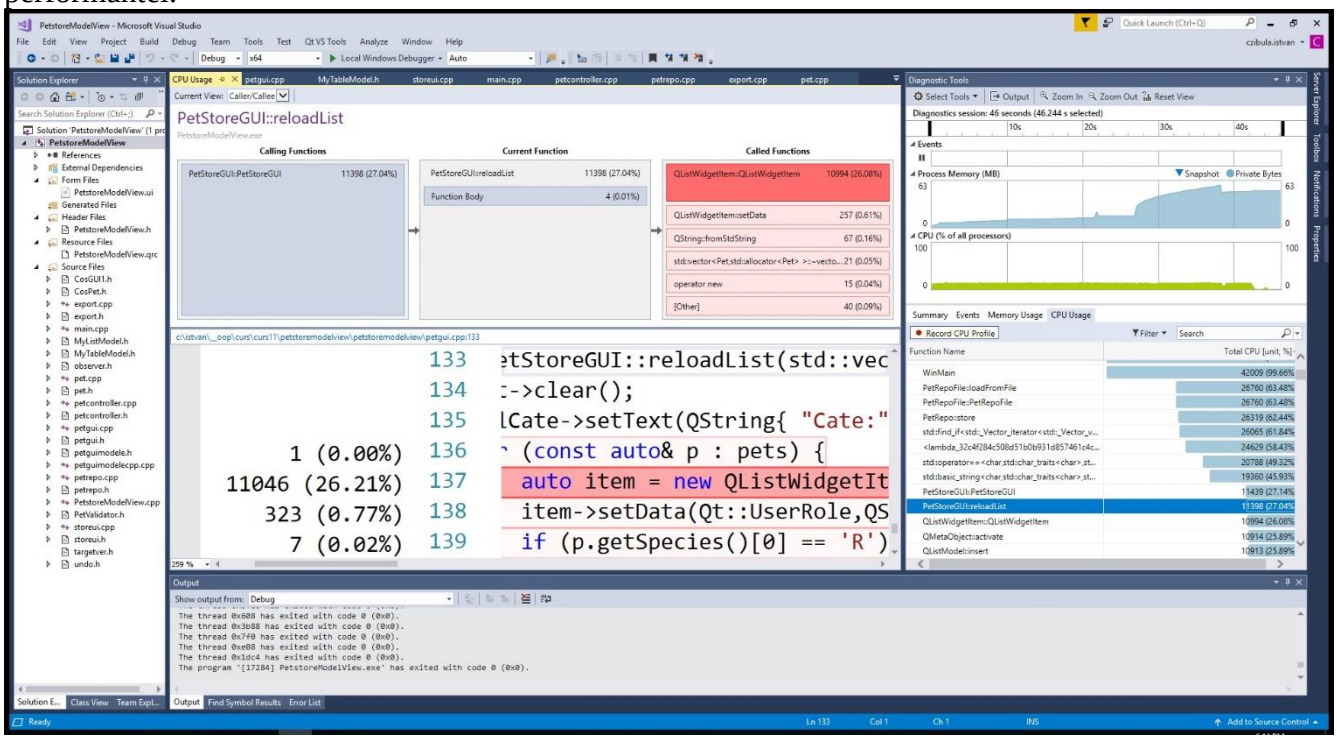
- **delegate** – oferă control asupra modului de prezentare a datelor și asupra editării

- Qt oferă implementări 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

Analiza performatei – Profiling

Instrumentele de tip Profiler facilitează analiza performantei sistemului (consum de memorie, timp de execuție).
In cazul in care avem probleme de performanta (anumite operații merg prea încet) putem folosi un profiler pentru a obține detalii despre care parte a aplicației contribuie la degradarea performantei.



Obs:

- Înainte de a optimiza codul sa ne asiguram ca este nevoie de optimizare.
- Regula de aur pentru îmbunătățirea performatei este măsurarea. Fără a măsura nu putem fi sigur ca am îmbunătățit ceva.
- In cazul C++ important sa compilam cu opțiunile de optimizare activate (release build, -O3) – nu folosiți debug build pentru a măsura timpul de execuție.

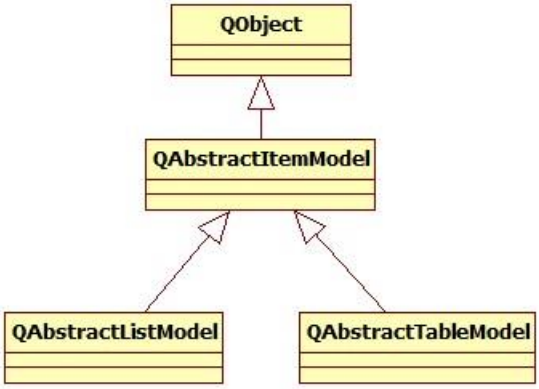
CPU Profiling: putem vedea care metoda si cu cat contribuie la timpul de execuție total.
Memory profiling: putem urmari numarul de obiecte create/distruse, metodele responsabile de crearea de obiecte, putem detecta memory leak si identifica cauza

.

Profiler-ul foloseste **sampling** (verifica din când in când care metoda/instrucțiune este executat) sau **instrumentare** (adaugă cod special in aplicație pentru măsurători)

Creare de modele noi

- Se creează 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 atașat un număr de elemente cu roluri diferite (DisplayRole, BackgroundRole, UserRole, etc)

Creare de modele noi

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

};
```

```
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
- **QStandardItemModel** - Date ierarhice
- **QDirModel** - System de fișiere
- **QSqlQueryModel** - SQL result set
- **QSqlTableModel** - SQL table
- **QSqlRelationalTableModel** - SQL table cu chei străine
- **QSortFilterProxyModel** - oferă sortare/filtrare

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

Modificare atribute legate de prezentarea datelor

enum Qt::ItemDataRole	Meaning	Type
Qt::DisplayRole	text	QString
Qt::FontRole	font	QFont
Qt::BackgroundRole	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();
}
```

Selectie in QListView/QTableView

```
lst->setModel(model); //Obs. Inainte de connect trebuie setat modelul
```

```
QObject::connect(lst->selectionModel(), &QItemSelectionModel::selectionChanged, [&]() {
		btnAddToCos->setEnabled(!lst->selectionModel()->selectedIndexes().isEmpty());
	});
```

```
QObject::connect(lst->selectionModel(), &QItemSelectionModel::selectionChanged, [&]() {
		if (lst->selectionModel()->selectedIndexes().isEmpty()) {
			//nu este nimic selectat (golesc detaliile)
			return;
		}
		auto selIndex = lst->selectionModel()->selectedIndexes().at(0);
		//putem lua date din lista
		QString species = selIndex.data(Qt::DisplayRole).toString();
		QString type = selIndex.data(Qt::UserRole).toString();
	});
```

```
//selectia in tabel
QObject::connect(tblV->selectionModel(), &QItemSelectionModel::selectionChanged, [this]() {
		if (tblV->selectionModel()->selectedIndexes().isEmpty()) {
			txtSpecies->setText("");
			return;
		}
		int selRow = tblV->selectionModel()->selectedIndexes().at(0).row();
		auto cel0Index = tblV->model()->index(selRow, 0);
		auto cel1Index = tblV->model()->index(selRow, 1);
		auto cellValue= tblV->model()->data(cel0Index, Qt::DisplayRole).toString();
		txtSpecies->setText(cellValue);
	});
```

Cap de tabel (Table headers)

- Modelul controlează și capul de tabel (header de coloane, rânduri) pentru tabel
- Suprascriem 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 sa emitem semnalul dataChanged și se actualizează 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)

QtDesigner

Proiect Qt in Visual Studio

- generează structura proiectului qt (setează modulele incluse, directoare , etc)
- .ui – fișier ce conține descrierea interfeței grafice
  - UIC (user interface copiler) utilitar ce transformă fișierul .ui in fișier c++ care construieste 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();
}
```

Creare de interfețe grafice vizual (folosind drag & drop)

Qt Designer permite creare 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

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