Curs 7 – Interfețe grafice utilizator

- **Semnale și sloturi**
- **Componente definite de utilizator**
- **Meta object compiler**

Semnale și sloturi (Signals and slots)

- Samnalele și sloturile sunt folosite in Qt pentru comunicare între obiecte

- Este mecanismul central în Qt pentru crearea de interfețe utilizator

- Mecanismul este specific Qt, diferă de mecanismele folosite în alte biblioteci de GUI.

- Cand facem modificări la o componentă (scriem un text, apasam butonul, selectăm un element,etc.) dorim ca alte părți ale aplicației să fie notificate (să actualizăm alte compnente, să executăm o metodă, etc).
  Ex. Dacă utilizatorul apasă pe butonul **Close** , dorim sa închidem fereastra, adică să apelăm metoda **close()** al ferestrei.

- În general bibliotecile pentru interfețe grafice folosesc callback pentru această interacțiune.

- Callback
  - este un pointer la o funcție,
  - dacă într-o metodă dorim să notificăm apariția unui eveniment, putem folosi un pointer la funcție (callback, primit ca parametru).
  - În momentul în care apare evenimentul se apelează această funcție (call back)

- Dezavantaje callback în c++ :
  - dacă avem mai multe evenimente de notificat, ne trubie funcții separate callback sau sa folosim paramterii generici (void*) care nu se pot verifica la compilare
  - metoda care apelează metoda callback este cuplat tare de callback (trebuie sa știe signatura funcției, parametrii, etc. Are nevoie de referința la metoda callback).

Signal. Slot.

- Semnalul (**signal)** este emis la apariția unui eveniment (ex.: cliked())
- Componentele Qt (widget) emit semnale pentru a indica schimbări de stări generate de utilizator
- Un **slot** este o funcție care este apelat ca și răspuns la un semnal.
- Semnalul se poate conecta la un slot, astfel la emiterea semnalului slotul este automat executat

```
QPushButton *btn = new QPushButton("&Close");
QObject::connect(btn,SIGNAL(clicked()),&app,SLOT(quit()));
btn->show();
```

- **Slot** poate fi folosit pentru a reacționa la semnale, dar ele sunt defapt metode normale.
- Semnalele și sloturile sunt decuplate între elementele
  - Obiectele care emit semnale nu au cunoștințe despre sloturile care sunt conectate la semnal
  - slotul nu are cunoștință despre semnalele conectate la el
  - Decuplarea permite crearea de componente cu adevărat independente folosind Qt.
- În general componentele Qt's au un set predefinit de semnale. Se pot adăuga și semnale noi folosind moștenire (clasa derivată poate adăuga semnale noi)
- Compenentel Qt's au și sloturi predefinite
- În general programatorul extinde componentele (moștenește din clasele QtWidget) și adaugă sloturi noi care se conectează la semnale

Conectarea semnalelor cu sloturi

Folosind metoda `QObject::connect` şi macrourile SIGNAL şi SLOT putem conecta semnale şi sloturi

```
QWidget* createButtons(QApplication &a) {
	QWidget* btns = new QWidget;
	QHBoxLayout* btnsL = new QHBoxLayout;
	btns->setLayout(btnsL);
	QPushButton* store = new QPushButton("&Store");
	btnsL->addWidget(store);
	QPushButton* close = new QPushButton("&Close");
	btnsL->addWidget(close);
	//connect the clicked signal from close button to the quit slot
(method)
	QObject::connect(close,SIGNAL(clicked()),&a,SLOT(quit()));
	return btns;
}
```

În urma conectării – slotul este apelat în momentul în care se generează semnalul. Sloturile sunt funcţii normale, apelul este la fel ca la orice funcție C++. Singura diferența între un slot și o funcție este că slotul se poate conecta la semnale .

La un semnal putem conecta mai multe sloturi, în urma emiteri semnalului se vor apela sloturile în ordinea în care au fost conectate

Exista o corespondență între signatura semnalului și signatura slotului (parametrii trebuie să corespundă).

Slotul poate avea mai puţine parametrii, parametrii de la semnal care nu au corespondent la slot se vor ignora.

Conectarea semnalelor cu sloturi

```
QSpinBox *spAge = new QspinBox();
QSlider *slAge = new QSlider(Qt::Horizontal);
```

```
//Synchronise the spinner and the slider
//Connect spin box - valueChanged to slider setValue
QObject::connect(spAge,
SIGNAL(valueChanged(int)),slAge,SLOT(setValue(int)));
//Connect - slider valueChanged to spin box setValue
QObject::connect(slAge,
SIGNAL(valueChanged(int)),spAge,SLOT(setValue(int)));
```

Dacă utilizatorul schimbă valoarea în spAge (Spin Box):

- se emite semnalul valueChanged(int) argumentul primește valoarea curentă din spinner
- fiindcă cele două componente (spinner, slider) sunt conectate se apelează metoda setValue(int) de la slider.
- Argumentul de la metoda valueChanged (valoarea curent din spinner) se transmite ca și parametru pentru slotul, metoda setValue din slider
- sliderul se actualizează pentru a reflecta valoarea primită prin setValue și emite la rândul lui un semnal valueChanged (valoarea din slider s-a modificat)
- sliderul este conectat la spinner, astfel slotul setValue de la spinner este apelat ca și răspuns la semnalul valueChanged.
- De data asta setValue din spinner nu emite semnal fiindcă valoarea curentă nu se schimba (este egal cu ce s-a primit la setValue) astfel se evită ciclul infinit de semnale

Componente definite de utilizator

- Se crează clase seprate pentru interfața grafică utilizator
- componentel grafice create de utilizator extind componentele existente în Qt
- scopul este crearea de componente independente cu semnale și sloturi proprii

```
/**
 * GUI for storing Persons
 */
class StorePersonGUI: public QWidget {
public:
	StorePersonGUI();
private:
	QLabel *lblName;
	QLineEdit *txtName;
	QLabel *lblAdr;
	QLineEdit *txtAdr;
	QSpinBox *spAge;
	QLabel *lblAge2;
	QSlider *slAge;
	QLabel *lblAge3;
	QPushButton* store;
	QPushButton* close;
	/**
	 * Assemble the GUI
	 */
	void buildUI();
	/**
	 * Link signals and slots to define the behaviour of the GUI
	 */
	void connectSignalsSlots();
};
```

QMainWindow

- În funcție de componenta ce definim putem extinde clasa QWidget, QMainWindow, QDialog etc.
- Clasa QMainWindow poate fi folosit pentru a crea fereastra principală pentru o aplicație cu interfață grafică utilizator.
- QMainWindow are propriul layout, se poate adăuga toolbar, meniuri, status bar.
- QMainWindow definește elementele de bază ce apar în mod general la o aplicație

Layout QMainWindow:

- Meniu – pe partea de sus

```
QAction *openAction = new QAction("&Load", this);
QAction *saveAction = new QAction("&Save", this);
QAction *exitAction = new QAction("E&xit", this);
fileMenu = menuBar()->addMenu("&File");
fileMenu->addAction(openAction);
fileMenu->addAction(saveAction);
fileMenu->addSeparator();
fileMenu->addAction(exitAction);
```

- Toolbar

```
QToolBar* fileToolBar = addToolBar("&File");
fileToolBar->addAction(openAction);
fileToolBar->addAction(saveAction);
```

- Componenta din centru

```
    middle = new QTableWidget(10, 10, this);
    setCentralWidget(middle);
```

- Status bar - în partea de jos

```
statusBar()->showMessage("Status Message ....");
```

Qt Build system

O aplicație c++ conține fișiere header (.h) și fișiere (.cpp)

Procesul de build pentru o aplicație c++ :

- se compilează fișierele cpp folosind un compilator (fișierele sursă pot referi alte fișiere header) → fișiere obiect (.o)
- folosind un linker, se combină fișierele obiect (link edit) → fișier executabil(.exe)

Qt introduce pași adiționali:

Meta-object compiler (moc)

- compilatorul meta-object compiler ia toate clasele care încep cu macroul Q_OBJECT și generează fișiere sursă C++ moc_*.cpp. Aceste fișiere sursă conțin informații despre clasele compilate (nume, ierarhia de clase) și informații despre signale și sloturi. Practic în fișierele surse generate gasim codul efectiv care apelează metodele slot când un semnal este emis (generate de moc).

User interface compiler

– Compilatorul pentru interfețe grafice are ca intrare fișiere create de Qt Designer și genereaza cod C++ (ulterior putem apela metoda setupUi pentru a instanția componentele GUI).

Qt resource compiler

– Se pot include icoane, imagini, fișiere text în fișierul executabil. Fișierele astfel incluse in executabil se pot accesa din cod ca și orice fișier de pe disc.

Qt Build – din linia de commandă

Se execută :

- qmake -project
  - generează un fișier de proiect Qt (.pro)
- qmake
  - pe baza fișierului .pro se generează un fișier make
- make
  - execută fișierul make (generat de qmake). Apelează tot ce e necesar pentru a transforma fișierele surse în fișer executabil(meta-object compiler, user interface compiler, resource compiler, c++ compiler, linker)

Semnale și sloturi definite - Q_OBJECT

Putem defini semnale și sloturi în componentele pe care le creăm

```
class Notepad : public QMainWindow
{
    Q_OBJECT
...
```

Macroul Q_OBJECT trebuie sa apară la începutul definiției clasei. El este necesar în orice clasă unde vrem să adăugam semnale și sloturi noi.

Qt introduce un nou mecanism **meta-object system** care oferă :

- funcționalitate de semnale și sloturi (signals–slots)
- introspecție.

Instrospecția este un mecanism care permite opținerea de informații despre clase dinamic, programatic în timpul rulări aplicației. Este un mecanism folosit pentru semnale și sloturi transparent pentru programator.
Prin introspecție se pot accesa meta-informații despre orice QObject în timpul execuției – lista de semanle, lista de sloturi, numele metodelor numele clasei etc.
Orice clasă care începe cu Q_OBJECT este QObject .

Instrumentul moc (meta-object compiler, moc.exe) inspecteaza clasele ce au Q_OBJECT în definiție și expun meta-informații prin metode normale C++. Moc genereaza cod c++ ce permite introspecție (in fișiere separate *.moc)

Semnale și sloturi definite de utilizator

Pentru a crea sloturi se foloseste cuvântul rezervat *slots* (este defapt un macro). Qt (moc utility) are nevoie de această informație pentru a genera meta-informații despre sloturile disponibile ale componentei
Slotul în sine este doar o metodă obișnuită .

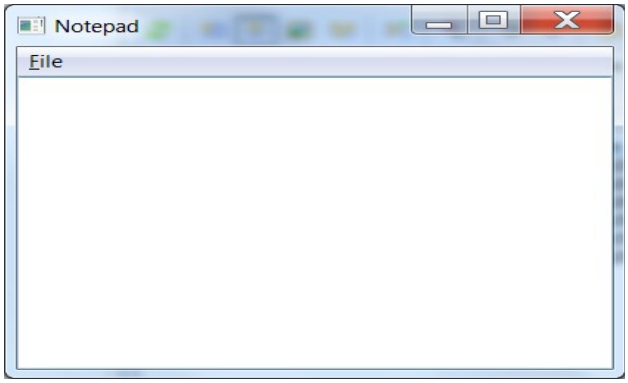
```
class Notepad : public QMainWindow
{
    Q_OBJECT

public:
    Notepad();

private slots:
    void open();
    void save();
    void quit();
```

```
void Notepad::save()
{
...
}
```

Notepad




```
class Notepad : public QMainWindow
{
    Q_OBJECT

public:
    Notepad();

private slots:
    void open();
    void save();
    void quit2();
```

```
    openAction = new QAction(tr("&Load"), this);
    saveAction = new QAction(tr("&Save"), this);
    exitAction = new QAction(tr("E&xit"), this);

    connect(openAction, SIGNAL(triggered()), this, SLOT(open()));
    connect(saveAction, SIGNAL(triggered()), this, SLOT(save()));
    connect(exitAction, SIGNAL(triggered()), this, SLOT(quit2()));
```

```
void Notepad::open()
{
    QString fileName = QFileDialog::getOpenFileName(this, tr("Open File"), "",
        tr("Text Files (*.txt);;C++ Files (*.cpp *.h)"));

    if (fileName != "") {
        QFile file(fileName);
        if (!file.open(QIODevice::ReadOnly)) {
            QMessageBox::critical(this, tr("Error"), tr("Could not open file"));
            return;
        }
        QTextStream in(&file);
        textEdit->setText(in.readAll());
        file.close();
    }
```

Semnale proprii

Folosind macroul *signals* se pot declara semnale proprii pentru componentele pe care le creăm.

```
private signals:
     storeRq(QString* name,QString* adr );
```

Cuvântul rezervat **emit** este folosit pentru a emite un semnal.

```
emit storeRq(txtName->text(),txtAdr->text());
```

Semnalele sloturile definite de programator au același status și comportament ca și cele oferite de componentele Qt