Curs 9-10 – Interfețe grafice utilizator

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

Curs 8

- **Polimorfism – exercitii**
- **Spații de nume (Namespace)**
- **Interfețe grafice utilizator - Qt**

Semnale și sloturi (Signals and slots)

- Semnalele 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.

- Când facem modificări la o componentă (scriem un text, apăsam butonul, selectăm un element,etc.) dorim ca alte părți ale aplicației să fie notificate (să actualizăm alte componente, 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 trebuie funcții separate callback sau sa folosim parametrii 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 (ex.: `&QPushButton::clicked`) la apariția unui eveniment
- 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

```
QApplication a(argc, argv);
QPushButton* btn = new QPushButton("&Close");
QObject::connect(btn, &QPushButton::clicked, &a, &QApplication::quit);
btn->show();
```

- **Slot** poate fi folosit pentru a reacționa la semnale, dar ele sunt de fapt metode normale sau funcții lambda.
- 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) sau se pot folosi funcții lambda

Conectarea semnalelor cu sloturi

Folosind metoda `QObject::connect` 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, QPushButton::clicked,
			&a, QApplication::quit);
	return btns;
}
```

În urma conectării – slotul este apelat în momentul în care se generează semnalul. Sloturile sunt funcții 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 puține parametrii, parametrii de la semnal care nu au corespondent la slot se vor ignora.

Conectarea semnalelor cu sloturi – referinta metode, lambda

Înainte de QT 5 semnale si sloturile se indicau folosind macrourile SIGNAL,SLOT

Odată cu QT 5 este permis folosirea de funcții lambda, referințe la funcții/metode

```
QObject::connect(close, &QPushButton::clicked,
                 &a,QApplication::quit);
```

```
QObject::connect(close, &QPushButton::clicked, [&a]() {
         a.quit();
    });
```

```
QObject::connect(close, SIGNAL(clicked()), &a, SLOT(quit()));
```

`&QPushButton::clicked  este o referinta la functia clicked, metoda din clasa  QPushButton`

Cele trei variante de mai sus sunt echivalente (realizează același lucru) dar este de preferat sa folosim variantele tipizate atât pentru semnale (`&QPushButton::clicked`) cat si pentru sloturi (`QApplication::quit).`

Folosind funcții lambda si referințe la funcții compilatorul poate verifica daca semnalul si slotul este compatibil

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)));
```

```
//prutem prelua valori de la semnal
QObject::connect(spAge, SIGNAL(valueChanged(int)), slAge,
SLOT(setValue(int)));
//Connect - slider valueChanged to spin box setValue
QObject::connect(slAge, &QSlider::valueChanged, [spAge](int val) {
           spAge->setValue(val);}
);
```

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 creează clase separate pentru interfața grafică utilizator
- componentele 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);
```

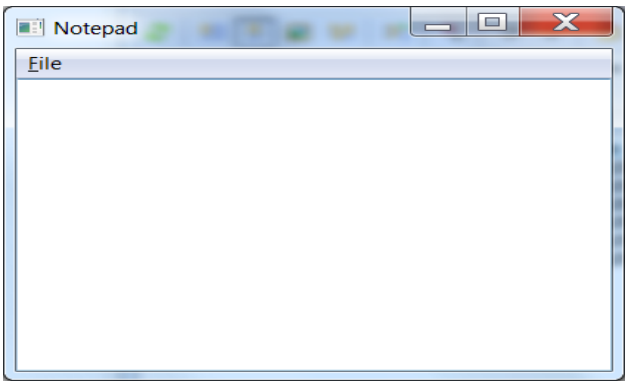
- Componenta din centru

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

- Status bar - în partea de jos

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

Notepad




```
class Notepad : public QMainWindow
{

public:
    Notepad();

private:
    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, &QAction::triggered, this, &Notepad::open);
    connect(saveAction, &QAction::triggered, this, &Notepad::save);
    connect(exitAction, &QAction::triggered, this, &Notepad::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();
    }
```

Exemplu PetStore:

```
class PetStoreGUI : public QWidget{
private:
	PetController& ctr;
	QListWidget* lst;
	QPushButton* btnSortByPrice;
	QPushButton* btnSortByType;
	QLineEdit* txtSpecies;
	QLineEdit* txtType;
	QLineEdit* txtPrice;
	void initGUICmps();
	void connectSignalsSlots();
	void reloadList(std::vector<Pet> pets);
public:
	PetStoreGUI(PetController& ctr) :ctr{ ctr } {
		initGUICmps();
		connectSignalsSlots();
		reloadList(ctr.getAllPets());
	}
};
void PetStoreGUI::reloadList(std::vector<Pet> pets) {
	lst->clear();
	for (auto& p : pets) {
		QListWidgetItem* item = new QListWidgetItem(p.getSpecies(), lst);
		item->setData(Qt::UserRole, p.getType());//adaug in lista (invizibil) si type
		//lst->addItem(p.getSpecies());
	}
}
void PetStoreGUI::connectSignalsSlots() {
	//cand se emite semnalul clicked de pe buton reincarc lista
	QObject::connect(btnSortByPrice, &QPushButton::clicked, [&]() {
		reloadList(ctr.getSortByPrice());
	});
	//cand se emite semnalul clicked de pe buton reincarc lista
	QObject::connect(btnSortByType, &QPushButton::clicked, [&]() {
		reloadList(ctr.getSortByType());
	});
	//cand se selecteaza elementul din lista incarc detaliile
	QObject::connect(lst, &QListWidget::itemSelectionChanged, [&]() {
		if (lst->selectedItems().isEmpty()) {
			//nu este nimic selectat (golesc detaliile)
			txtSpecies->setText("");
			txtType->setText("");
			txtPrice->setText("");
			return;
		}
		QListWidgetItem* selItem =  lst->selectedItems().at(0);
		txtSpecies->setText(selItem->text());
		txtType->setText(selItem->data(Qt::UserRole).toString());
	});
}
```

Clase QT - utile

QString – sir de caractere Unicode

Metodele care primesc un parametru QString accepta si char* (QString are un constructor cu char*)

```
txtSpecies->setText("ceva text");
```

Exista metode de a transforma din QString in std::string si invers

```
QString s = "145";
std::string ss = s.toStdString();
QString s2 = QString::fromStdString(ss);
```

Exista posibilitatea de a transforma numere in QString si invers

```
QString s = "145";
int a = s.toInt();
double d = 3.8;
QString s3 = QString::number(d);
```

QList/QVector lista/vector variante Qt pentru containere din STL

au operatii similare ca cele din stl: [0], at(0), pop(), isEmpty(), size(), etc

Se pot transforma in variantele stl

```
QList<int> l;
l.push_back(3);
l.push_front(4);
l[1] = 4; l.at(0);
```

```
std::list<int> ll = l.toStdList();
QList<int> l2 = QList<int>::fromStdList(ll);
```

QMessageBox

Putem afisa o fereastra cu informatii

```
QMessageBox::information(this, "Info", "Selection changed");//afiseaza mesaj

int ret = QMessageBox::warning(this, "My Application",
             "The document has been modified.\nDo you want to save your changes?",
             QMessageBox::Save | QMessageBox::Discard| QMessageBox::Cancel,
             QMessageBox::Save);
if (ret == QMessageBox::Save) {
    //do save
}
```

QDebug – stream pentru a scrie informații ce ajuta la depanarea programelor

```
qDebug() << "Date:" << QDate::currentDate();
```

Item based Widgets – QlistWidget,QTableWidget

QListWidget / QListWidgetItem	QTableWidget / QTableWidgetItem

`QListWidget* lst = new QListWidget;`
 	
`//se pot adauga elemente`
`QListWidgetItem* item = new`
`QListWidgetItem("Bla", lst);`
	`//se creaza`
	`int nrLinii = 4;`
	`int nrColoane = 3;`
	`QTableWidget* tbl = new`
	`QTablWidget{ nrLinii,nrColoane };`
	
	`//se pot adauga elemente`
	`QTableWidgetItem* cellItem1 = new`
	`QTableWidgetItem("Linie1");`
	`tbl->setItem(0, 0, cellItem1);`
	`tbl->setItem(0, 1, new`
	`QTableWidgetItem("Linie1`
	`coloana2"));`

`//se poate configura modul de selectie`
`lst->setSelectionMode(`
`QAbstractItemView::SingleSelection);`
`//se poate obtine selectia`
`auto selItms = lst->selectedItems();`
	`//se poate configura modul de selectie`
	`tbl->setSelectionBehavior(`
	`QAbstractItemView::SelectRows);`
	`tbl->setSelectionMode(`
	`QAbstractItemView::SingleSelection);`
	`//se poate obtine selectia`
	`auto selTblItms = tbl->selectedItems();`

`//putem reactiona la semnale`
`QObject::connect(lst,`
`&QListWidget::itemSelectionChanged,`
`[lst]() {`
`qDebug() << "Selectie \n" <<`
`lst->selectedItems() << "\n";`
`});`
	`//putem reactiona la semnale`
	`QObject::connect(tbl,`
	`&QTableWidget::itemSelectionChanged,`
	`[tbl]() {`
	`qDebug() << "Selectie tabel\n"<<`
	`tbl->selectedItems()<<"\n";`
	`});`

Fiecare celula din tabel / lista conține textul dar si alte informații:

```
//informatii suplimentare in item
item->setBackground(QBrush{ Qt::red, Qt::SolidPattern });
item->setTextColor(Qt::blue);
item->setData(Qt::UserRole, QString{ "informatii care nu se vad" });
item->setCheckState(Qt::Checked);
item->setIcon(QApplication::style()->standardIcon(QStyle::SP_BrowserReload));
```

```
//informatii suplimentare pentru fiecare celula
cellItem1->setBackground(QBrush{ Qt::red, Qt::SolidPattern });
cellItem1->setTextColor(Qt::blue);
cellItem1->setData(Qt::UserRole, QString{ "informatii care nu se vad" });
cellItem1->setCheckState(Qt::Unchecked);
cellItem1->setIcon(QApplication::style()->standardIcon(QStyle::SP_ArrowBack));
```

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 găsim 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 generează 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

Util daca folosiți altceva decât Visual Studio. In cazul folosiri Visual Studio aceste aspecte sunt rezolvate automat folosind extensia Qt.

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 fișier 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.

Introspecția este un mecanism care permite obținerea 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) inspectează clasele ce au Q_OBJECT în definiție și expun meta-informații prin metode normale C++. Moc generează cod c++ ce permite introspecție (in fișiere separate *.moc)

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

Semnale proprii exemplu

```
class BrickGameEngine: public QObject{
	Q_OBJECT  //e nevoie de acest macro daca vrem semnale custom
	int score = 0;
	int dead = 0;
	int nrBricks = 0;
	QTimer timer;
	int ballMoveDelay = 20;
	int elapsedMoves = 0;
signals:
	//semnale generate de engine
	void scoreChanged(int currentScore);
	void deadChanged(int currentNrDead);
	void advanceBoard();
	void brickCreated(int x, int y, int brickW, int brickH);
	void gameFinished(bool win);

public:
	BrickGameEngine() {
		emit scoreChanged(score);
		emit deadChanged(dead);
	}

	void brickHit() {
		score += 1;
		nrBricks--;
		emit scoreChanged(score);
	}
QObject::connect(&engine, &BrickGameEngine::gameFinished, [&](bool win) {
			if (win) {
				QMessageBox::information(this, "Info","You win!!!");
			}
			else {
				QMessageBox::information(this, "Info", "You lose!!!");
			}
		});
```

Function Callback

Mecanismul de semnal/slot este proprietar Qt, este o implementare speciala (folosește moc compiler pentru a genera codul C++ automat) nu o sa le găsiți in alte biblioteci. In general același idee se poate implementa folosind ideea de callback, idee des implementata in proiecte reale care nu folosesc Qt.

Ce este:

o funcție callback este o funcție apelata folosind o referenta la funcție (pointer la functie)

Ce problema rezolva:

Dorim sa decuplam obiectul care apelează metoda de obiectul care are metoda de apelat. Cel care apelează metoda nu trebuie sa știe nimic despre obiectul apelat, numitorul comun intre cele doua obiecte este declarația funcției.

Când este util:

Ex1: creez o biblioteca de sortare, doresc ca sortarea sa fie independent de metoda de comparare

Ex2: este util pentru notificări.

Daca doresc sa creez un obiect timer (măsoară timpul si emite notificări la intervale de timp). In momentul in care dezvolt clasa Timer nu știu cine o sa folosească aceasta clasa si in ce scop. Ar trebui sa creez o soluție generala astfel încât oricine are nevoie de funcționalitatea de timer sa poate folosi cu codul lui.

Implementatorul clasei Timer o sa definească prototipul funcției callback, si cel care dorește notificări trebuie sa implementeze o funții cu aceasta signatura

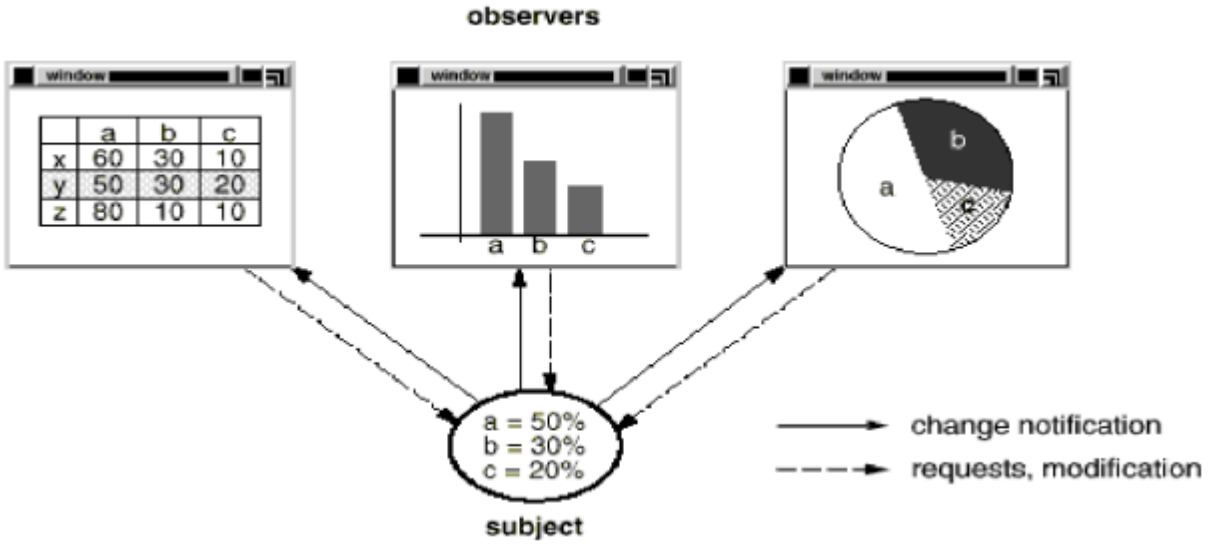
```
//callback function prototipe
typedef void(*ProgressListener)(int percent);

//function that notify progress using callback
void someComputation(ProgressListener callback)
{
	for (int i = 0; i < 100; i++) {
		//do stuff
		callback(i);
	}
}
```

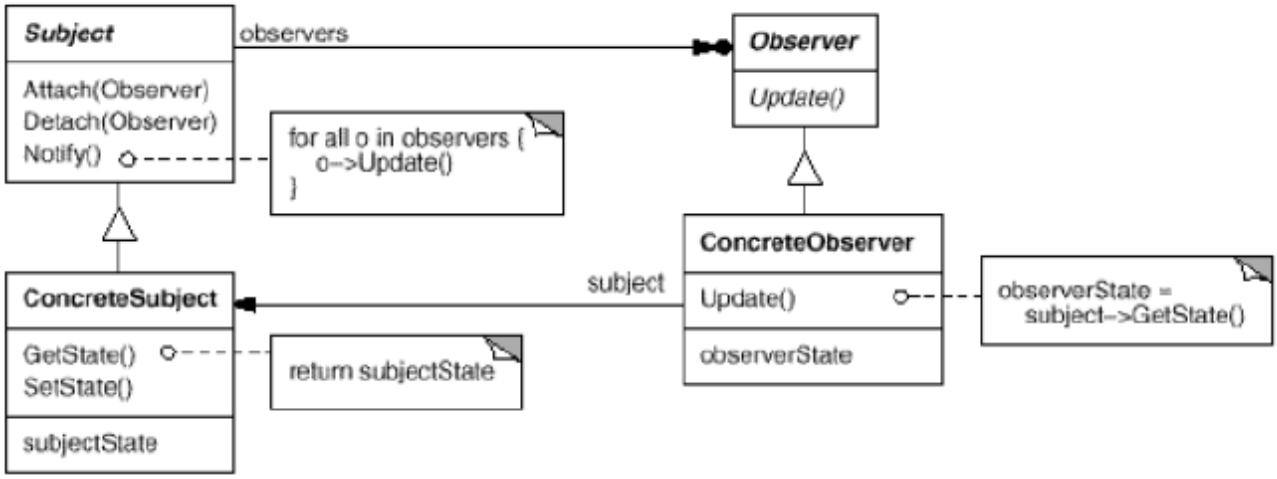
```
void onProgress(int percent) {
	std::cout << "progress:" << percent;
}
int main(int argc, char *argv[])
{
	someComputation(onProgress);
}
```

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++

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

class InterestedObj: public Observer {
public:
	void update() {
		std::cout << "Notified" << std::endl;
	}
};


void notify(Observer* obs) {
	obs->update();
}
class Observable {
public:
	void addObserver(Observer *obs) {
		observers.push_back(obs);
	}
	void doStuff() {
		//some stuff
		notifyObservers();
	}
private:
	std::vector<Observer*> observers;
	void notifyObservers() {
		for_each(observers.begin(), observers.end(), notify);
	}
};


int main() {
	Observable someObject;
	Observer* obs1 = new InterestedObj();
	Observer* obs2 = new InterestedObj();
	someObject.addObserver(obs1);
	someObject.addObserver(obs2);
	someObject.doStuff();
	return 0;
}
```

Gestiunea evenimentelor de mouse/tastatura/desenare

Evenimentele de mouse si de tastatura sunt transmise componentelor GUI Qt (orice clasa care extinde QWidget/QObject)

Când apare evenimentul Qt creează un obiect **QEvent** cu detalii despre eveniment (clase derivate: **QResizeEvent**, **QPaintEvent**, **QMouseEvent**, **QKeyEvent**, and **QcloseEvent**)

Gestiunea evenimentelor (**Event handlers**)
Evenimentele ajung la componentele Qt prin metode virtuale (sistemul Qt apelează metode virtuale) definite in clasa QWidget.
Intr-o componenta definita de utilizator putem suprascrie metodele pentru a trata evenimentul:

```
class SampleW :public QWidget {
public:
	SampleW() {
	  //If mouse tracking is disabled(default)the widget only receives mouse move events
	  //when at least one mouse button is pressed while the mouse is being moved
		setMouseTracking(true);
	}
protected:
	void paintEvent(QPaintEvent* ev) override {
		qDebug() << "paint requested";
		...
	}
	void mouseMoveEvent(QMouseEvent* ev) override {
		qDebug() << ev->pos();
	}
};
```

Trimitere de evenimente

Este posibil generarea de evenimente proprii folosind metoda QCoreApplication::sendEvent()

Se creează obiectul QEvent dorit si prin funcția sendEvent se poate trimite către componentele Qt.

QCoreApplication Gestionează o coada de evenimente care sunt gestionate de un singur fir de execuție (Event loop).

Desenare low-level

Clasa **QPainter** permite desenarea “manuala”, are funcții optimizate pentru a desena orice elemente avem nevoie intr-o aplicație grafica (linii, dreptunghi, elipse, imagini, etc)

In general obiect **QPainter** se folosește in interiorul metodei pentEvent, metoda ce este apelata de fiecare data când s-a cerut redesenarea widgetului(repaint() sau update() au fost apelate ori de Qt automat ori de programatic din cod).

```
void paintEvent(QPaintEvent* ev) override {
            QPainter p{ this };

            p.drawLine(0, 0, width(), height());
            p.drawImage(x,0,QImage("sky.jpg"));

            p.setPen(Qt::blue);
            p.setFont(QFont("Arial", 30));
            p.drawText(rect(), Qt::AlignTop | Qt::AlignHCenter, "Qt QPainter");

            p.fillRect(0, 100, 100, 100,Qt::BrushStyle::Dense1Pattern);

      }
```

Metoda paintEvent se poate suprascrie in orice clasa care mosteneste din QWidget

Graphics View Framework

Graphics View oferă suport pentru aplicații care gestionează si interacționează cu un număr mare de obiecte 2D (**2D graphical items**)

Ofera suport pentru:

- zoom/rotatii/transformari
- animatii
- tiparire
- drag and drop
- OpenGL

Arhitectura framworkului Graphic View

QGraphicScene – conține elementele grafice **QGraphicItem**, se ocupa cu propagarea evenimentelor către obiectele grafice 2D

QGraphicView – vizualizează conținutul unei scene, este defapt un "scroll area" oferă posibilitatea de a naviga prin scena. Se pot atașa multiple vederi la același scena

QGraphicItem – si clasele derivate din el (**QGraphicsRectItem, QGraphicsElipseItem, QGraphicsTextItem, QGraphicsRectItem)** sunt elementele grafice ce se pot adaugă într-o scena.

Oferă suport pentru: evenimente (mouse/tastatura/context menu), grupare de elemente grafice, drag and drop, detectare de coliziuni

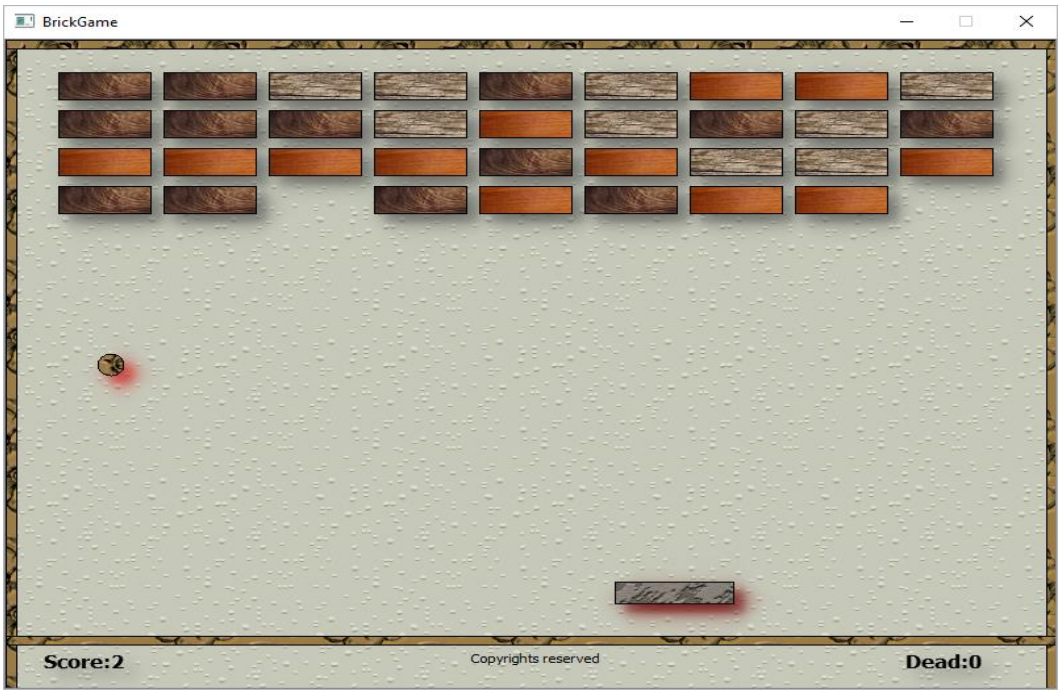
```
QGraphicsView* view = new QGraphicsView;
QGraphicsScene* scene = new QGraphicsScene;
view->setScene(scene);
view->setFixedSize(800, 600);
scene->setSceneRect(0, 0, 800, 600);

//add items to scene
QGraphicsEllipseItem* ball = new QGraphicsEllipseItem{ 0,0,20,20 };
ball->setPos(scene->width() / 2, scene->height()/2 );
scene->addItem(ball);

QGraphicsRectItem* r = new QGraphicsRectItem{ 0,0,40,30 };
r->setPos(20, scene->height() / 2);
r->setBrush(QBrush(QImage("wood1.jpg")));
scene->addItem(r);

view->show();
```

Graphics View Framework




```
class BrickGame:public QGraphicsView {
       QGraphicsScene*  scene;
       Paddle* player;
       Ball* ball;
public:
       BrickGame() {
              setMouseTracking(true);
              initScene();
              createPlayer();
              loadLevel();
              addBall();
              startGame();
       }
void startGame() {
    QTimer* timer = new Qtimer;
    //advanceGame invoked every time
    QObject::connect(timer, &Qtimer::timeout,
                this,&BrickGame::advanceGame);
     //generate timeot signal every 20ms
     timer->start(20);
   }

//handle mouse move
void mouseMoveEvent(QMouseEvent* ev) override{
       //works only if setMouseTracking(true);
       auto x = ev->pos().x();
       player->setPos(x, player->y());

}
}
```

```
class Ball :public QGraphicsEllipseItem {
private:
       QGraphicsDropShadowEffect * effect;
       int dx = 5;
       int dy = -5;
public:
   Ball() {
     setRect(0, 0, 20, 20);
     setBrush(QBrush(QImage("ball.jpg")));

     effect = new QGraphicsDropShadowEffect();
     effect->setBlurRadius(30);
     setGraphicsEffect(effect);
}

  void move() {
     setPos(x() + dx, y() + dy);
    setTransformOriginPoint(rect().width() / 2,
                      rect().height() / 2);
     setRotation(rotation() + 45);
    }

    void changeXDir() {
              dx *= -1;
    }
}
```