

Ghid de Arhitectură și Implementare Qt pentru Aplicații C++

C++ Modern Aplicat în Inteligența Artificială

Asist. Mrd. Andrei Rozmarin

Asist. Mrd. Cezar Constandoiu

Anul II – Semestrul I | 2025 – 2026

Facultatea de Matematică și Informatică
Universitatea *Transilvania* din Brașov

Cuprins

I	Configurarea Mediului de Dezvoltare	3
1	Instalare și Configurare Qt 6	4
1.1	Instalarea Qt SDK și Qt Creator	4
1.2	Configurarea Kit-urilor (MinGW vs MSVC)	4
1.3	Structura unui proiect Qt: fișierul .pro vs CMakeLists.txt	4
II	Concepte Fundamentale și Arhitectură	5
2	Modelul de Obiecte Qt (Qt Object Model)	6
2.1	Clasa QObject: Fundația Framework-ului	6
2.2	Sistemul Meta-Object (MOC)	7
2.2.1	Cum funcționează?	7
2.3	Managementul Memoriei: Ierarhia Părinte-Copil	7
2.3.1	Regula de aur	8
2.3.2	Când folosim smart pointers în Qt?	8
3	Comunicarea între Componente: Semnale și Sloturi	9
3.1	Conceptul Observer Pattern în Qt	9
3.2	Semnale (Signals)	9
3.3	Sloturi (Slots)	10
3.4	Conectarea Obiectelor (Connect)	10
3.4.1	Exemplu aplicat: Resetarea jocului	10
3.5	Utilizarea Lambda Expressions pentru parametrizare	11
3.6	Decuplarea Logicii de Interfață	11
4	Elemente de Interfață Grafică (Qt Widgets)	12
4.1	Clasa QWidget: Atomul Interfeței	12
4.2	Manageri de Layout (Layout Managers)	12
4.2.1	Tipuri de Layout-uri	12
4.3	Componente Esențiale pentru Joc	13
5	Sistemul de Resurse	14
5.1	Fișierele .qrc și Compilatorul de Resurse (RCC)	14
5.2	Organizarea și Prefixele	14
5.3	Accesarea Resurselor din Cod	15
5.4	Exemplu Aplicat: Îmbunătățirea aspectului X și O	15

III	Studiu de Caz Complet: X și O	16
6	Implementarea Jocului X și O	17
6.1	Structura Proiectului	17
6.2	Implementarea Business Logic	17
6.3	Integrarea Logică - UI	17
IV	Concepte Avansate pentru Proiecte Complex	18
7	Graphics View Framework	19
7.1	Arhitectura Model-View (Scenă-Vizualizare)	19
7.2	Configurarea Scenei și a View-ului	19
7.3	Crearea Elementelor Personalizate (QGraphicsItem)	20
7.4	Interacțiunea: Drag and Drop simplificat	21
7.4.1	Detectarea Click-ului (fără mutare)	21

Partea I

Configurarea Mediului de Dezvoltare

Capitolul 1

Instalare și Configurare Qt 6

- 1.1 Instalarea Qt SDK și Qt Creator
- 1.2 Configurarea Kit-urilor (MinGW vs MSVC)
- 1.3 Structura unui proiect Qt: fișierul .pro vs CMakeLists.txt

Partea II

Concepte Fundamentale și Arhitectură

Capitolul 2

Modelul de Obiecte Qt (Qt Object Model)

C++ este un limbaj puternic și eficient, însă standardul său original nu a fost conceput cu gândul la interfețe grafice dinamice, introspecție¹ sau comunicare asincronă între componente. Pentru a acoperi aceste lipsuri fără a sacrifica performanța, Qt a extins C++ printr-un sistem propriu de obiecte.

În acest capitol vom explora fundația pe care este construit întregul framework și vom vedea cum implementarea jocului nostru de X și O beneficiază de aceste extensii.

2.1 Clasa QObject: Fundația Framework-ului

La baza bibliotecii Qt se află clasa `QObject`. Aproape orice clasă cu care veți interacționa în Qt (de la `QPushButton` și `QTimer` până la `QThread`) moștenește direct sau indirect din `QObject`.

Definiție

QObject este clasa de bază a modelului de obiecte Qt. Ea oferă funcționalități precum:

- Comunicarea între obiecte prin *Semnale* și *Sloturi*.
- Gestionarea memoriei prin ierarhia părinte-copil.
- Proprietăți dinamice și introspecție (Reflection).

O distincție fundamentală în Qt este diferența dintre **Tipuri Valoare** (Value Types) și **Tipuri Identitate** (Identity Types):

- **Tipuri Valoare (ex: `QString`, `QList`):** Sunt copiate la atribuire. Două string-uri cu conținutul „X” sunt considerate egale și interschimbabile.
- **Tipuri Identitate (ex: `QObject`, `QWidget`):** Reprezintă entități unice. Un buton “Start Joc” este o entitate unică în memorie; nu are sens să îl copiem. Dacă am copia un buton, ce s-ar întâmpla cu semnalele conectate la el? Ar trebui să se ducă la două conexiuni?

De reținut

Clasele derivate din `QObject` **nu pot fi copiate**. Operatorul de atribuire și constructorul de copiere sunt dezactivați prin macro-ul `Q_DISABLE_COPY`. De aceea, obiectele Qt sunt manipulate aproape exclusiv prin pointeri.

¹**Introspecția** (sau *Reflection* în alte limbaje) este capacitatea unui program de a examina tipul și structura obiectelor (numele clasei, metodele disponibile, proprietățile) în timpul execuției (*runtime*). Deși C++ standard oferă un suport limitat prin RTTI (*Runtime Type Information*: `typeid`, `dynamic_cast`), Qt extinde acest concept permițând interogarea detaliată a metadatelor oricărui obiect derivat din `QObject`.

2.2 Sistemul Meta-Object (MOC)

C++ standard este un limbaj compilat static, ceea ce înseamnă că structura claselor este „înghețată” după compilare. Nu putem întreba un obiect la runtime „ce metode ai?” sau „ce nume are clasa ta?”. Totuși, pentru a realiza conexiuni dinamice între interfață (GUI) și logică, avem nevoie de aceste informații.

Soluția Qt este **Meta-Object Compiler (MOC)**. Acesta nu este o parte a compilatorului C++ (gcc/clang/msvc), ci un utilitar separat care rulează *înainte* de compilare.

2.2.1 Cum funcționează?

1. MOC scanează fișierele header (.h) din proiect.
2. Când întâlnește o clasă care conține macro-ul `Q_OBJECT`, extrage informațiile despre semnale, sloturi și proprietăți.
3. Generează un fișier C++ suplimentar (de obicei numit `moc_filename.cpp`) care conține implementarea meta-datelor.
4. Compilatorul C++ compilează atât sursa originală, cât și fișierul generat de MOC.

Observație

Dacă uitați să includeți macro-ul `Q_OBJECT` în definiția clasei, veți putea compila codul, dar semnalele și sloturile nu vor funcționa, iar funcțiile precum `tr()` (pentru traduceri) nu vor fi disponibile.

```
1 // game_logic.h
2 #ifndef GAMELOGIC_H
3 #define GAMELOGIC_H
4
5 #include <QObject>
6
7 class GameLogic : public QObject
8 {
9     Q_OBJECT // Macro OBLIGATORIU pentru semnale/sloturi
10
11     public:
12     explicit GameLogic(QObject *parent = nullptr);
13     // ... metodele jocului ...
14
15     signals:
16     // Semnal emis când jocul s-a terminat
17     void gameOver(QString winner);
18 };
19
20 #endif // GAMELOGIC_H
```

Secvența de cod 2.1: Exemplu din clasa GameLogic (X și O)

2.3 Managementul Memoriei: Ierarhia Părinte-Copil

Una dintre cele mai frecvente surse de erori în C++ este gestionarea memoriei (memory leaks, double free). Deși C++ modern (C++11/14/17/20/23) rezolvă multe probleme prin pointeri inteligenți (de tip `std::unique_ptr`, `std::shared_ptr`), Qt folosește un sistem propriu, bazat pe o ierarhie de proprietate (*ownership tree*).

2.3.1 Regula de aur

Orice `QObject` poate avea un părinte (`parent`) și oricâți copii. Când un obiect părinte este distrus (dealocat), el își distruge automat toți copiii.

Acest mecanism simplifică drastic codul de GUI. Gândiți-vă la fereastra principală a jocului X și O:

1. Avem o fereastră principală (`MainWindow`).
2. Aceasta conține un widget central (`QWidget`).
3. Widget-ul central conține un layout (`QGridLayout`).
4. Layout-ul gestionează 9 butoane (`QPushButton`).

Dacă ar trebui să ștergem manual fiecare buton în destructorul ferestrei, codul ar fi stufos și predispus la erori. În Qt, este suficient să ștergem fereastra principală (`MainWindow`).

Exemplu practic: Crearea tablei de joc

Când creăm dinamic butoanele pentru grila de X și O, le pasăm pointerul către părinte în constructor (sau prin `setParent`).

```
1 // Varianta C++ Standard (necesită delete manual sau vector de smart pointers)
2 QPushButton* btn = new QPushButton("X");
3 // Dacă uităm 'delete btn', avem memory leak.
4
5 // Varianta Qt (recomandată)
6 // 'this' este fereastra sau containerul părinte
7 QPushButton* btn = new QPushButton("X", this);
8
9 // Chiar dacă nu apelăm 'delete btn', acesta va fi șters automat
10 // când obiectul 'this' este distrus.
```

Secvența de cod 2.2: Alocare dinamică în Qt vs. C++ Standard

2.3.2 Când folosim smart pointers în Qt?

Deși sistemul părinte-copil se ocupă de obiectele GUI și de componentele majore ale framework-ului, pentru datele interne (Data Models) care nu moștenesc din `QObject`, se recomandă utilizarea pointerilor inteligenți din C++ modern.

În arhitectura jocului X și O, starea internă a tablei (ex: o clasă `BoardModel` sau o matrice de date) nu trebuie să fie neapărat compusă din widget-uri. Aceste structuri de date „pure” C++ ar trebui gestionate folosind `std::unique_ptr` sau `std::shared_ptr`, în timp ce reprezentarea lor grafică (butoanele `QPushButton` din interfață) vor fi gestionate automat de sistemul părinte-copil al Qt.

Capitolul 3

Comunicarea între Componente: Semnale și Sloturi

În programarea interfețelor grafice, o problemă fundamentală este comunicarea între elemente: cum știe logica jocului că utilizatorul a apăsă un buton? Și invers, cum știe fereastra că jocul s-a terminat și trebuie afișat un mesaj?

Abordările clasice (precum funcțiile *callback* în C) sunt adesea rigide și nesigure (*not type-safe*). Qt introduce un mecanism puternic și flexibil numit **Semnale și Sloturi** (*Signals & Slots*).

3.1 Conceptul Observer Pattern în Qt

La nivel arhitectural, acest mecanism este o implementare a șablonului de proiectare *Observer*.

- **Subiectul (Sender):** Obiectul care își schimbă starea și „strigă” (emite un semnal). Nu știe cine îl ascultă.
- **Observatorul (Receiver):** Obiectul care așteaptă un eveniment și execută o funcție (slot) când acesta are loc.

De reținut

Avantajul major este **decuplarea slabă** (loose coupling). Clasa care gestionează logica X și O nu trebuie să includă header-ul ferestrei grafice. Ea doar emite semnale, iar cine este interesat (interfața) le ascultă.

3.2 Semnale (Signals)

Un semnal este o funcție declarată în secțiunea `signals:` a clasei. Nu se implementează niciodată în fișierul `.cpp`; implementarea este generată automat de MOC.

În jocul nostru, clasa de logică (`GameLogic`) ar putea emite semnale când starea tablei se schimbă.

```
1 class GameLogic : public QObject
2 {
3     Q_OBJECT
4
5 public:
6     // ...
7
8 signals:
9     // Emis când un jucător a mutat valid
10    // Interfața va folosi asta pentru a desena X sau O pe buton
11    void boardUpdated(int row, int col, char playerSymbol);
```

```

12
13 // Emis când jocul s-a terminat
14 // Interfața va afișa un MessageBox
15 void gameOver(QString winnerName);
16 };

```

Secvența de cod 3.1: Definirea semnalelor în GameLogic.h

Pentru a declanșa evenimentul, folosim cuvântul cheie `emit`:

```

1 // În GameLogic.cpp
2 void GameLogic::checkWinCondition() {
3     if (/*...cineva a castigat...*/) {
4         emit gameOver("Jucatorul X");
5     }
6 }

```

3.3 Sloturi (Slots)

Un slot este o funcție membră normală a clasei. Singura diferență este că poate fi conectată la un semnal. Începând cu Qt 5, orice funcție membră (sau chiar funcții lambda) poate fi folosită ca slot, nefiind obligatorie secțiunea `public slots:`, deși este recomandată pentru claritate.

În interfața grafică (MainWindow), vom avea sloturi care reacționează la acțiunile utilizatorului sau la semnalele din logică.

```

1 class MainWindow : public QMainWindow
2 {
3     Q_OBJECT
4
5     public slots:
6         // Funcție apelată când logica ne spune că jocul e gata
7         void onGameOver(QString winner);
8
9         // Funcție apelată când apăsăm "Joc Nou"
10        void resetBoard();
11 };

```

Secvența de cod 3.2: Sloturi în MainWindow.h

3.4 Conectarea Obiectelor (Connect)

Legătura dintre un semnal și un slot se face la runtime folosind funcția statică `QObject::connect`.

Există două sintaxe. Vă recomandăm insistent să folosiți **sintaxa bazată pe pointeri la funcții** (disponibilă din Qt 5), deoarece verifică existența metodelor și compatibilitatea parametrilor la momentul compilării.

Semnătura funcției connect

```
connect(sender, &Sender::signal, receiver, &Receiver::slot);
```

3.4.1 Exemplu aplicat: Resetarea jocului

Să presupunem că avem un buton `resetBtn` în interfață și o instanță a logicii jocului `gameLogic`.

```

1 // În constructorul MainWindow
2 // Când butonul este apăsat (signal), logica resetează matricea (slot)
3 QObject::connect(ui->resetBtn, &QPushButton::clicked, m_gameLogic, &GameLogic::
    resetGame);

```

Secvența de cod 3.3: Conectarea unui buton la logică

3.5 Utilizarea Lambda Expressions pentru parametrizare

Aici apare o problemă specifică jocului X și O. Avem 9 butoane într-un QGridLayout. Semnalul standard al unui buton este `clicked(bool checked)`. Acest semnal nu ne spune care buton a fost apăsat (coordonatele lui).

Logica jocului așteaptă ceva de genul: `makeMove(int row, int col)`. Cum conectăm `clicked()` (fără parametri utili) la `makeMove(int, int)`?

Soluția modernă este utilizarea funcțiilor Lambda din C++11:

```

1 // Presupunem că buttons[3][3] este matricea de butoane din GUI
2
3 for(int i = 0; i < 3; ++i) {
4     for(int j = 0; j < 3; ++j) {
5
6         // Conectăm fiecare buton individual
7         // [=] capturează i și j prin valoare pentru a fi folosite în lambda
8         connect(buttons[i][j], &QPushButton::clicked, this, [=]() {
9
10            // Când butonul (i,j) este apăsat, apelăm logica
11            // cu coordonatele corecte
12            m_gameLogic->makeMove(i, j);
13
14        });
15    }
16 }

```

Secvența de cod 3.4: Maparea butoanelor la coordonate folosind Lambda

3.6 Decuplarea Logicii de Interfață

Acesta este cel mai important principiu arhitectural pentru proiectele voastre. **Greșeala comună:** Logica jocului modifică direct interfața.

```

1 // NU faceți așa în GameLogic.cpp!
2 void GameLogic::processMove() {
3     // Eroare de design: Logica depinde de MainWindow
4     mainWindow->labelStatus->setText("Move X");
5 }

```

Abordarea corectă (Qt way):

1. GameLogic își face calculele interne.
2. Când starea se schimbă, GameLogic **emite un semnal**: `emit statusChanged("Mută X")`.
3. MainWindow a conectat acel semnal la un slot propriu care actualizează eticheta.

Astfel, puteți lua clasa GameLogic și o puteți muta într-o aplicație de consolă sau într-o aplicație mobilă fără a schimba o singură linie de cod în ea. Acesta este mecanismul prin care decuplăm logica jocului de interfața grafică.

Capitolul 4

Elemente de Interfață Grafică (Qt Widgets)

Modulul **Qt Widgets** oferă o colecție de elemente UI standard (butoane, meniuri, bare de progres) care arată nativ pe orice sistem de operare (Windows, macOS, Linux). Pentru jocul nostru de X și O, nu vom desena manual liniile și cercurile pe ecran, ci vom compune tabla de joc din aceste elemente prefabricate.

4.1 Clasa QWidget: Atomul Interfeței

Clasa **QWidget** este clasa de bază pentru toate obiectele interfeței utilizator.

- Dacă un widget nu are părinte (`parent = nullptr`), el devine o **Fereastră** (Window).
- Dacă un widget are părinte, el devine un element vizual afișat în interiorul părintelui său.

De reținut

În Qt, „totul este un widget”. Butonul este un widget. Fereastra principală este un widget. Chiar și containerul care grupează alte widget-uri este, la rândul lui, un widget.

4.2 Manageri de Layout (Layout Managers)

Observație

O greșeală comună a începătorilor este poziționarea absolută (ex: `button->setGeometry(10, 10, 100, 50)`). Această abordare este problematică:

1. Dacă utilizatorul redimensionează fereastra, butoanele rămân pe loc, lăsând spațiu gol.
2. Pe ecrane cu DPI diferit (ex: 4K vs Full HD), interfața poate arăta distorsionată.
3. Dacă schimbăm textul butonului („Start” vs „Reîncepe Jocul”), acesta poate ieși din cadru.

Soluția Qt este utilizarea **Layout-urilor**. Acestea sunt clase invizibile care calculează automat poziția și dimensiunea copiilor.

4.2.1 Tipuri de Layout-uri

- **QHBoxLayout**: Aranjează elementele orizontal, unul lângă altul.
- **QVBoxLayout**: Aranjează elementele vertical, unul sub altul.
- **QGridLayout**: Aranjează elementele într-o matrice. Este ideal pentru tabla de X și O.

Exemplu aplicat: Tabla de X și O

Pentru a crea grila de 3x3, nu vom trage manual 9 butoane în Qt Designer. Este mult mai eficient să le generăm din cod și să le punem într-un QGridLayout. Astfel, dacă mărim fereastra, butoanele se vor mări automat proporțional.

```

1 // În constructorul MainWindow
2 QWidget *centralWidget = new QWidget;
3 QGridLayout *layout = new QGridLayout;
4
5 // Generăm cele 9 butoane
6 for (int row = 0; row < 3; ++row) {
7     for (int col = 0; col < 3; ++col) {
8         QPushButton *btn = new QPushButton;
9
10        // Politica de mărime: să se extindă cât de mult posibil
11        btn->setSizePolicy(QSizePolicy::Expanding, QSizePolicy::Expanding);
12
13        // Adăugăm butonul în grilă la poziția (row, col)
14        layout->addWidget(btn, row, col);
15
16        // (Aici ar urma conectarea semnalelor - vezi Cap. 3)
17    }
18 }
19
20 centralWidget->setLayout(layout);
21 setCentralWidget(centralWidget);

```

Secvența de cod 4.1: Generarea tablei folosind QGridLayout

4.3 Componente Esențiale pentru Joc

Pentru implementarea interfeței grafice a jocului X și O, ne vom baza pe câteva componente cheie:

1. **QPushButton (Butonul):** Reprezintă o celulă a tablei.
 - **Stare:** Poate afișa text („X”, „O”) sau o iconiță.
 - **Interacțiune:** Emite semnalul `clicked()`.
 - **Proprietăți utile:** `setEnabled(false)` este crucială. După ce un jucător a marcat o căsuță, trebuie să dezactivăm butonul pentru a preveni o a doua mutare în același loc.
2. **QLabel (Eticheta):** Folosită pentru a afișa informații pasive.
 - „Este rândul lui X” sau „Câștigător: O”.
 - Poate afișa și imagini (folosind `setPixmap()`), util dacă vrem să afișăm logo-ul jocului.
3. **QMessageBox (Mesaje modale):** O clasă specială pentru afișarea notificărilor care blochează interacțiunea cu restul aplicației până la închidere.

```

1 void MainWindow::showWinner(QString winner) {
2     QMessageBox::information(this, "Joc Terminat", "Castigatorul: " + winner);
3     // După ce utilizatorul dă OK, putem reseta tabla
4 }

```

Secvența de cod 4.2: Afișarea câștigătorului

Capitolul 5

Sistemul de Resurse

Una dintre provocările majore în distribuirea aplicațiilor este gestionarea fișierelor externe (imagini, iconițe, fișiere de traducere, sunete). Dacă folosim căi relative (ex: ". /img/logo.png"), aplicația va crăpa dacă executabilul este mutat sau dacă directorul cu imagini este șters accidental.

Qt oferă o soluție robustă numită **Qt Resource System**. Aceasta permite stocarea fișierelor binare direct în executabilul final al aplicației.

5.1 Fișierele .qrc și Compilatorul de Resurse (RCC)

Sistemul se bazează pe fișiere cu extensia .qrc (bazate pe XML), care listează fișierele de pe disc ce trebuie incluse în proiect.

În timpul procesului de compilare, utilitarul **rcc** (Resource Compiler):

1. Citește fișierul .qrc.
2. Identifică fișierele menționate (imagini, etc.).
3. Le convertește în tablouri de byți (*byte arrays*) C++ statici.
4. Le compilează direct în codul binar al aplicației.

Definiție

Avantaj major: Aplicația rezultată este un singur fișier .exe (pe Windows) care conține toate imaginile necesare. Nu există riscul de "missing files" la rulare.

5.2 Organizarea și Prefixele

Fișierul de resurse (.qrc) este, în esență, un fișier XML, dar extensia Qt pentru Visual Studio oferă un editor vizual dedicat (**Qt Resource Editor**).

1. **Adăugarea fișierului:** Se face prin click dreapta pe proiect → *Add* → *New Item* → *Qt* → *Qt Resource File*.
2. **Editarea:** Un dublu-click pe fișierul .qrc din *Solution Explorer* va deschide editorul de resurse.

În interiorul editorului, organizarea se face pe două niveluri:

- **Prefixul (Prefix):** Funcționează ca un „folder virtual” pentru a grupa resursele logic. Prefixul implicit este /, dar pentru claritate este recomandat să creai prefixe explicite (ex: /img, /data).

- **Fișierele (Files):** Se adaugă sub un anumit prefix. *Atenție: Asigurați-vă că imaginile sunt copiate fizic în folderul proiectului înainte de a le adăuga în editorul de resurse.*

De exemplu, pentru jocul X și O, structura logică din editor va arăta astfel:

```
/ (prefixul rădăcină)
|-- images (prefix adăugat manual)
|-- x_symbol.png
|-- o_symbol.png
|-- reset_icon.png
```

5.3 Accesarea Resurselor din Cod

Pentru a utiliza o resursă în C++, folosim o cale specială care începe cu caracterul două puncte (:).

- Cale fizică (NU o folosim): "C:/Proiecte/TicTacToe/x.png"
- Cale relativă (Riscont): "images/x.png"
- **Cale resursă (Corect):** " :/images/x_symbol.png"

```
1 // Încărcarea unei imagini din resurse
2 QPixmap pixmap(":/images/x_symbol.png");
3
4 if (pixmap.isNull()) { qDebug() << "Eroare: Imaginea nu a fost gasita în resurse!"; }
5
6 // Setarea imaginii pe un label
7 ui->labelStatus->setPixmap(pixmap);
```

Secvența de cod 5.1: Încărcarea unei imagini într-un QPixmap

5.4 Exemplu Aplicat: Îmbunătățirea aspectului X și O

Până acum, butoanele noastre afișau text ("X" sau "O"). Folosind sistemul de resurse, putem folosi imagini stilizate.

Clasa QPushButton are metode dedicate pentru iconițe:

```
1 void MainWindow::updateButton(QPushButton* btn, char player) {
2     // Curățăm textul vechi
3     btn->setText("");
4
5     if (player == 'X') { btn->setIcon(QIcon(":/images/x_symbol.png")); }
6     else { btn->setIcon(QIcon(":/images/o_symbol.png")); }
7
8     // Setăm mărimea iconiței (altfel va fi foarte mică)
9     btn->setIconSize(QSize(32, 32));
10 }
```

Secvența de cod 5.2: Setarea iconițelor pe butoanele de joc

De reținut

De reținut: Atunci când modificați sau adăugați fișiere noi în .qrc, este uneori necesar să rulați **Run qmake** din meniul *Build* pentru ca sistemul de build să observe noile fișiere și să regenereze codul C++.

Partea III

Studiu de Caz Complet: X și O

Capitolul 6

Implementarea Jocului X și O

În acest capitol punem cap la cap conceptele discutate anterior pentru a finaliza aplicația.

6.1 Structura Proiectului

Separarea în două module: `TicTacToeLogic` (C++ pur/Qt non-GUI) și `TicTacToeGui` (Interfața).

6.2 Implementarea Business Logic

Matricea internă, verificarea condițiilor de victorie.

6.3 Integrarea Logică - UI

Instantțierea clasei de logică în `MainWindow` și conectarea semnalelor.

Partea IV

Concepte Avansate pentru Proiecte Complex

Capitolul 7

Graphics View Framework

Până acum am construit interfețe folosind widget-uri standard (`QPushButton`, `QLabel`) aranjate în layout-uri. Această abordare este excelentă pentru ferestre de dialog și formulare, dar devine limitativă pentru aplicații grafice complexe, cum ar fi jocurile de strategie sau de cărți.

Când aveți nevoie să gestionați un număr mare de obiecte 2D personalizate, care trebuie să suporte interacțiuni complexe (drag & drop, coliziuni, zoom, rotații), soluția oferită de Qt este **Graphics View Framework**.

7.1 Arhitectura Model-View (Scenă-Vizualizare)

Graphics View urmează un model arhitectural similar cu MVC, împărțit în trei componente distincte:

1. **Scena** (`QGraphicsScene`): Este containerul de date (Modelul). Ea ține evidența tuturor obiectelor, gestionează starea lor și detectează coliziunile. Scena este invizibilă; ea există doar în memorie.
2. **Vizualizarea** (`QGraphicsView`): Este componenta GUI (Widget-ul) care afișează conținutul scenei pe ecran. Putem avea mai multe vizualizări pentru aceeași scenă (ex: vedere principală și mini-map).
3. **Elementele** (`QGraphicsItem`): Sunt „actorii” de pe scenă (cărți de joc, pioni, jetoane).

Definiție

Gândiți-vă la **Scenă** ca la o lume virtuală infinită, iar la **View** ca la o cameră video care se plimbă prin această lume și o proiectează pe monitor.

7.2 Configurarea Scenei și a View-ului

Deoarece `QGraphicsView` moștenește din `QWidget`, el poate fi adăugat în fereastra principală la fel ca orice alt buton sau layout.

```
1 // În MainWindow.cpp (constructor)
2
3 // 1. Creăm scena
4 // Definim spațiul de coordonate (x, y, width, height)
5 m_scene = new QGraphicsScene(0, 0, 800, 600, this);
6
7 // 2. Creăm view-ul care va afișa scena
8 m_view = new QGraphicsView(m_scene, this);
9
```

```

10 // 3. Setăm proprietăți de performanță și aspect
11 m_view->setRenderHint(QPainter::Antialiasing); // Linii fine
12 m_view->setHorizontalScrollBarPolicy(Qt::ScrollBarAlwaysOff);
13 m_view->setVerticalScrollBarPolicy(Qt::ScrollBarAlwaysOff);
14
15 // 4. Adăugăm view-ul în layout-ul ferestrei
16 ui->mainLayout->addWidget(m_view);

```

Secvența de cod 7.1: Inițializarea Graphics View în MainWindow

7.3 Crearea Elementelor Personalizate (QGraphicsItem)

Deși Qt oferă elemente standard (ex: `QGraphicsPixmapItem` pentru imagini), pentru un joc veți dori să creați propriile clase (ex: `CardItem`).

Pentru a crea un element propriu, trebuie să moșteniți clasa `QGraphicsItem` și să suprascrieți două metode pure virtuale:

- `boundingRect()`: Definește „zona sensibilă” a obiectului (pentru click-uri și redesenare).
- `paint()`: Definește cum arată obiectul (desenarea efectivă).

Observație

Atenție la coordonate! În interiorul metodelor `paint` și `boundingRect`, coordonatele sunt locale (relative la centrul sau colțul obiectului^a), nu globale (relative la scenă).

^aOriginea sistemului de coordonate local (0,0) este definită de modul în care implementați `boundingRect()`. Dacă returnați `QRectF(0, 0, w, h)`, originea este în colțul stânga-sus. Dacă returnați `QRectF(-w/2, -h/2, w, h)`, originea este în centrul geometric al obiectului. Această alegere este critică pentru transformări: un obiect se rotește întotdeauna în jurul originii sale (0,0).

```

1 // CardItem.h
2 class CardItem : public QGraphicsItem
3 {
4     public:
5     CardItem();
6
7     // 1. Returnează aria ocupată de carte (hitbox)
8     QRectF boundingRect() const override {
9         return QRectF(0, 0, 100, 150); // 100x150 pixeli
10    }
11
12    // 2. Desenează conținutul
13    void paint(
14        QPainter *painter,
15        const QStyleOptionGraphicsItem *option,
16        QWidget *widget
17    ) override {
18        // Desenăm conturul
19        painter->setBrush(Qt::white);
20        painter->setPen(Qt::black);
21        painter->drawRect(0, 0, 100, 150);
22
23        // Aici am putea desena o imagine (QPixmap)
24        // painter->drawPixmap(...)
25    }
26 };

```

Secvența de cod 7.2: Exemplu schelet pentru o clasă CardItem

7.4 Interacțiunea: Drag and Drop simplificat

Unul dintre cele mai mari avantaje ale Graphics View este că oferă funcționalitate de *Drag and Drop* „la cheie”, fără a fi nevoie să calculați manual poziția mouse-ului.

Pentru a face un element (o carte de joc) să poată fi mutat cu mouse-ul, trebuie doar să activăm un flag în constructorul acestuia:

```
1 // În constructorul CardItem::CardItem()
2
3 // Permite selectarea obiectului
4 setFlag(ItemIsSelectable);
5
6 // Permite mutarea obiectului cu mouse-ul
7 setFlag(ItemIsMovable);
```

Secvența de cod 7.3: Activarea mutării cu mouse-ul

Odată setat acest flag, Qt se ocupă automat de:

1. Detectarea click-ului pe obiect (folosind `boundingRect`).
2. Urmărirea cursorului.
3. Actualizarea poziției obiectului în scenă.
4. Redesenarea scenei.

7.4.1 Detectarea Click-ului (fără mutare)

Dacă doriți să reacționați doar la click (de exemplu, pentru a întoarce o carte sau pentru a o juca), trebuie să suprascrieți metoda `mousePressEvent`.

```
1 void CardItem::mousePressEvent(QGraphicsSceneMouseEvent *event) {
2     qDebug() << "Cartea a fost apasata!";
3
4     // IMPORTANT: Apelăm implementarea de bază pentru a păstra
5     // funcționalitatea de selecție/mutare dacă este activă
6     QGraphicsItem::mousePressEvent(event);
7 }
```

De reținut

Dacă arhitectura aplicației voastre necesită ca obiectele grafice să comunice prin **Semnale și Sloturi** (ex: `cardClicked()`), clasa voastră trebuie să moștenească din `QGraphicsObject` în loc de `QGraphicsItem`.