Ghid de Arhitectură și Implementare Qt pentru Aplicații C++

C++ Modern Aplicat în Inteligența Artificială

Asist. Mrd. Andrei Rozmarin
Asist. Mrd. Cezar Constăndoiu

Anul II – Semestrul I | 2025 – 2026

Facultatea de Matematică și Informatică
Universitatea *Transilvania* din Brașov

Cuprins

Partea I

Configurarea Mediului de Dezvoltare

Capitolul 1

Instalare și configurare Qt pentru Visual Studio 2022/2026

1. Se descarcă Qt Online Installer pentru platforma dorită.
2. Dacă nu aveți deja unul, creați-vă un cont Qt folosind adresa de e-mail de student.
3. În programul de instalare, selectați opțiunea **Custom Installation**.
4. Instalați doar **Qt 6.10.1 MSVC 2022 64-bit**.
5. Deschideți Visual Studio și accesați **Extensions** > **Manage Extensions**.
6. Căutați și instalați extensia **'Qt Visual Studio Tools'**.
7. După instalare, navigați la **Extensions** > **Qt VS Tools** > **Qt Versions**.
8. Apăsați pe **Add**, iar pentru câmpul **Location** selectați executabilul `qmake.exe` din locația unde ați instalat Qt. Calea ar trebui să fie similară cu:
   `C:\Qt\6.10.1\msvc2022_64\bin\qmake.exe`
9. Deschideți proiectul *TicTacToe*, selectați *TicTacToeUI* ca proiect de startup și rulați. Erorile ce pot apărea (neidentificarea directorului `include` sau a fișierelor `.dll` din `bin`) sunt cauzate de căi configurate greșit; verificați secțiunea **'Qt Project Settings'**.
10. Creați un proiect nou de tip **'Qt Widgets Application'**.
11. La secțiunea **'Build Configurations'** ar trebui să aveți deja selectată versiunea de Qt instalată anterior. Dacă nu apare, folosiți butonul **Add** pentru a selecta din nou fișierul `qmake.exe`.
12. În configurarea ulterioară a proiectului, recomandăm să folosiți opțiunea **'Member Pointer'** pentru setarea **'Ui Class Inclusion'**.

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, introspecție[1] 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 0 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 dupliceze și conexiunile?

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.

```cpp
// game_logic.h
#ifndef GAMELOGIC_H
#define GAMELOGIC_H

#include <QObject>

class GameLogic : public QObject
{
    Q_OBJECT // Macro OBLIGATORIU pentru semnale/sloturi

    public:
    explicit GameLogic(QObject *parent = nullptr);
    // ... metodele jocului ...

    signals:
    // Semnal emis când jocul s-a terminat
    void gameOver(QString winner);
};

#endif // GAMELOGIC_H
```

Secvența de cod 2.1: Exemplu din clasa GameLogic (X și 0)

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 0:

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 0, le pasăm pointerul către părinte în constructor (sau prin `setParent`).

```
// Varianta C++ Standard (necesită delete manual sau vector de smart pointers)
QPushButton* btn = new QPushButton("X");
// Dacă uităm 'delete btn', avem memory leak.

// Varianta Qt (recomandată)
// 'this' este fereastra sau containerul părinte
QPushButton* btn = new QPushButton("X", this);

// Chiar dacă nu apelăm 'delete btn', acesta va fi șters automat
// 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 0, 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 apăsat 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 0 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.

```
class GameLogic : public QObject
{
    Q_OBJECT

public:
    // ...

signals:
    // Emis când un jucător a mutat valid
    // Interfața va folosi asta pentru a desena X sau 0 pe buton
    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`.

```
// În constructorul MainWindow
// Când butonul este apăsat (signal), logica resetează matricea (slot)
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 0. 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:

```
// Presupunem că buttons[3][3] este matricea de butoane din GUI

for(int i = 0; i < 3; ++i) {
    for(int j = 0; j < 3; ++j) {

        // Conectăm fiecare buton individual
        // [=] capturează i și j prin valoare pentru a fi folosite în lambda
        connect(buttons[i][j], &QPushButton::clicked, this, [=]() {

            // Când butonul (i,j) este apăsat, apelăm logica
            // cu coordonatele corecte
            m_gameLogic->makeMove(i, j);

        });
    }
}
```

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.

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

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 0, 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 0.

Exemplu aplicat: Tabla de X și 0

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.

```
// În constructorul MainWindow
QWidget *centralWidget = new QWidget;
QGridLayout *layout = new QGridLayout;

// Generăm cele 9 butoane
for (int row = 0; row < 3; ++row) {
    for (int col = 0; col < 3; ++col) {
        QPushButton *btn = new QPushButton;

        // Politica de mărime: să se extindă cât de mult posibil
        btn->setSizePolicy(QSizePolicy::Expanding, QSizePolicy::Expanding);

        // Adăugăm butonul în grilă la poziția (row, col)
        layout->addWidget(btn, row, col);

        // (Aici ar urma conectarea semnalelor - vezi Cap. 3)
    }
}

centralWidget->setLayout(layout);
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 0, ne vom baza pe câteva componente cheie:

1. **`QPushButton` (Butonul)**: Reprezintă o celulă a tablei.
   - **Stare:** Poate afișa text („X", „0") 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. **`QLable` (Eticheta)**: Folosită pentru a afișa informații pasive.
   - *„Este rândul lui X"* sau *„Câștigător: 0"*.
   - 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.

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

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 creați 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 0, 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 (Riscant): `"images/x.png"`
- **Cale resursă (Corect):** `":/images/x_symbol.png"`

```
// Încărcarea unei imagini din resurse
QPixmap pixmap(":/images/x_symbol.png");

if (pixmap.isNull()) { qDebug() << "Eroare: Imaginea nu a fost gasita în resurse!"; }

// Setarea imaginii pe un label
ui->labelStatus->setPixmap(pixmap);
```

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

5.4 Exemplu Aplicat: Îmbunătățirea aspectului X și 0

Până acum, butoanele noastre afișau text (”X” sau ”0”). Folosind sistemul de resurse, putem folosi imagini stilizate.

Clasa `QPushButton` are metode dedicate pentru iconițe:

```
void MainWindow::updateButton(QPushButton* btn, char player) {
    // Curățăm textul vechi
    btn->setText("");

    if (player == 'X') { btn->setIcon(QIcon(":/images/x_symbol.png")); }
    else { btn->setIcon(QIcon(":/images/o_symbol.png")); }

    // Setăm mărimea iconiței (altfel va fi foarte mică)
    btn->setIconSize(QSize(32, 32));
}
```

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 0

Capitolul 6

Implementarea Jocului X și 0

Link repository

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

6.1 Structura Proiectului

Soluția este organizată în două proiecte distincte pentru a asigura o separare clară a responsabilităților între logica jocului și interfața cu utilizatorul (UI). Această abordare modulară este esențială pentru mentenabilitate, testabilitate și reutilizarea codului.

- `TicTacToe`: Acesta este proiectul *backend*. El conține toată logica fundamentală a jocului X și 0. Este scris în C++ standard și nu are nicio dependență de biblioteci grafice sau de interfață (precum Qt). Responsabilitățile sale includ gestionarea stării tablei de joc, validarea mutărilor, alternarea jucătorilor și detectarea condițiilor de victorie, remiză sau continuare a jocului.
- `TicTacToeUI`: Acesta este proiectul *frontend*, responsabil cu prezentarea vizuală a jocului și interacțiunea cu utilizatorul. Este o aplicație Qt care consumă logica expusă de proiectul `TicTacToe`. El se ocupă de randarea tablei de joc, preluarea input-ului de la utilizator (click-uri pe celule) și afișarea stării curente a jocului (e.g., cine a câștigat).

Dependența este unidirecțională: `TicTacToeUI` depinde de `TicTacToe`, dar nu și invers. Proiectul de logică nu are nicio cunoștință despre existența unei interfețe grafice, ceea ce îi permite să fie reutilizat cu ușurință în alte contexte, cum ar fi o aplicație consolă, o interfață web sau un set de teste unitare automate.

6.1.1 Structura Fișierelor

Organizarea fișierelor reflectă această separare a responsabilităților:

```
/TicTacToe           # Proiectul de logică (backend)
|-- /include         # Headere
|   |-- /interfaces
|   |   `-- IListener.h
|   |-- /module   # Pentru scopuri demonstrative, n-am reușit să
|   |   |         # compilăm module cu Qt, poate reușiți voi
|   |   |-- Board.ixx
|   |   `-- Game.ixx
|   |-- Board.h
|   |-- ForwardDeclarations.h
|   `-- Game.h
|-- /src             # Implementarea logicii
|   |-- Board.cpp
|   `-- Game.cpp
`-- Source.cpp       # Punct de intrare pentru testare consolă

/TicTacToeUI         # Proiectul de interfață Qt (frontend)
|-- /include         # Organizare Headere UI
|   |-- /viewmodel
|   |   `-- GameViewModel.h
|   |-- /views
|   |   `-- MainWindow.h
|   `-- /widgets
|       |-- BoardWidget.h
|       `-- CellWidget.h
|-- /src             # Implementare și resurse Qt
|   |-- /viewmodel
|   |   `-- GameViewModel.cpp
|   |-- /views
|   |   |-- MainWindow.cpp
|   |   |-- MainWindow.qrc
|   |   `-- MainWindow.ui
|   `-- /widgets
|       |-- BoardWidget.cpp
|       `-- CellWidget.cpp
`-- main.cpp         # Punctul de intrare al aplicației grafice
```

Fișierele din directorul `TicTacToe/include` formează contractul public al modulului de logică. Orice componentă externă (precum `TicTacToeUI`) interacționează cu logica jocului exclusiv prin intermediul acestor headere. Fișierele din `src` conțin detaliile de implementare, care pot fi modificate fără a afecta consumatorii librăriei, atâta timp cât interfața publică rămâne neschimbată.

6.1.2 Rolul Observer Pattern

Pentru a decupla complet logica de UI, este necesar un mecanism prin care *backend*-ul să poată notifica *frontend*-ul despre schimbările de stare (e.g., o mutare a fost efectuată, jocul s-a terminat) fără a-l cunoaște direct. Aici intervine design pattern-ul **Observer**.

1. **Subiectul (Subject)**: Clasa `Game` din proiectul de logică joacă rolul de subiect. Ea menține o listă de pointeri către obiecte care implementează o interfață generică de ascultător (de exemplu, `IListener`).
2. **Observatorul (Observer)**: O clasă din proiectul UI (de exemplu, `GameViewModel` sau `MainWindow`) implementează interfața `IListener`.
3. **Mecanismul**: La inițializare, obiectul din UI se "abonează" la obiectul `Game`, transmițându-i un pointer către sine. Când o acțiune relevantă are loc în `Game` (e.g., `PlayMove()`), acesta iterează prin lista sa de ascultători și apelează o metodă de notificare definită în interfață (e.g., `OnGameStateChanged()`).

Astfel, `Game` notifică "în orb" orice observator interesat, fără a avea nevoie să știe cine sunt aceștia sau ce fac cu informația primită. Această arhitectură permite ca logica jocului să rămână pură și independentă, în timp ce interfața grafică poate reacționa în timp real la evenimentele din joc, actualizându-se corespunzător.

6.1.3 Crearea Proiectelor și Dependențelor

Structura soluției cuprinde două entități distincte:

1. **Proiectul Backend (Biblioteca Statică)**:
   - Conține toate clasele de logică pură (`Board`, `Game`, `IListener`).
   - În setările proiectului (*Properties -> General*), tipul configurației este setat pe **Static library (.lib)**.
2. **Proiectul Frontend (Aplicația Qt)**:
   - Creat ca un proiect de tip *Qt Widgets Application*.
   - Acest proiect depinde direct de funcționalitățile oferite de backend.

6.1.4 Integrarea Backend-ului în Frontend

Pentru ca proiectul de Frontend să poată utiliza codul din Backend, au fost parcurși următorii pași de configurare în Visual Studio:

- **Adăugarea Referinței**: În proiectul de Frontend, s-a accesat *Add -> Reference* și s-a bifat proiectul de Backend. Acest lucru forțează Visual Studio să compileze mai întâi biblioteca și să o lege automat (*linking*) la executabilul final.
- **Configurarea Căilor de Incluziune (Include Directories)**: Pentru ca instrucțiunile `#include` să fie rezolvate corect, calea către folderul de headere al backend-ului a fost adăugată în *Properties -> C/C++ -> General -> Additional Include Directories*.
- **Alinierea Standardului C++**: Ambele proiecte au fost configurate să utilizeze același standard de limbaj (**ISO C++20 Standard** sau **C++23**) prin setarea *C++ Language Standard* în proprietățile proiectului, asigurând compatibilitatea binară între biblioteca statică și aplicație.

6.1.5 Procesul de Build

Datorită setării referințelor, procesul de compilare este simplificat: la declanșarea comenzii *Build Solution*, Visual Studio verifică dependențele, compilează sursele backend în fișierul `.lib` și apoi compilează frontend-ul, realizând legătura statică a logicii de joc în executabilul final.

6.2 Implementarea Business Logic (Backend)

Proiectul `TicTacToe` este conceput ca o bibliotecă de logică pură, independentă de interfața grafică, utilizând concepte avansate de C++ modern pentru a asigura o separare clară a responsabilităților.

6.2.1 Organizarea fișierelor

- **Board.h / Board.cpp**: Gestionează starea internă a tablei de joc folosind o matrice de tip `std::array`. Conține algoritmul de verificare a victoriei pe rânduri, coloane și diagonale prin utilizarea algoritmilor standard (`std::all_of`).
- **Game.h / Game.cpp**: Reprezintă motorul principal de joc. Acesta controlează fluxul (schimbarea jucătorului, validarea mutărilor) și deține structura `GameData`. Implementează mecanismul de notificare a interfeței prin intermediul unei liste de ascultători.
- **IListener.h**: Definește interfața abstractă (*pure virtual*) pentru pattern-ul **Observer**, permițând UI-ului să reacționeze la schimbările de stare (ex: `OnCellChanged`, `OnGameStateChanged`).
- **ForwardDeclarations.h**: Conține declarații anticipate și alias-uri de tipuri (`using`) pentru a reduce timpul de compilare și dependențele circulare.
- **Source.cpp**: Punct de intrare pentru o versiune simplificată în consolă, utilizat pentru testarea rapidă a logicii înainte de integrarea în interfața grafică.
- **Board.ixx / Game.ixx**: Fișiere de interfață de modul care permit exportarea funcționalităților către alte componente folosind noul sistem de module.

6.2.2 Elemente de C++ Modern utilizate

În implementare au fost integrate facilități moderne care sporesc siguranța tipurilor și claritatea codului:

- **Utilizarea `auto` și deducerea tipurilor**:
  - Cuvântul cheie `auto` este folosit pentru a simplifica codul și a evita repetarea tipurilor complexe, cum ar fi în cazul iteratorilor sau al rezultatelor returnate de funcții.
  - Exemplu: În `Board::CheckWinner`, `auto` preia automat tipul `std::optional<Symbol>` returnat de metodele de verificare.
- **Declararea variabilelor în condiția `if` (C++17)**:
  - Se utilizează sintaxa `if (init; condition)` pentru a limita durata de viață a variabilelor temporare.
  - Exemplu: `if (const auto& rowWinner = CheckRow(i); rowWinner.has_value())` asigură că `rowWinner` există doar în interiorul acelui bloc `if`.
- **Particularități ale Expresiilor Lambda (C++17/20)**:
  - **Mutable Lambdas**: În `CheckDiagonals`, se folosește `mutable` pentru a permite modificarea variabilelor capturate prin valoare (precum contorul n) în interiorul algoritmului.
  - **Predicate pentru Algoritmi**: Lambda-urile sunt transmise către `std::all_of` pentru a verifica starea celulelor într-un mod declarativ și concis.
- **Utilizarea `const_cast` pentru operatori**:

  - În `Board.cpp`, `const_cast` este utilizat în cadrul operatorului de indexare non-const pentru a refolosi implementarea versiunii `const`, evitând astfel duplicarea logicii de acces la date.

- **Gestiunea memoriei și a datelor**:
  - **`std::optional`**: Reprezintă celulele tablei, unde absența valorii (X sau O) indică o celulă liberă, eliminând utilizarea valorilor magice.
  - **`std::unique_ptr` & `std::make_unique`**: Gestionează ciclul de viață al obiectului `Board`, asigurând eliberarea memoriei fără intervenție manuală.
  - **`std::reference_wrapper`**: Folosit pentru `GameDataConstRef`, permițând UI-ului să vizualizeze datele jocului prin referință, având avantajul de a fi nulabil (folositor când aveți nevoie de containere de referințe).
- **Afișare de text în consolă (C++23)**:
  - Se utilizează `std::println` pentru afișarea stării jocului în consolă, oferind o alternativă mai sigură și mai rapidă la `std::cout`.

6.3 Implementarea Interfeței Grafice (Frontend)

Interfața grafică a aplicației a fost dezvoltată utilizând framework-ul **Qt 6**, adoptând un design reactiv bazat pe arhitectura **MVVM (Model-View-ViewModel)**.

6.3.1 Gestiunea Resurselor

- **Smart Pointers (`std::shared_ptr`)**: `MainWindow` și `BoardWidget` partajează posesia obiectului `GameViewModel`. Aceasta asigură că logica de business rămâne validă pe tot parcursul ciclului de viață al componentelor vizuale, prevenind accesările de memorie nevalidă.

6.3.2 Componentele Vizuale și Modern C++

- **MainWindow**: Configurează layout-ul principal și elementele de stare. Un aspect important este utilizarea `m_viewModel.get()` în cadrul funcțiilor `connect`, separând clar posesia resursei de utilizarea ei în sistemul de semnale și sloturi Qt.
- **BoardWidget și C++20 Ranges**: În loc de bucle `for` imbricate, se utilizează biblioteca `<ranges>`:
  - `std::views::join`: Permite tratarea matricei bidimensionale de celule ca pe o listă liniară (platirea structurii).
  - `std::ranges::for_each`: Aplică funcția `Reset` pe fiecare `CellWidget` într-o singură instrucțiune declarativă, eliminând erorile de indexare.
- **CellWidget**: Extinde `QPushButton` și încapsulează logica de afișare a simbolurilor. Starea internă este gestionată prin `std::optional<Symbol>`, iar dezactivarea butonului după marcare este automată, asigurând integritatea regulilor de joc.

6.3.3 Integrarea și Comunicarea (Signals & Slots)

Sistemul de comunicare este declanșat de interfața `IListener` din backend, care este implementată de `GameViewModel`:

1. **Input**: Clicul pe un `CellWidget` emite un semnal captat de `BoardWidget`, care apelează `m_viewModel->PlayMove`.
2. **Sincronizare**: ViewModel-ul primește notificarea de la motorul de joc și emite semnalul `CellChanged`.

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.

```
// În MainWindow.cpp (constructor)

// 1. Creăm scena
// Definim spațiul de coordonate (x, y, width, height)
m_scene = new QGraphicsScene(0, 0, 800, 600, this);

// 2. Creăm view-ul care va afișa scena
m_view = new QGraphicsView(m_scene, this);

```

```
// 3. Setăm proprietăți de performanță și aspect
m_view->setRenderHint(QPainter::Antialiasing); // Linii fine
m_view->setHorizontalScrollBarPolicy(Qt::ScrollBarAlwaysOff);
m_view->setVerticalScrollBarPolicy(Qt::ScrollBarAlwaysOff);

// 4. Adăugăm view-ul în layout-ul ferestrei
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).

[a] Originea 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 $(0, 0)$.

```
// CardItem.h
class CardItem : public QGraphicsItem
{
    public:
    CardItem();

    // 1. Returnează aria ocupată de carte (hitbox)
    QRectF boundingRect() const override {
        return QRectF(0, 0, 100, 150); // 100x150 pixeli
    }

    // 2. Desenează conținutul
    void paint(
        QPainter *painter,
        const QStyleOptionGraphicsItem *option,
        QWidget *widget
    ) override {
        // Desenăm conturul
        painter->setBrush(Qt::white);
        painter->setPen(Qt::black);
        painter->drawRect(0, 0, 100, 150);

        // Aici am putea desena o imagine (QPixmap)
        // painter->drawPixmap(...)
    }
};
```

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:

```
// În constructorul CardItem::CardItem()

// Permite selectarea obiectului
setFlag(ItemIsSelectable);

// Permite mutarea obiectului cu mouse-ul
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`.

```
void CardItem::mousePressEvent(QGraphicsSceneMouseEvent *event) {
    qDebug() << "Cartea a fost apasata!";

    // IMPORTANT: Apelăm implementarea de bază pentru a păstra
    // funcționalitatea de selecție/mutare dacă este activă
    QGraphicsItem::mousePressEvent(event);
}
```

De reținut

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