

# Összetett alkalmazások

# Ablakok

---

- ❑ A grafikus felületű alkalmazásokban a vezérlőket ablakokra helyezzük
  - ablaknak minősül bármely vezérlő, amely egy **QWidget**, vagy bármely leszármazottjának példánya, és nincs szülője
  - adottak speciális ablaktípusok is, pl.:
    - *üzenőablak* (**QMessageBox**), elsősorban üzenetek közlésére, vagy kérdések feltételére
    - *dialógusablak* (**QDialog**), amelynek eredménye van, elfogadható (**accept**), vagy elutasítható (**reject**)
    - *főablak* (**QMainWindow**), amely számos kiegészítést biztosít összetett ablakok megvalósítására

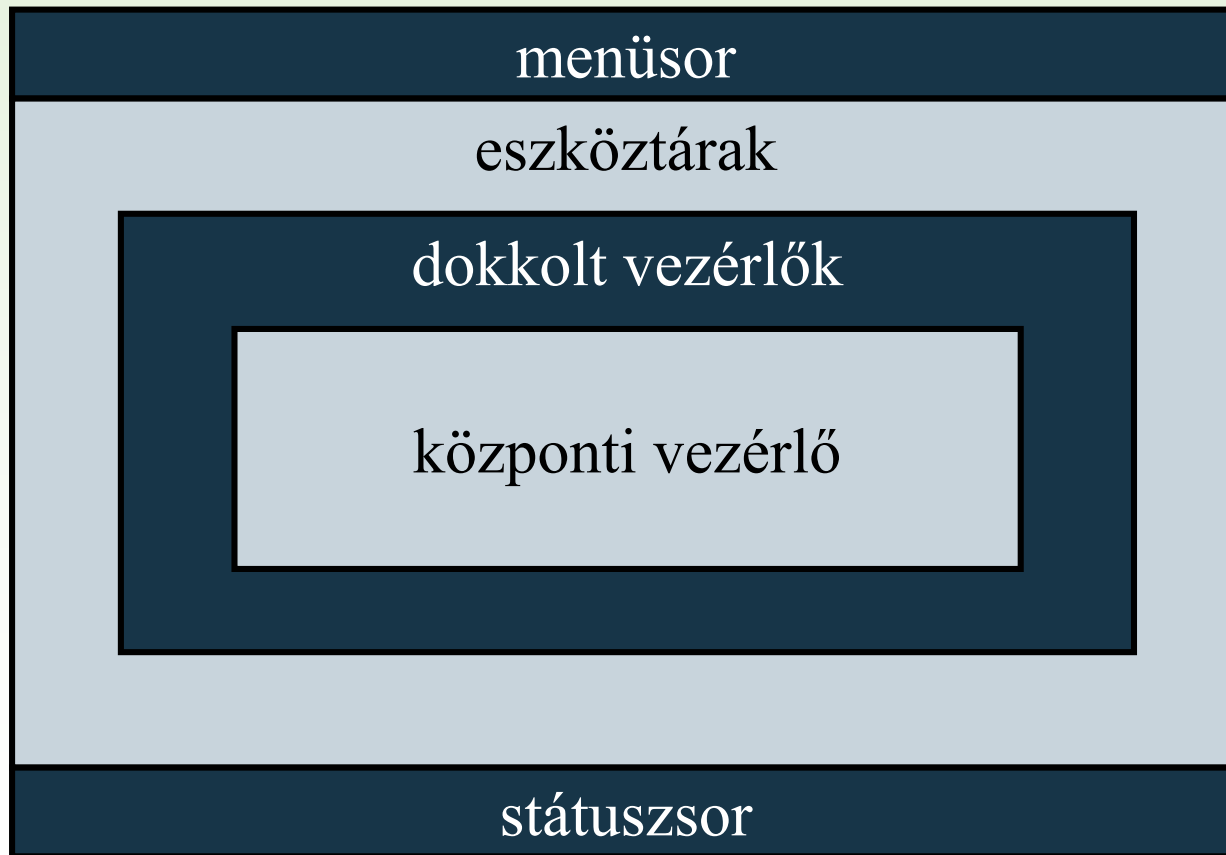
# Főablak

---

- ❑ A *főablak* (**QMainWindow**) egy olyan speciális ablaktípus, amely megkönnyíti összetett, speciális vezérlőket tartalmazó ablakok létrehozását, úgymint
  - *menüsor* (*Menu Bar*): menüpontok gyűjteménye az ablak tetején
  - *státuszsor* (*Status Bar*): állapotkijelző sor az ablak alján
  - *eszköztár* (*Toolbar*): ikongyűjteményeket tartalmazó funkciógombok, amely az ablak bármely szélére elhelyezhetők
- ❑ Az ablakon belül további vezérlőket helyezhetünk el, amelyeket dokkolhatunk az ablak széléhez, vagy középre

# Főablak

---



# Akciók

---

- ❑ A különböző vezérlők sokszor ugyanazon funkciókat biztosítják más formában (ikon, szöveg, gyorsbillentyű)
- ❑ A funkciókat egységesen *akcióként* (**QAction**) kezelhetjük, amely
  - rendelkezik felirattal (**text**), ikonnal (**icon**), gyorsbillentyűvel (**shortcut**), segédüzenettel (**statusTip**)
  - lehetőséget ad kijelölésre (**checked**), valamint billentyűs gyorsnavigálásra (az **&** karakterrel)
  - kiváltható billentyűzettel vagy egérrel, a kiváltás eseménye: **triggered**
  - felhelyezhető tetszőleges menüre, illetve eszköztárra

# Példa

---

```
// ikon és név megadása, a j billentyűre gyorsnavigál a menüben:
    QAction newAct = new QAction(QIcon("new.png"), tr("Ú&j"), this);

// a keretrendszer által kirendelt "új" billentyűkombináció:
    newAct->setShortcuts(QKeySequence::New);
        // QKeySequence(Qt::CTRL + Qt::Key_N)

    newAct->setStatusTip(tr("Új fájl létrehozása"));

// eseménykezelő társítás:
    connect(newAct, SIGNAL(triggered()), this, SLOT(newFile()));

// felhelyezés:
    fileMenu->addAction(newAct);
    fileToolBar->addAction(newAct);
```

# Menü

---

- ❑ A menüt (**QMenu**) a főablak **menuBar** tulajdonságán keresztül kezelhetjük, a menühöz felvehetünk almenüket, akciókat és elválasztókat (**separator**).
  - A menük tetszőlegesen egymásba ágyazhatóak.

```
QMenu fileMenu = this->menuBar()->addMenu(tr("&Fájl"));  
    // új almenü létrehozása  
fileMenu->addAction(newAct);  
    // menüpont felvétele  
fileMenu->addSeparator();  
    // elválasztó  
fileMenu->addMenu(tr("&Legutóbbi fájlok"));  
    // beágyazott almenü
```

# Eszköztár

---

- ❑ Eszköztárakból (`QToolBar`) tetszőlegesen sokat vehetünk fel, amelyek alapértelmezetten az ablak tetején jelennek meg.
  - Ikonok sorozatát adják, esetleges elválasztókkal szeparálva.
  - Az eszköztárak alapértelmezés szerint utólag áthelyezhetőek bármely szélére az ablaknak, illetve lehetnek lebegő (`floating`) állapotban is.

```
QToolBar fileBar = this->addToolBar(tr("Fájl"));  
    // új eszköztár felvétele  
fileBar->addAction(newAct);  
    // új akció felvétele  
fileBar->addSeparator();  
    // elválasztó
```



# Státuszsor

---

- ❑ A státuszsor (**QStatusBar**) alapvetően státuszüzenetek kiírására szolgál, ugyanakkor bármilyen vezérlő ráhelyezhető
  - üzenetet kiírni a **showMessage (<üzenet>)** utasítással tudunk, törölni a **clearMessage ()** utasítással
  - pl.: `this->statusBar () ->showMessage (tr ("Kész")) ;`

# Státuszsor és tartalom

---

- ❑ Az ablak területére célszerű egy külön vezérlőben elhelyezni a tartalmat, ez a központi vezérlő (**centralWidget**)
- ❑ Amennyiben több tartalmat helyeznénk az ablakra, lehetőségünk van azokat dokkolni a **QDockWidget** osztály segítségével, amelyet az **addDockWidget(<vezérlő>)** művelettel helyezhetünk az ablakra

# Alkalmazásszintű tulajdonságok

---

- ❑ A Qt alkalmazásokat minden esetben egy alkalmazás (`QApplication`) objektum vezérli, amely számos értéket tárol, úgymint:
  - alkalmazás információk (`applicationName`, `organizationName`, `applicationVersion`),
  - környezeti információk (`applicationDirPath`, `arguments`, `keyboardModifiers`, `clipboard`),
  - grafikus környezeti adatok (`allWindows`, `windowIcon`, `palette`, `styleSheet`, `font`).
- ❑ Az alkalmazás értékeihez bárhonnan, statikus műveletekkel hozzáférhetünk.

# Példa

---

```
QApplication::setOrganizationName("MySoft");
QApplication::setApplicationName("MyApp");
    // beállítunk némi információt
...
QString executableName = QApplication::arguments()[0];
    // lekérjük a programnevet
if (QApplication::arguments().size() > 1){
    // ha még van ezen felül argumentum
    QString arg1 = QApplication::arguments()[1];
}
```

# Beállítások kezelése

- ❑ Nagyobb alkalmazások olyan alkalmazásszintű *beállításokkal* rendelkeznek, amelyek elmenthetünk, és újabb futtatáskor betöltenünk.
- ❑ A beállítások eltárolhatóak egyedileg, de használhatjuk a beépített **QSettings** osztályt, amely egyszerűsíti a beállítások kezelését.
  - A beállítások eltárolásának módja platformonként változik (Linux esetén konfigurációs fájlok, Windows esetén regisztrációs adatbázis), ezt az osztály elfedi, így a programozónak a tárolás módjával nem kell törődnie.
  - A beállítások egy adott alkalmazásra és felhasználóra vonatkoznak.
  - A beállításokba kulcs/érték párokat vehetünk fel a **setValue(<kulcs>, <érték>)** utasítással, ahol a kulcs szöveges, az érték tetszőleges **QVariant** lehet.
    - a **value(<kulcs>)** utasítás lekérdez
    - a **contains(<kulcs>)** függvény ellenőrzi a kulcs létezését

A **QVariant** egy általános típus, amely a primitív típusokat tud „becsomagolni”, és rendelkezik konverziós műveletekkel (**toInt()**, **value<típus>()**).

# Erőforrások

---

- ❑ A főablakon használt akciókat célszerű ellátni ikonokkal, amelyeket az alkalmazáshoz kell, hogy csatoljunk
  - ❑ Az alkalmazáshoz használt ikonokat és egyéb nem kód tartalmat lehetőségünk van *erőforrásként* (*resource*) csatolni az alkalmazáshoz
    - az erőforrások tartalma belefordul a futtatandó állományba, így nem kell külön másolni őket
    - az erőforrásokat a projekthez tartozó **.qrc** fájlban nevezhetjük meg
    - az erőforrásként megadott fájlokat a **:<elérési útvonal>** hivatkozással hívhatjuk be
- ```
QIcon(":/images/new.png"); // erőforrás elérése
```

# Feladat

---

Készítsünk egy *Memory* kártyajátékot, amelyben két játékos küzd egymás ellen. A játékmezőn kártyapárok találhatók, és a játékosok feladata ezek megtalálása.

- A játék különböző kártyacsomagokkal játszható, amelyek könyvtárakból tölthetők be, minden ilyen könyvtárban található egy **name.txt**, ami a csomag nevét tartalmazza, és tetszőleges számú kép (ezek a kártyák), valamint egy hátlap (**back** fájlnevével).
- Lehetőségünk van egy beállító ablakban megadni a kiválasztott kártyacsomagot, valamint a játéktábla méretét (csak páros méretű, de legalább 4 kártyából álló lehet), valamint a játékosok neveit.

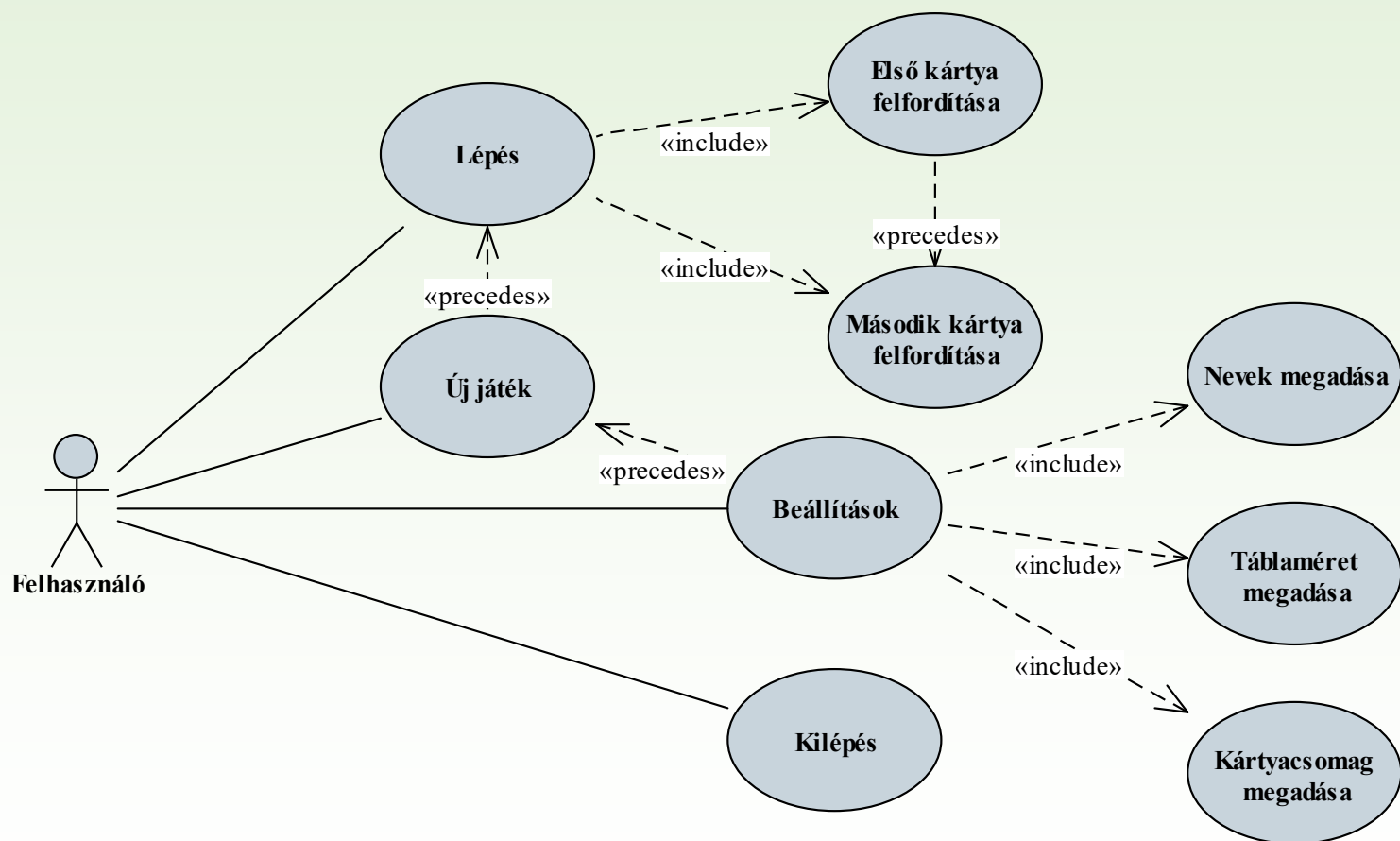
# Feladat folytatása

---

- Kezdetben minden kártya le van fordítva, a játékosok felváltva lépnek, minden lépésben felfordíthatnak két kártyát.
- Amennyiben a kártyák egyeznek, úgy felfordítva maradnak és a játékos ismét léphet, különben 1 másodperc múlva visszafordulnak, és a másik játékos következik.
- A játékot az nyeri, aki több kártyapárt talált meg.
- Megnyert játékok számát göngyölítve jelenítjük meg, amíg új játékosokat nem állítunk be.
- A felületen folyamatosan megjelenítjük a játékosok adatait (sikeres, sikertelen lépések száma, megnyert játszmák száma).



# Elemzés



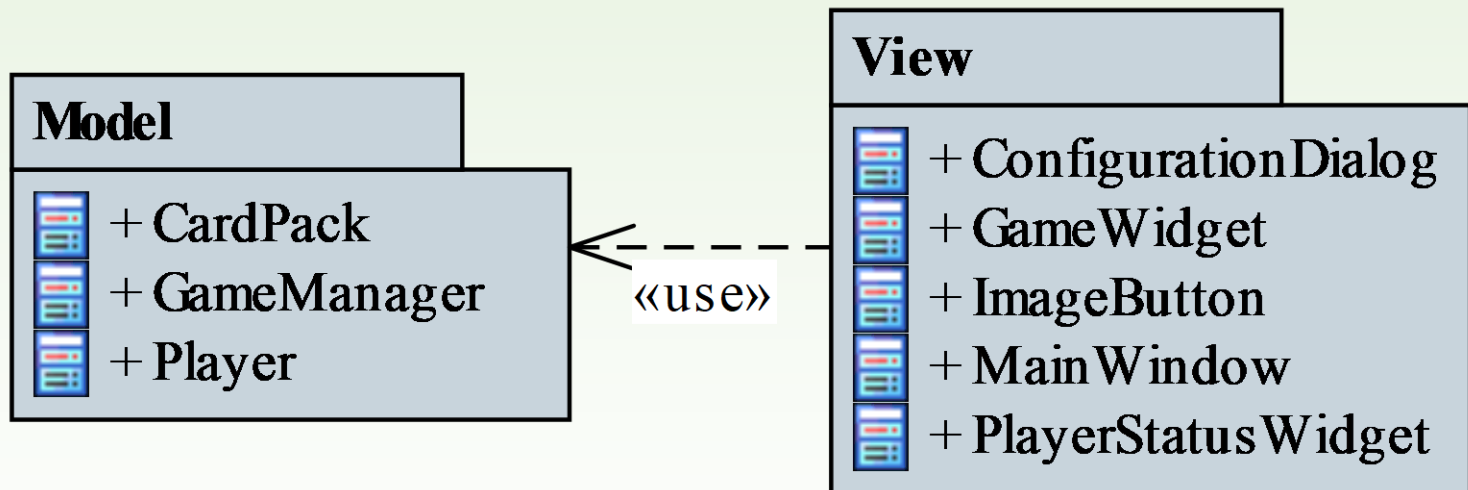
# Architektúra

---

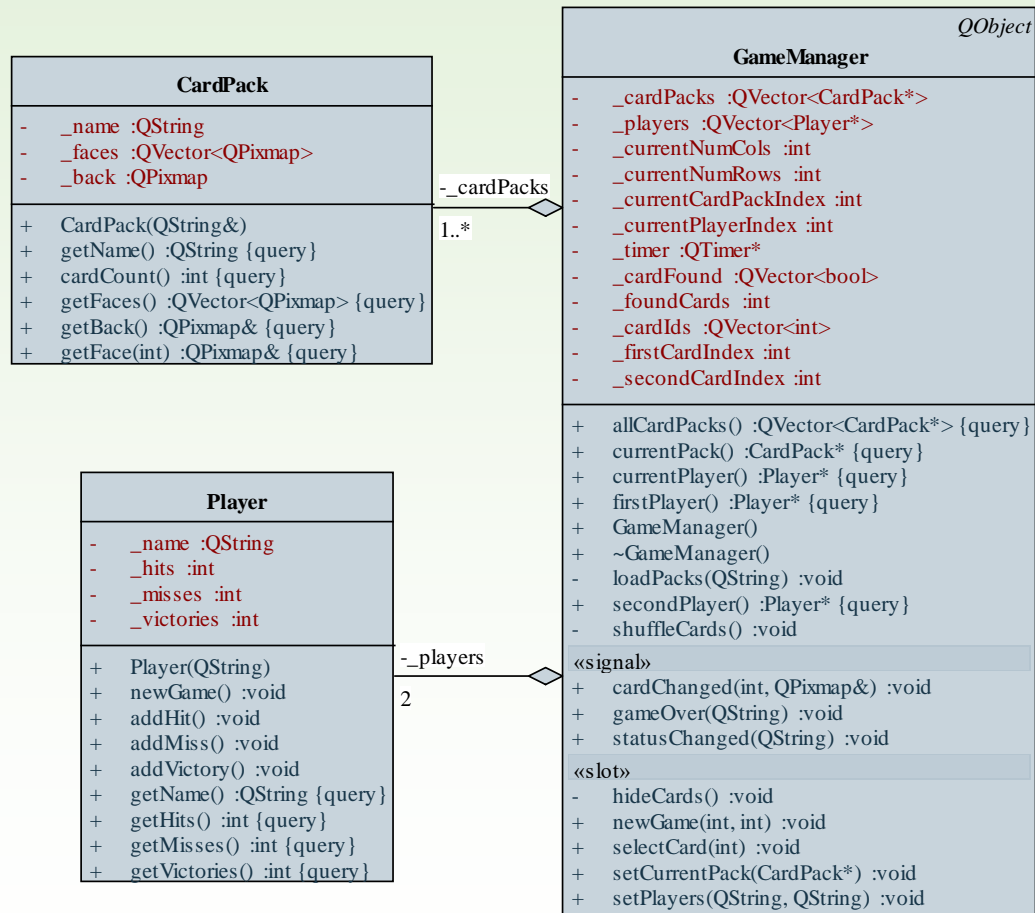
- ❑ A játékot kétrétegű (M/V) architektúrában valósítjuk meg.
- ❑ A modell tartalmazza:
  - magát a játékot, amit egy kezelőosztály felügyel (**GameManager**), valamint hozzá segédosztályként a játékost (**Player**),
  - a kártyacsomagokat (**CardPack**).
- ❑ A nézet tartalmazza:
  - a főablakot (**MainWindow**) menüvel és státuszszorral,
  - a beállítások segédablakát (**ConfigurationDialog**),
  - a játéktér felületet megjelenítő vezérlőt (**GameWidget**), amely tartalmazza a játékmezővel kapcsolatos tevékenységeket,
  - a felhasználói információkat kiíró vezérlőt (**PlayerStatusWidget**, ezt előléptetett vezérlővel állítjuk be a felülettervezőben), valamint a képet megjeleníteni tudó egyedi gombot (**ImageButton**).
- ❑ Egy csomag kártyát erőforrásként csatolunk az alkalmazáshoz (**packs.qrc**), hogy mindig legyen legalább egy ilyen.

# Csomag diagram

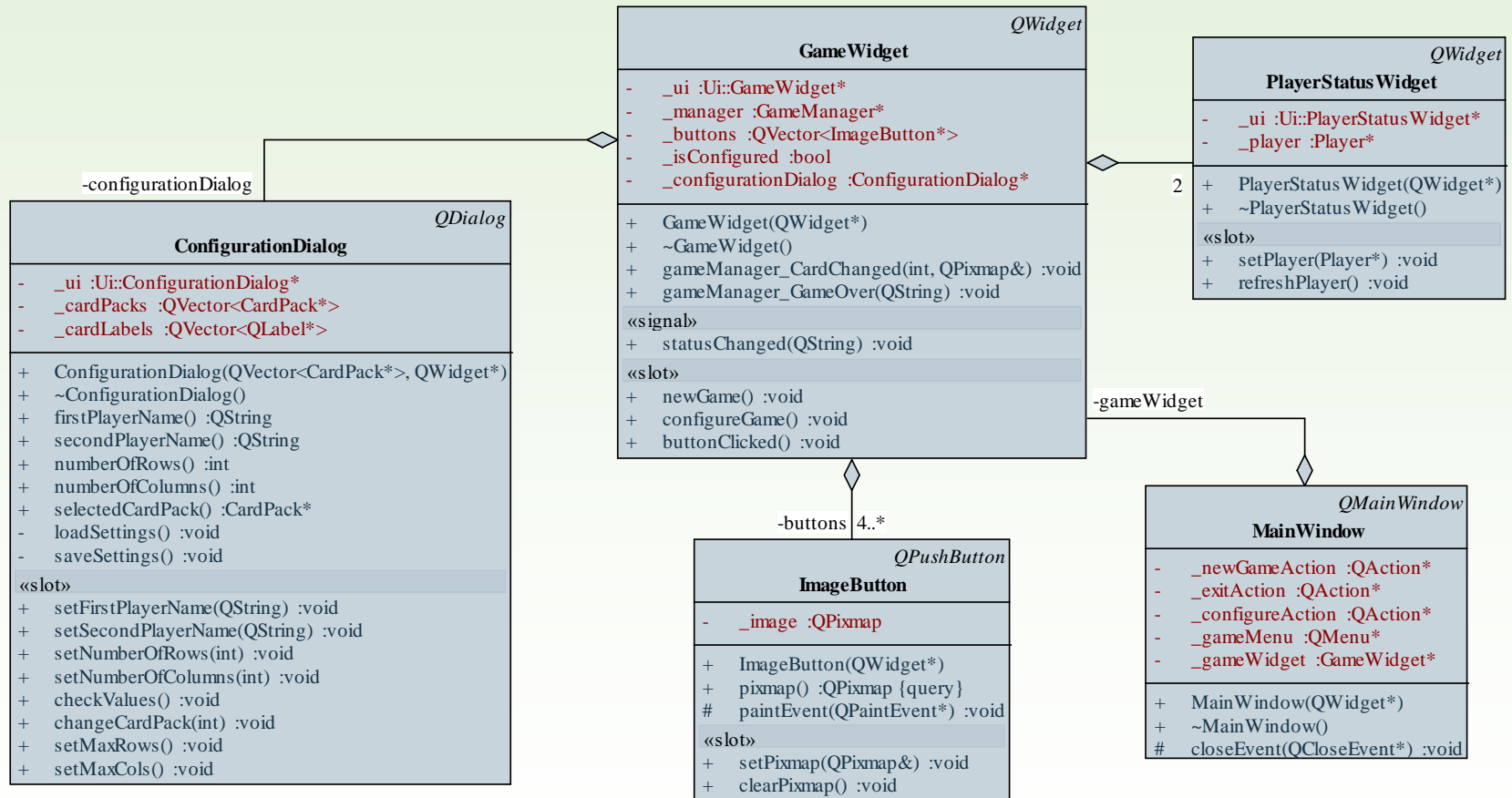
---



# Modell osztályai



# Nézet osztályai

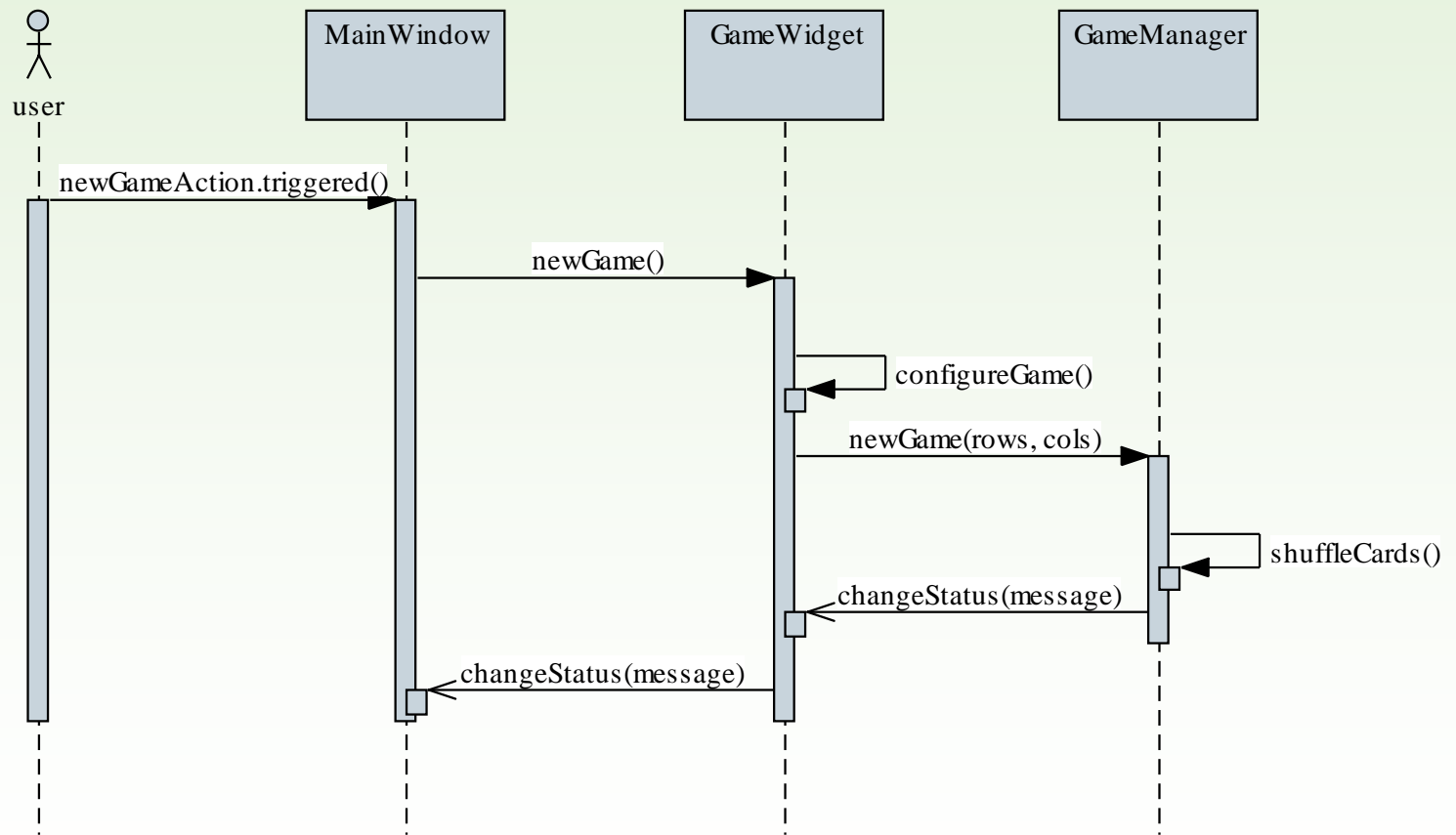


# Viselkedés

---

- ❑ Új játék indításához először a főablakban (**MainWindow**) kell kiváltanunk (**triggered**) a megfelelő akciót (**newGameAction**).
- ❑ Ennek hatására a főablak új játékot indít (**newGame**) a játék nézetében (**GameWidget**).
- ❑ A nézet beállítja a játék paramétereit (**configureGame**).
- ❑ A nézet létrehozza az új játékot (**newGame**) a modellben (**GameManager**).
- ❑ A modell megkeveri a kártyákat (**shuffleCards**), majd eseménnyel jelzi az állapot változását (**changeStatus**).

# Szekvencia diagram



# Viselkedés

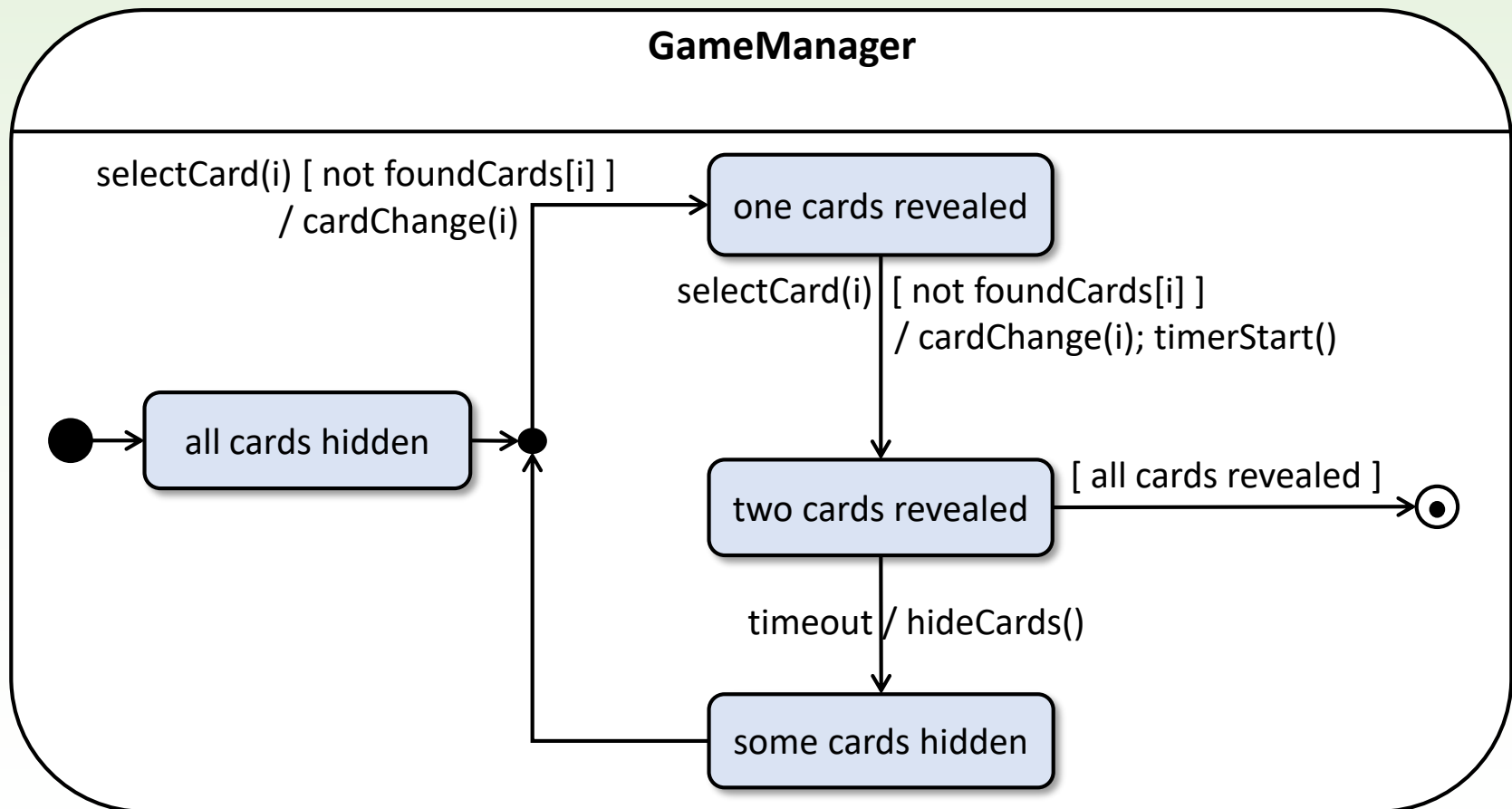
---

- ❑ Amennyiben új játékot kezdünk (**newGame**), a felület aktív lesz, játék végén (**gameOver**) pedig inaktívvá válik.
- ❑ A játék modellje kezdetben egy kártyát sem mutat, de új játék kezdésekor (**newGame**) az összes kártyát megmutatja, majd automatikusan elrejtő őket (**hideCards**).
- ❑ Kiválasztás (**selectCard**) hatására előbb egyet, majd kettőt megmutathat (**cardChanged**).
- ❑ Amennyiben a két kártya egyezik, és minden kártyát felfedtünk, vége a játéknak (**gameOver**).



# Állapotgép

bármikor kiléphetünk,  
új játékot kezdhethetünk,  
változtathatunk a beállításokon



# Megvalósítás: eseménykezelés

---

```
MainWindow::MainWindow()
{
    ...
    connect(_newGameAction, SIGNAL(triggered()),
            _gameWidget, SLOT(newGame()));
    connect(_configureAction, SIGNAL(triggered()),
            _gameWidget, SLOT(configureGame()));
    connect(_exitAction, SIGNAL(triggered()),
            this, SLOT(close()));

    connect(_gameWidget, SIGNAL(statusChanged(QString)),
            this->statusBar(), SLOT(showMessage(QString)));
}
```

# Megvalósítás: eseménykezelés

```
GameWidget::GameWidget(QWidget *parent) :
    QWidget(parent), _ui(new Ui::GameWidget)
{
    _ui->setupUi(this);
    _manager = new GameManager();
    _configurationDialog = 0;
    _isConfigured = false; // kezdetben nincs konfigurálva a játék

    connect(_manager, SIGNAL(statusChanged(QString)),
            this, SIGNAL(statusChanged(QString)));
    connect(_manager, SIGNAL(statusChanged(QString)),
            _ui->firstPlayerStatus, SLOT(refreshPlayer()));
    connect(_manager, SIGNAL(statusChanged(QString)),
            _ui->secondPlayerStatus, SLOT(refreshPlayer()));
    connect(_manager, SIGNAL(cardChanged(int,QPixmap)),
            this, SLOT(gameManager_CardChanged(int, QPixmap)));
    connect(_manager, SIGNAL(gameOver(QString)),
            this, SLOT(gameManager_GameOver(QString)));
}
```

a logikai réteg szignálja egy újabb szignált vált ki

# Megvalósítás: eseménykezelés

```
ConfigurationDialog(QVector<CardPack*> cps, QWidget *parent) :
    QDialog(parent), _ui(new Ui::ConfigurationDialog), _cardPacks(cps)
{
    _ui->setupUi(this);
    ...
    foreach(CardPack* pack, _cardPacks)
        _ui->comboCardPack->addItem(pack->getName());
    connect(_ui->comboCardPack, SIGNAL(currentIndexChanged(int)),
            this, SLOT(changeCardPack(int)));
    connect(_ui->spinRows, SIGNAL(valueChanged(int)), this, SLOT(setMaxCols()));
    connect(_ui->spinCols, SIGNAL(valueChanged(int)), this, SLOT(setMaxRows()));
    connect(_ui->buttonOk, SIGNAL(clicked()), this, SLOT(checkValues()));
    connect(_ui->buttonCancel, SIGNAL(clicked()), this, SLOT(reject()));
    if (_cardPacks.size() > 0){
        _ui->comboCardPack->setCurrentIndex(0);
        changeCardPack(0);
    }
    loadSettings();
}
```

# Megvalósítás: eseménykezelés

```
GameManager::GameManager()
{
    loadPacks(QApplication::applicationDirPath() +
              QDir::separator() + "packs");
    _currentPlayerIndex = 0;

    _timer = new QTimer(this);
    _timer->setInterval(1000);
    connect(_timer, SIGNAL(timeout()), this, SLOT(hideCards()));
}
```

```
void GameManager::newGame(int numRows, int numCols){
    ...
    statusChanged(trUtf8("Új játék elindítva, ") +
                  players[currentPlayerIndex]->getName() +
                  trUtf8(" következik."));
}
}
```

# Általános szoftver architektúrák

# Szoftverek architektúrája

---

- ❑ *Szoftver architektúrának* nevezzük a szoftver fejlesztése során meghozott *elsődleges tervezési döntések* halmazát.
  - Az architektúra létrehozása során mintákra hagyatkozunk, a szoftver teljes architektúráját definiáló mintákat nevezzük *architekturális mintáknak* (*architectural pattern*).
  - A legegyszerűbb felépítést a *monolitikus architektúra* adja, amelyben nincsenek szétválasztva a funkciók.
  - A legegyszerűbb felbontás a felhasználói felület leválasztása a háttérbeli tevékenységekről, ezt nevezzük *modell/nézet* (*MV*, *model-view*) architektúrának, vagy más néven *kétrétegű* (*two-tier*) architektúrának, ahol a két réteg egymásra épül

# Perzisztencia

---

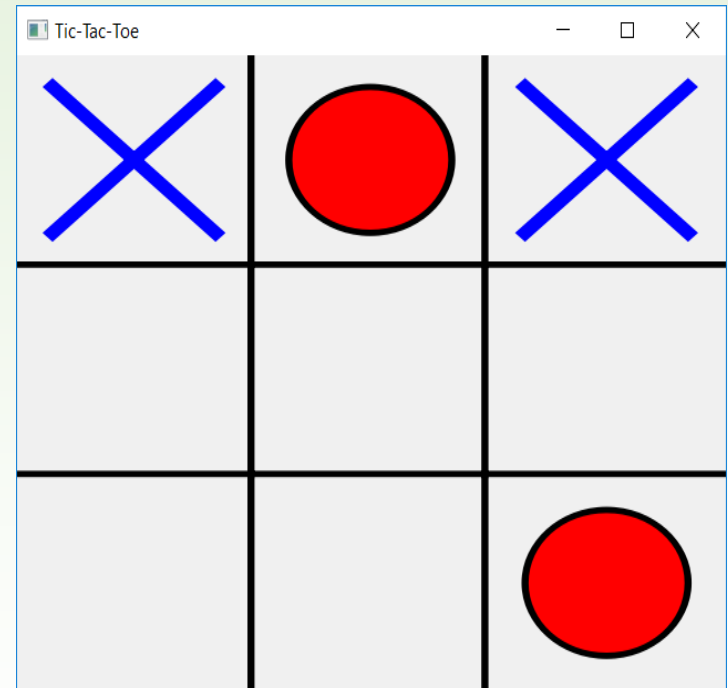
- ❑ Az adatkezelésnek egy fontos része az adatok tárolása egy *perzisztens* (hosszú távú) adattárban.
  - Az adattár lehet fájlrendszer, adatbázis, hálózati szolgáltatás, stb.
  - Az adattárolás formátuma lehet egyedi (bináris, vagy szöveges), vagy valamilyen struktúrát követő (XML, JSON, ...) annak függvényében, hogy az adatokat meg szeretnénk-e osztani már szoftverekkel.
  - A kétrétegű architektúrában a perzisztens adattárolás is a modell feladata, hiszen a modell adatait kell megfelelően eltárolnunk.



# 1.Feladat

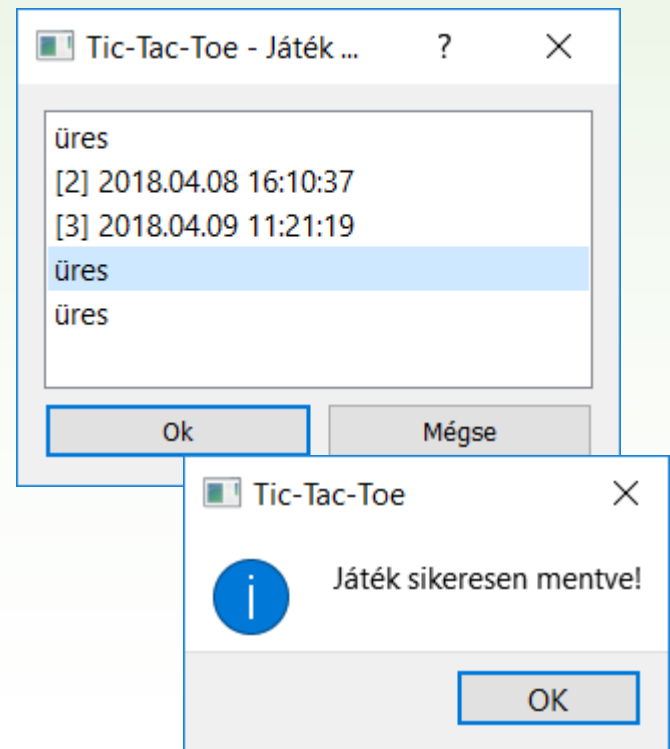
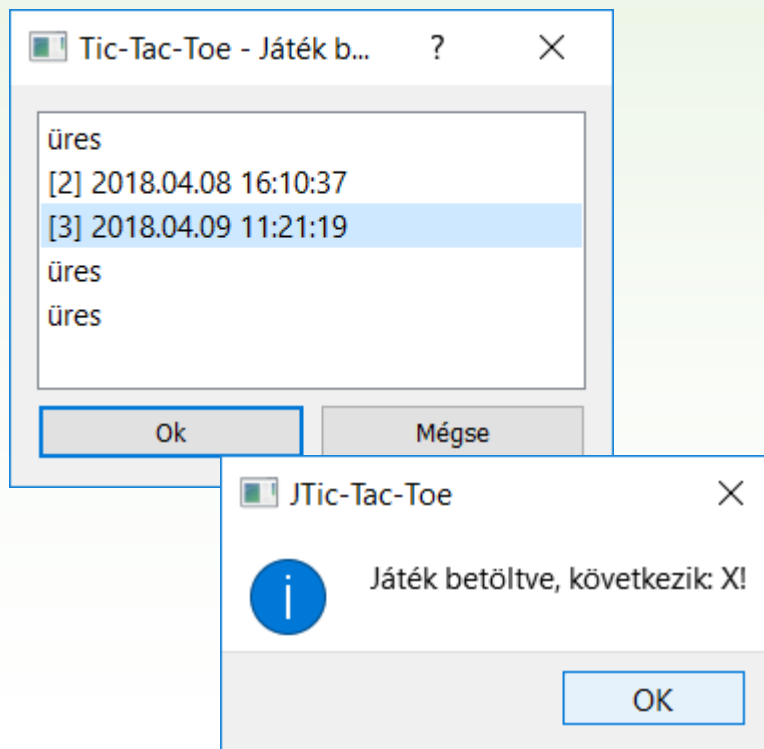
Készítsünk egy Tic-Tac-Toe programot, amelyben két játékos küzdhet egymás ellen.

- Lehetőséget adunk játékállás elmentésére (**Ctrl+L**) és betöltésére (**Ctrl+S**), ehhez a felhasználó 5 mentési hely közül választhat (egy külön ablakban).
- A mentést egyszerű szöveges fájlban végezzük (**game1.sav**, ..., **game5.sav**), elmentjük a lépésszámot, a soron következő játékost és a tábla állását.

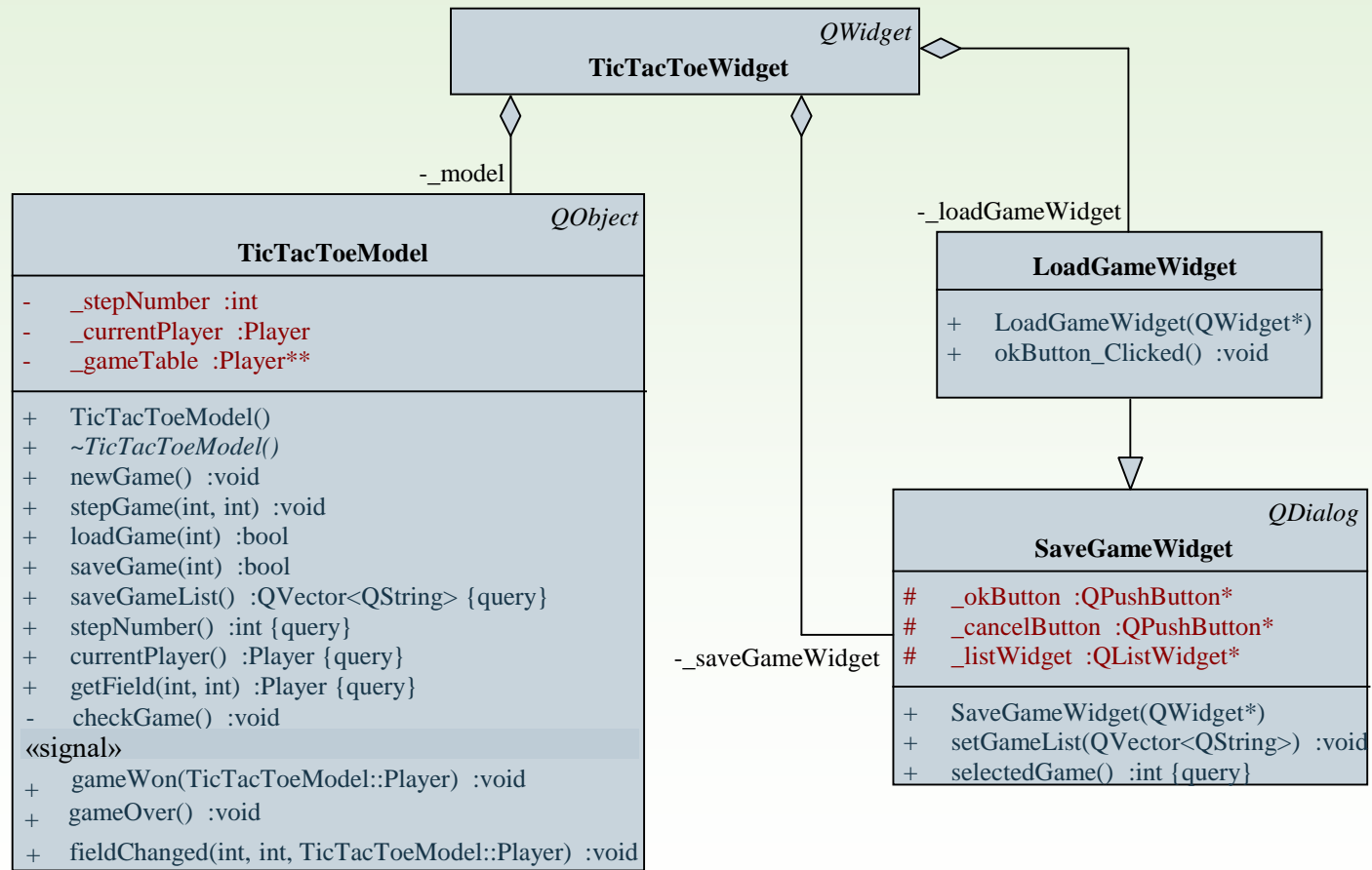


# 1.Feladat

- Létrehozunk egy betöltésre és egy mentésre szolgáló ablakot (**SaveGameWidget**, **LoadGameWidget**), a modellt pedig kiegészítjük a műveletekkel (**saveGame**, **loadGame**), valamint a játéklista lekérdezésével (**saveGameList**).



# 1.Feladat: tervezés



# 1.Feladat: modell megvalósítás

---

```
bool TicTacToeModel::saveGame(int gameIndex){
    QFile file("game" + QString::number(gameIndex)+ ".sav");
    if (!file.open(QFile::WriteOnly)) return false;

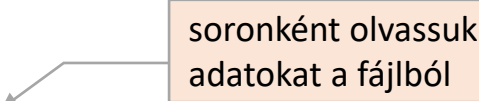
    QTextStream stream(&file);
    stream << _stepNumber << endl;
    stream << (int)_currentPlayer << endl;
    for (int i = 0; i < 3; ++i)
        for (int j = 0; j < 3; ++j) {
            stream << (int)_gameTable[i][j] << endl;
        }
    file.close();
    return true;
}
```

← soronként egy adatot írunk ki

# 1.Feladat: modell megvalósítás

```
bool TicTacToeModel::loadGame(int gameIndex)
{
    QFile file("game" + QString::number(gameIndex) + ".sav");
    if (!file.open(QFile::ReadOnly)) return false;

    QTextStream stream(&file);
    _stepNumber = stream.readLine().toInt();
    _currentPlayer = (Player)stream.readLine().toInt();
    for (int i = 0; i < 3; ++i)
        for (int j = 0; j < 3; ++j) {
            _gameTable[i][j] = (Player)stream.readLine().toInt();
        }
    file.close();
    return true;
}
```



soronként olvassuk az adatokat a fájlból

# 1.Feladat: modell megvalósítás

---

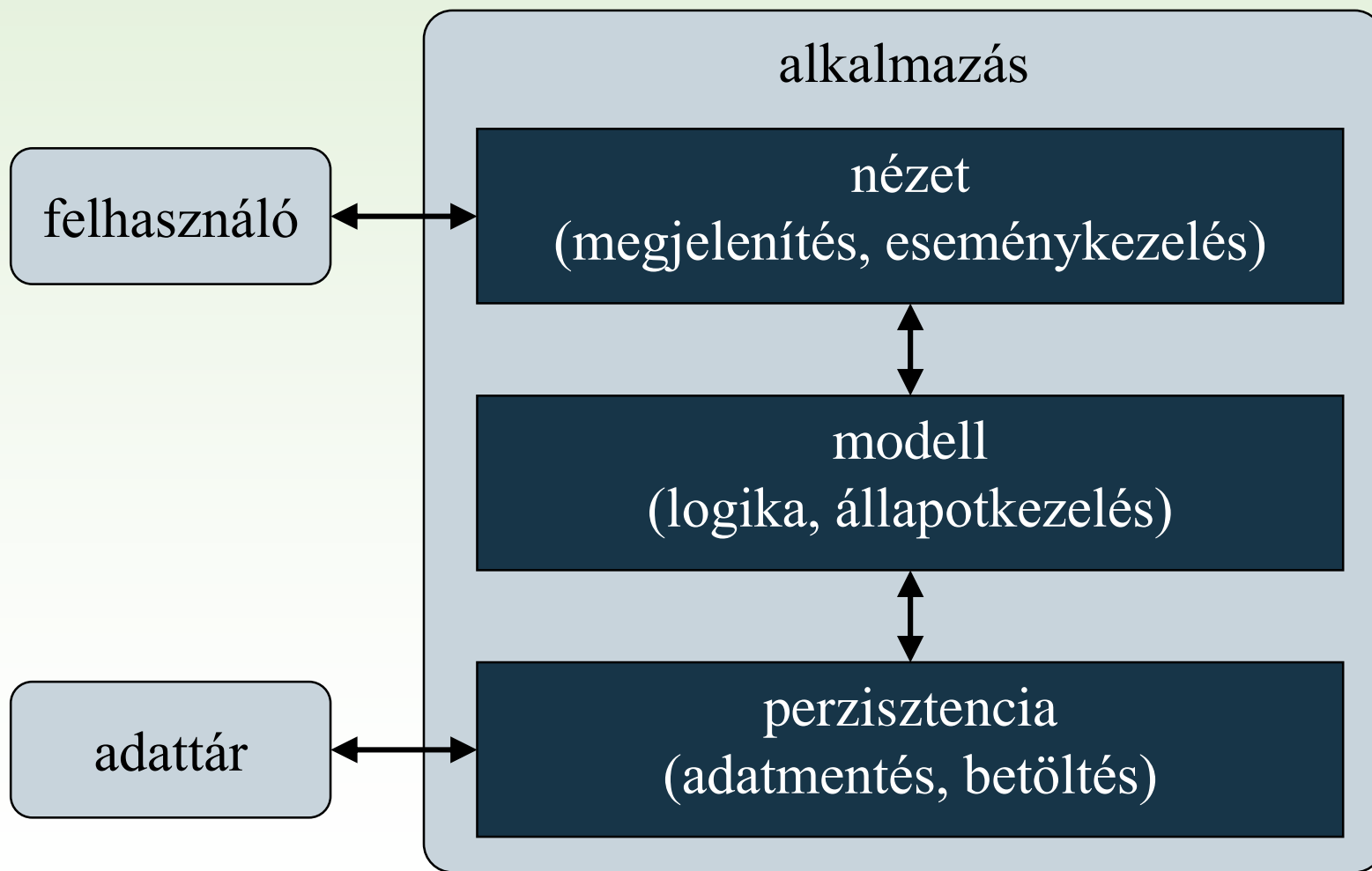
```
QVector<QString> TicTacToeModel::saveGameList() const
{
    QVector<QString> result(5);
    // végigmegyünk az 5 helyen
    for (int i = 0; i < 5; i++){
        if (QFile::exists("game" + QString::number(i) + ".sav")) {
            // ha a fájl létezik akkor betöltjük a módosítása időpontját
            QFile::Info info("game" + QString::number(i) + ".sav");
            result[i] =
                "[" + QString::number(i + 1) + "]" +
                info.lastModified().toString("yyyy.MM.dd HH:mm:ss");
        }
    }
    return result;
}
```

# Háromrétegű architektúra

---

- ❑ A perzisztens adatkezelés formája, módja nem függ a modelltől, ezért könnyen leválasztható róla, függetleníthető
  - a leválasztás lehetővé teszi, hogy a két komponenst egymástól függetlenül módosítsuk, vagy cseréljük, és egy komponensnek se kelljen több dologért felelnie (*single responsibility principle*)
- ❑ Ez elvezet minket a *háromrétegű (three-tier)* architektúrához, amelyben elkülönül:
  - a nézet (*presentation/view tier, presentation layer*)
  - a modell (*logic/application tier, business logic layer*)
  - a perzisztencia, vagy adatelérés (*data tier, data access layer, persistence layer*)

# Háromrétegű architektúra





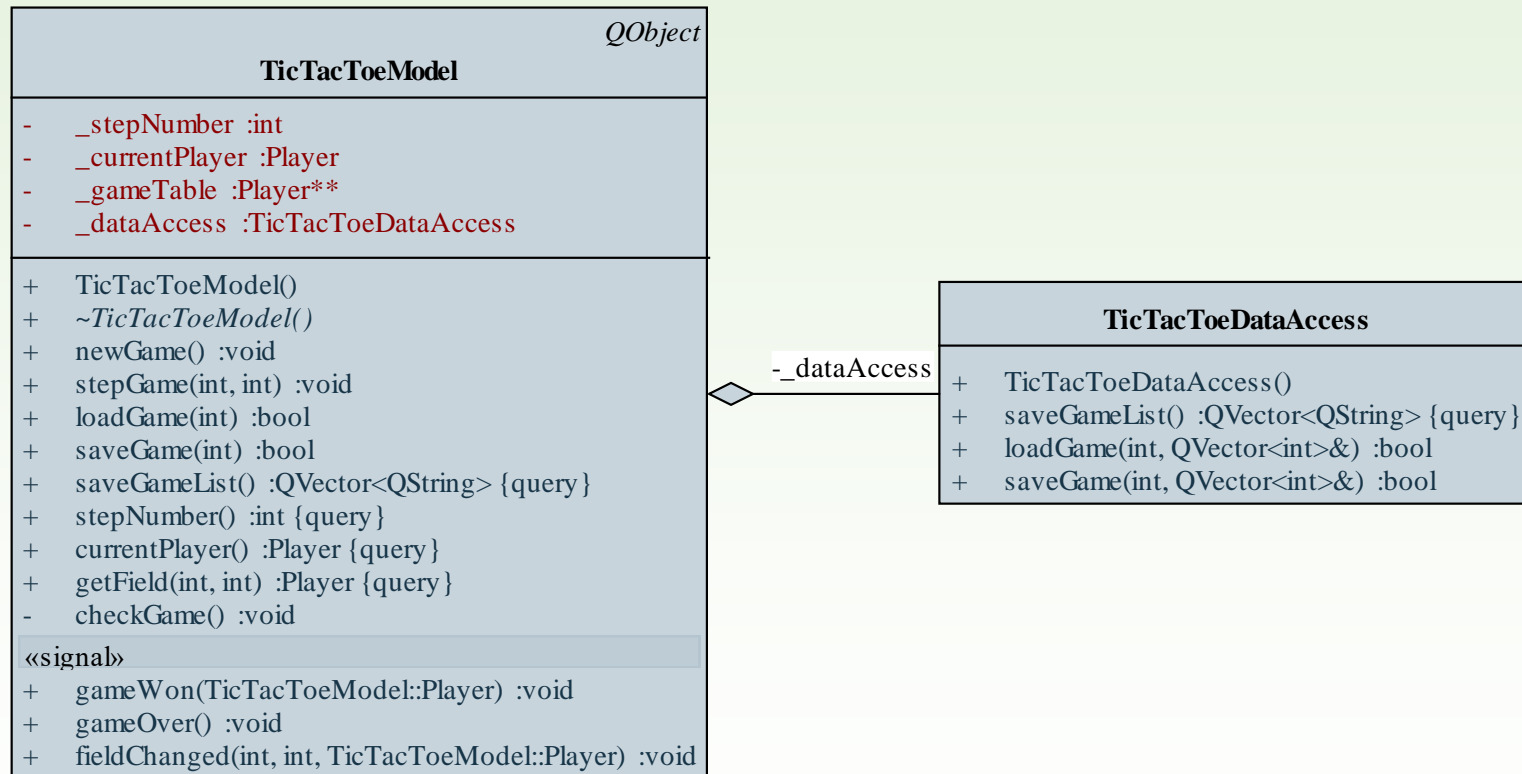
## 2.Feladat

---

Készítsünk egy Tic-Tac-Toe programot háromrétegű architektúrában.

- Leválasztjuk az adatelérést a modelltől egy új osztályba (**TicTacToeDataAccess**), amely biztosítja a három adatkezelési műveletet (**saveGame**, **loadGame**, **saveGameList**).
- Az adatok modell és adatelés közötti egyszerű kommunikáció érdekében az adatelési réteg egészek vektorát fogja kezelni, amely 11 értéket tárol a korábbi sorrendnek megfelelően (lépésszám, játékos, 9 mező sorfolytonos sorrendben).

## 2.Feladat: tervezés



## 2.Feladat: modell megvalósítás

```
bool TicTacToeModel::saveGame(int gameIndex)
{
    QVector<int> saveGameData;
    // összerakjuk a megfelelő tartalmat
    saveGameData.push_back(_stepNumber);
    saveGameData.push_back((int)_currentPlayer);
    for (int i = 0; i < 3; ++i)
        for (int j = 0; j < 3; ++j) {
            saveGameData.push_back((int)_gameTable[i][j]);
        }
    return _dataAccess.saveGame(gameIndex, saveGameData);
}
```

az adatelérés végzi a tevékenységeket

```
QVector<QString> TicTacToeModel::saveGameList() const
{
    return _dataAccess.saveGameList();
}
```

## 2.Feladat: modell megvalósítás

---

```
bool TicTacToeModel::loadGame(int gameIndex)
{
    QVector<int> saveGameData;
    if (!_dataAccess.loadGame(gameIndex, saveGameData)) return false;
    // feldolgozzuk a kapott vektort
    _stepNumber = saveGameData[0];
    _currentPlayer = (Player)saveGameData[1];
    for (int i = 0; i < 3; ++i)
        for (int j = 0; j < 3; ++j) {
            _gameTable[i][j] = (Player)saveGameData[2 + i * 3 + j];
        }
    return true;
}
```

# 3.Feladat

---

Készítsünk egy Tic-Tac-Toe programot háromrétegű architektúrában.

- Módosítsuk úgy az adatkezelést, hogy az adatok tárolása a **game** adatbázisnak a **games** táblájában történjen, ahol a **mezők** az alábbiak: id, saveTime, stepCount, currentPlayer, tableData).
- Továbbra is 5 mentési hely lesz, és az adatokat is a korábbiaknak megfelelően mentjük (mivel nincs utolsó módosítás dátuma, ezért a mentés időpontját is a táblázatba írjuk).
- Ehhez csupán az adatelérést kell módosítanunk, a program többi része változatlan marad, felhasználjuk a Qt adatbázis modult (**QSqlDatabase, QSqlQuery**).

# 3.Feladat: adatelérés megvalósítás

---

```
bool TicTacToeDataAccess::
    saveGame(int gameIndex, const QVector<int> &saveGameData)
{
    QSqlQuery query; // kitöröljük a korábbi játékállást (ha volt)
    query.exec("remove from games where id = " + QString::number(gameIndex));
    QString tableData; // a tábla kitöltését egy adatként mentjük el
    for (int i = 2; i < 11; i++) tableData += QChar::fromLatin1(saveGameData[i]);
    // beszúrjuk az adatokat
    return query.exec(
        "insert into games (id, saveTime, stepCount, currentPlayer, tableData)
        values(" +
            QString::number(gameIndex) + ", ' " +
            QDateTime::currentDateTime().toString("yyyy.MM.dd HH:mm:ss") +
            "' , " + QString::number(saveGameData[0]) + ", " +
            QString::number(saveGameData[1]) + ", ' " + tableData + " '");
}
```

# 3.Feladat: adatelérés megvalósítás

---

```
QVector<QString> TicTacToeDataAccess::saveGameList() const
{
    QVector<QString> result(5);
    QSqlQuery query;
    query.exec("select id, saveTime from games");
    // adatbázisban futtatjuk a lekérdezést a sorszámra és a dátumra
    while (query.next()) {
        result[query.value(0).toInt()] = "[" +
            query.value(0).toString() + "]" + " " +
            query.value(1).toString();
    }
    return result;
}
```

# 3.Feladat: adatelérés megvalósítás

---

```
bool TicTacToeDataAccess::
    loadGame(int gameIndex, QVector<int> &saveGameData)
{
    QSqlQuery query;
    query.exec("select stepCount, currentPlayer,
                  tableData from games where id = " +
                  QString::number(gameIndex));
    if (!query.next()) return false; // ha nincs eredmény, nincs mentés
    saveGameData.resize(11);
    // betöltjük a mentés egyes elemeit
    saveGameData[0] = query.value(0).toInt();
    saveGameData[1] = query.value(1).toInt();
    for (int i = 0; i < 9; i++)
        saveGameData[i + 2] = query.value(2).toString()[i].toLatin1();
    return true;
}
```



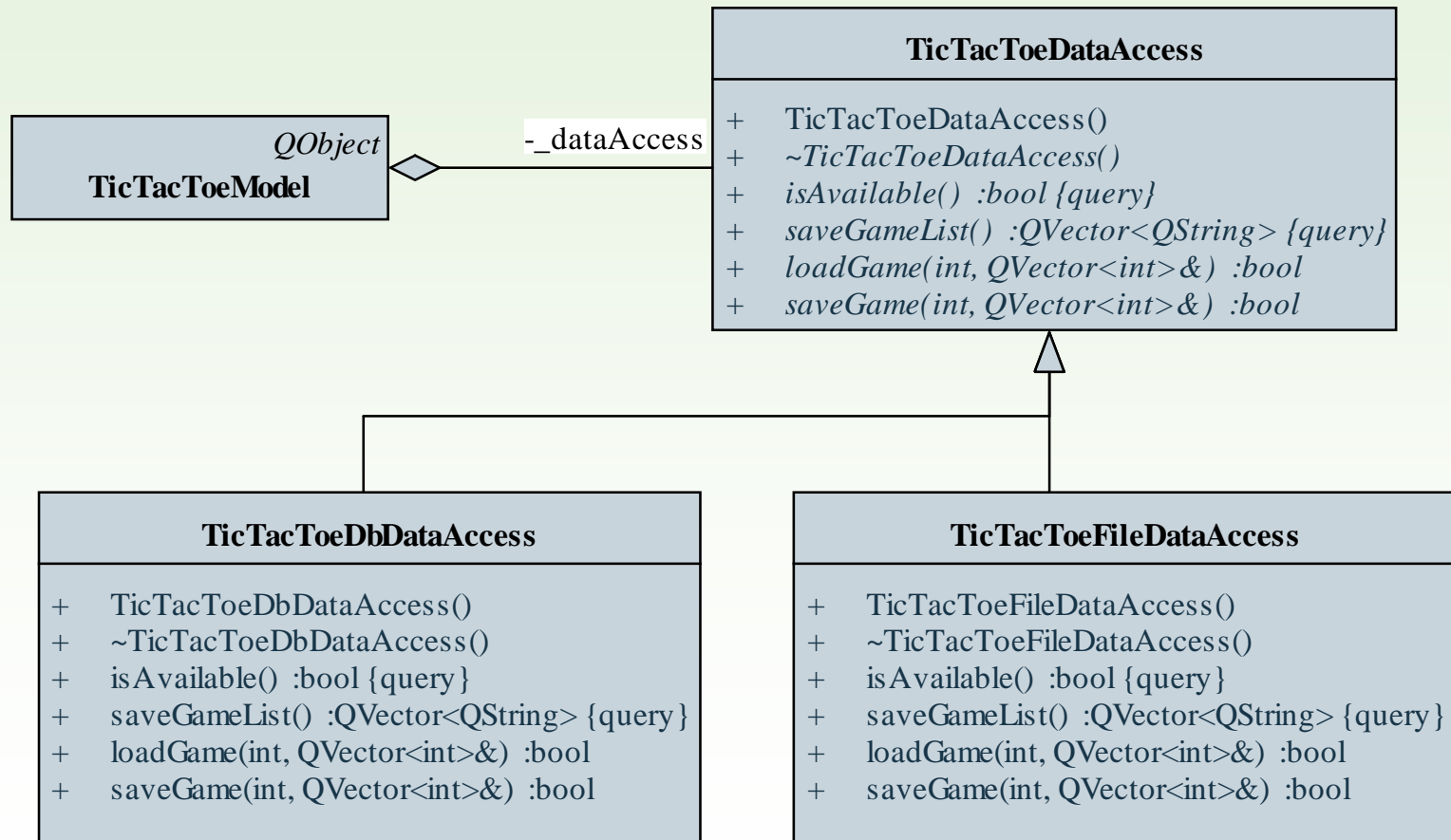
# 4.Feladat

---

Készítsünk egy Tic-Tac-Toe programot háromrétegű architektúrában.

- A program alapértelmezetten az adatbázist használja mentésre, de amennyiben az nem elérhető, használjon fájl alapú adatkezelést.
- Az adatelérés befecskendezzük a modellbe, és a nézet fogja megállapítani, milyen adatelérést adunk át.
- Az adatelérés osztályunk absztrakt lesz, és származtatjuk belőle a fájl (**TicTacToeFileDataAccess**) és adatbázis (**TicTacToeDbDataAccess**) alapú elérést.
- Az osztály kiegészül a rendelkezésre állás lekérdezésével (**isAvailable**).

# 4.Feladat: tervezés



## 4.Feladat: megvalósítás

---

```
TicTacToeWidget::TicTacToeWidget(QWidget *parent): QWidget(parent) {  
    ...  
    _dataAccess = new TicTacToeDbDataAccess();  
    // alapértelmezetten adatbázist használunk  
    if (!_dataAccess->isAvailable()){  
        // de ha az nem elérhető átváltunk fájlra  
        _dataAccess = new TicTacToeFileDataAccess();  
    }  
    _model = new TicTacToeModel(_dataAccess);  
    // a modellt létrehozuk az adateléréssel  
    ...  
}
```

# Többrétegű alkalmazások megvalósítása

---

- ❑ A függőség befecskendezés a fejlesztés során is nagyobb szabadságot ad, mivel elég a felhasznált osztály interfészét megadni az a függő osztály fejlesztéséhez.
  - Tehát a függő osztály implementációját nem zavarja a konkrét megvalósítás hiánya.
  - Azonban tesztelés csak akkor hajtható végre, ha a konkrét megvalósítás adott, ez lassíthatja a fejlesztést.
  - Ráadásul az egységtesztek esetén problémát jelenthet, ha a felhasznált osztály megvalósítása hibás, mivel így az a függő osztály is hibás viselkedést produkál (noha a hiba másik osztályban található).

# Mock objektumok

---

- ❑ Megoldást jelent, ha nem támaszkodunk a felhasznált osztály megvalósítására, hanem biztosítunk egy olyan megvalósítást, amely *szimulálja annak működését*.
  - Implementálja a felületet, így felhasználható a függő osztályban.
  - Egyszerű viselkedést biztosít, amelynek célja, hogy a függő osztály tesztelésére lehetőséget adjon.
  - Garantáltan hibamentes, így az egységteszt során valóban csak a tényleges hibákra derül fény.
- ❑ A szimulációt megvalósító objektumokat nevezzük *mock objektumoknak*

# Példa

---

```
class ServiceMock : public ServiceInterface {  
public:  
    void Provide() {  
        qDebug() << "Running service."  
    }  
}
```

mock-olást megvalósító osztály, amely  
egy egyszerű megvalósítást biztosít

ezt használjuk, és így már tesztelhető  
a Client osztály

```
Client client(new ServiceMock());
```

# 5.Feladat

---

Teszteljük le a Tic-Tac-Toe játék háromrétegű megvalósításának modelljét.

- A modell függ az adateléréstől, de azt nem akarjuk tesztelni, ezért viselkedését kiváltjuk egy mock objektummal.
- Létrehozunk egy tesztprojektet, amelyben bemásoljuk a **TicTacToeModel**, valamint **TicTacToeDataAccess** osztályokat.
- Elkészítjük a teszteseteket, amelyekhez létrehozunk egy mock objektumot az adatelérésre (**TicTacToeDataAccessMock**), amely egyszerű funkciókat biztosít, és a konzolra (**QDebug**) üzen, ennek egy példányát felhasználjuk a tesztben.

```
void TicTacToeModelTest::initTestCase() {  
    _dataAccess = new TicTacToeDataAccessMock();  
    _model = new TicTacToeModel(_dataAccess);  
}
```

# 5.Feladat: megvalósítás

---

```
class TicTacToeDataAccessMock : public TicTacToeDataAccess {
public:
    bool isAvailable() const { return true; } // rendelkezésre állás
    QVector<QString> saveGameList() const;
    bool loadGame(int gameIndex, QVector<int> &saveGameData);
    bool saveGame(int gameIndex, const QVector<int> &saveGameData);};
```

```
bool TicTacToeDataAccessMock::
    loadGame(int gameIndex, QVector<int> &saveGameData)
{
    saveGameData.resize(11); // minden érték 0 lesz
    saveGameData[1] = 1; // kivéve a rákövetkező játékos
    qDebug() << "game loaded to slot (" << gameIndex << ") with values: ";
    for (int i = 0; i < 11; i++) qDebug() << saveGameData[i] << " ";
    qDebug() << endl;
    return true;
}
```



# 5.Feladat: megvalósítás

---

```
QVector<QString> TicTacToeDataAccessMock::saveGameList() const
// mentett játékok lekérdezése
{
    return QVector<QString>(5); // üres listát adunk vissza
}
```

```
bool TicTacToeDataAccessMock::
    saveGame(int gameIndex, const QVector<int> &saveGameData)
{
    qDebug() << "game saved to slot (" << gameIndex << ") with values: ";
    for (int i = 0; i < 11; i++) qDebug() << saveGameData[i] << " ";
    qDebug() << endl;
    return true;
}
```