Három rétegű szoftverarchitektúra

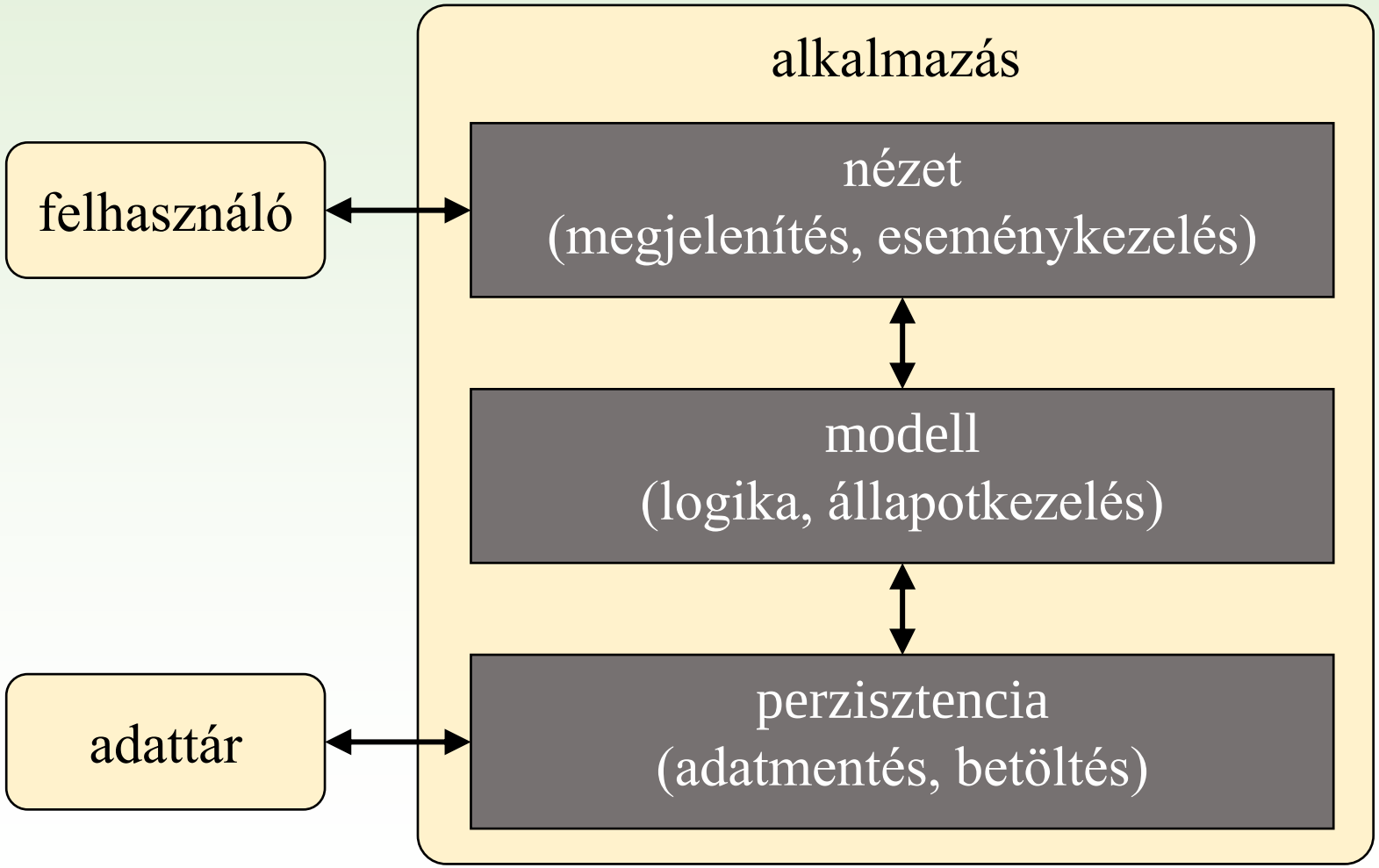
Perzisztencia

- 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 más szoftverekkel.
  - A kétrétegű architektúrában a perzisztens adattárolás is a modell feladata, hiszen a modell adatait kell megfelelően eltárolnunk.

A modell és a perzisztencia szétválasztása

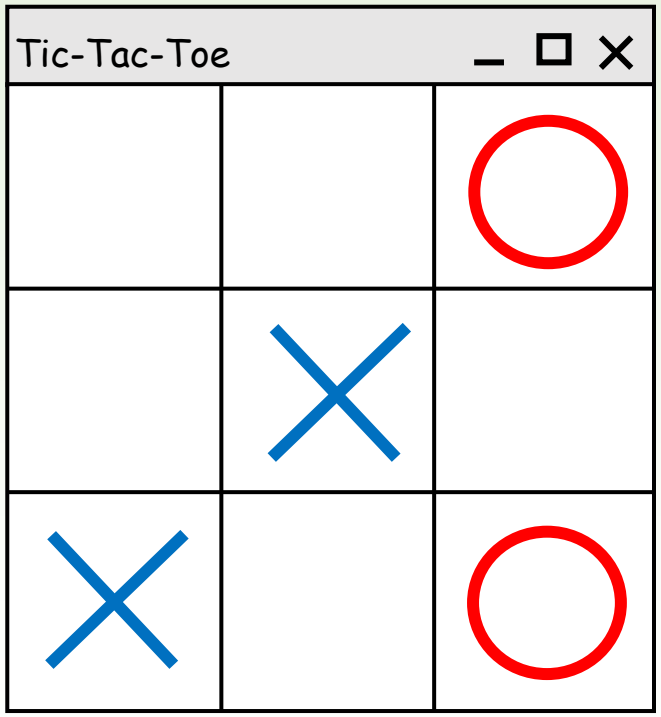
- A perzisztens adatkezelés formája, módja nem függ a modelltől, ezért könnyen leválasztható róla.
  - A leválasztás lehetővé teszi, hogy a két komponenst egymástól függetlenül módosítsuk, vagy cseréljük, és egy komponensnek se kelljen több dologért felelnie (*single responsibilty principle*) (lásd SOLID elvek)
- 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



Feladat

Egészítsük ki a korábban létrehozott Tic-Tac-Toe alkalmazást úgy, hogy lehetőséget adjon legfeljebb öt játékállás elmentésére (**`Ctrl+S`**) és visszatöltésére (**`Ctrl+L`**).

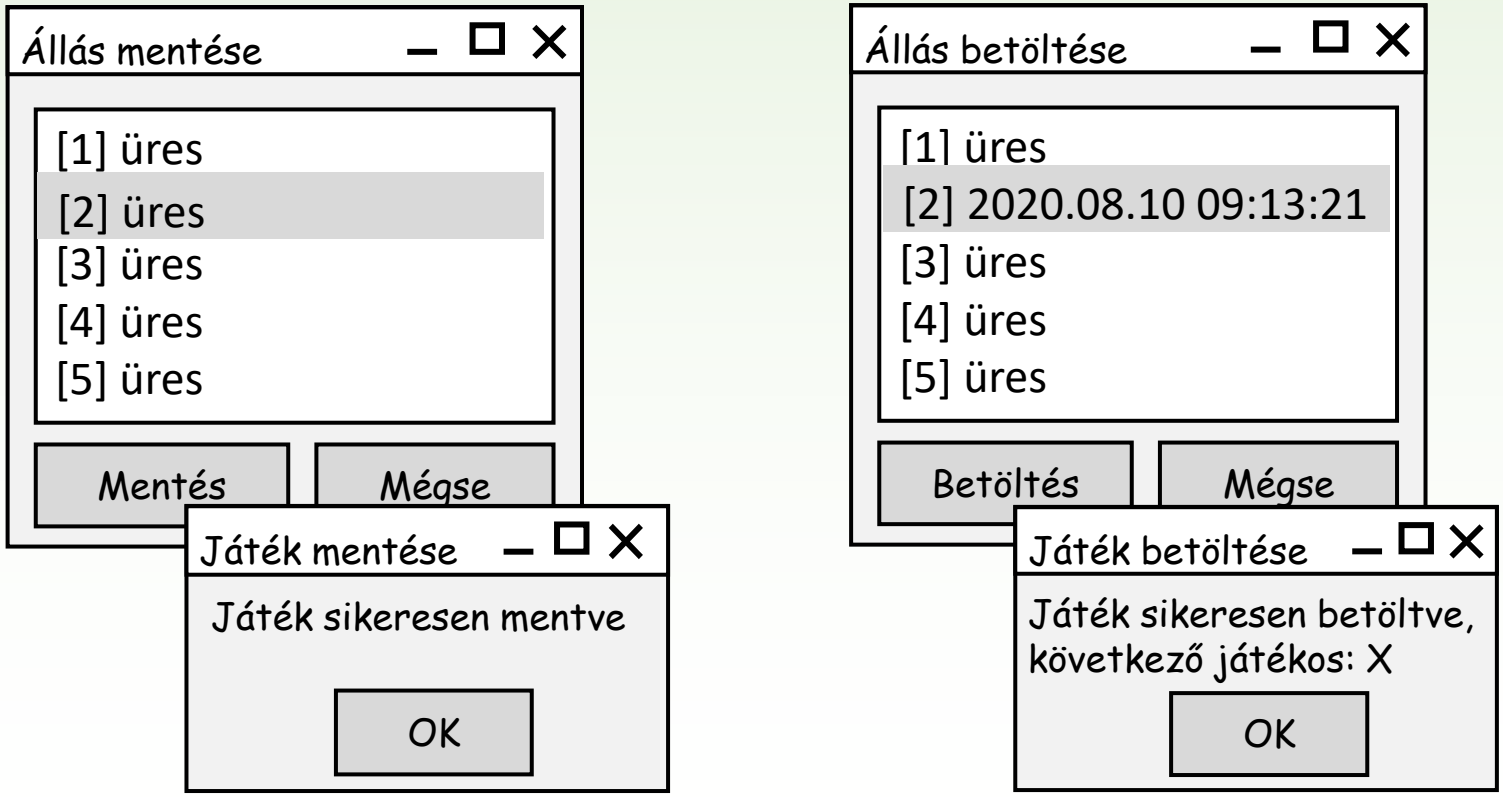


1.Megoldás

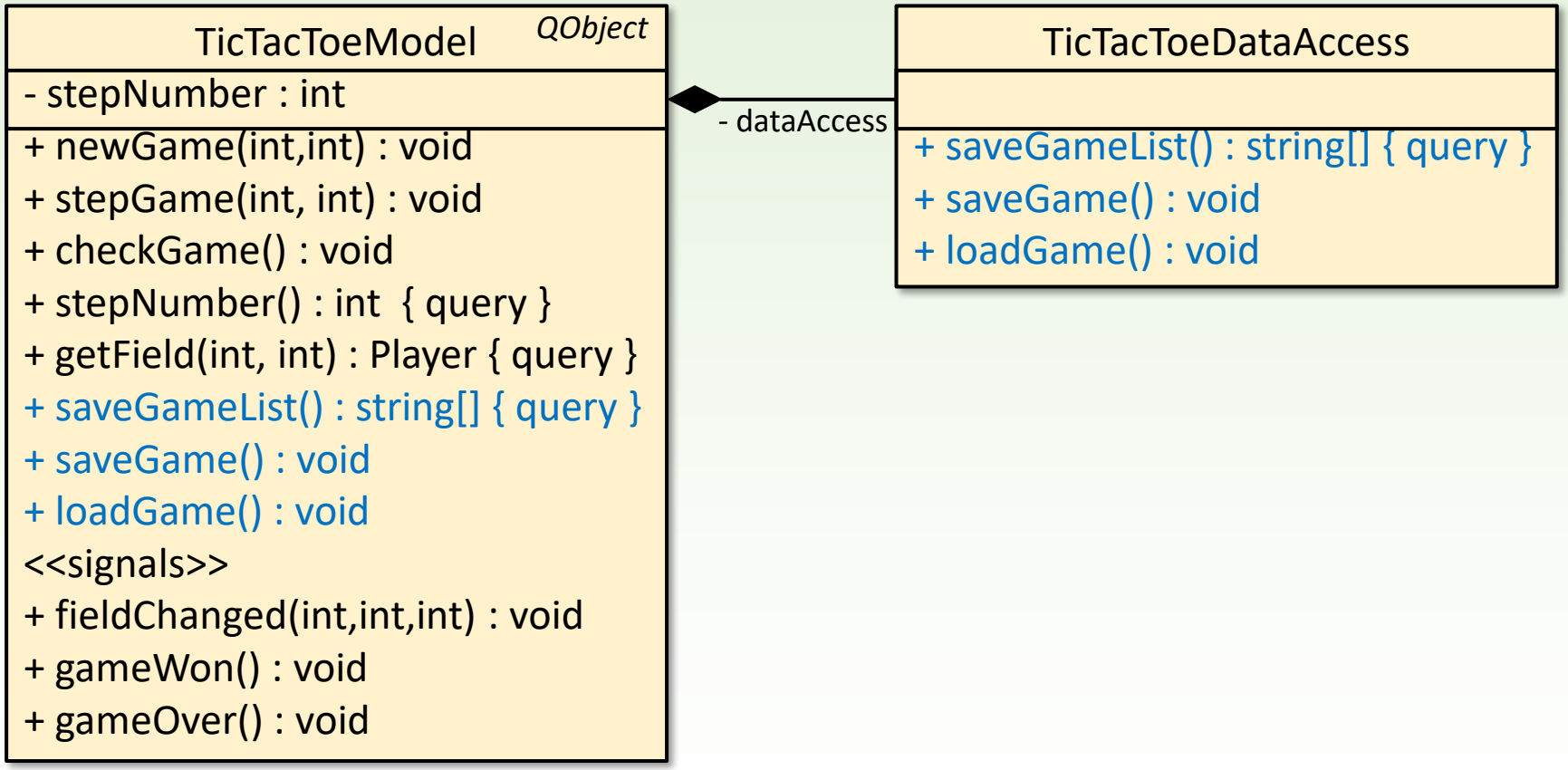
- A mentést egyszerű szöveges fájlban végezzük (**`game1.sav`**, …, **`game5.sav`**). Elmentjük a tábla állását, a soron következő játékost, és az aktuális lépésszámot.
- A felhasználó egy külön ablakban választhat sorszám alapján öt mentési hely közül.
- Alkalmazzunk háromrétegű architektúrát. Leválasztjuk az adatelérést a modellről egy új osztályba (**`TicTacToeDataAccess`**), amely biztosítja a három adatkezelési műveletet (**`saveGame`**, **`loadGame`**, **`saveGameList`**).
- A kommunikáció során a modell és a perzisztencia közötti mindig egy 11 darab egész számból álló sorozatot fogunk átadni. Ebben a sorozatban az alább leírtak sorrendjében szerepel az aktuális lépésszám, a következő játékos, és a tábla 9 mezője sorfolytonos sorrendben.

1.Megoldás: tervezés

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



1.Megoldás: tervezés



1.Megoldás: perzisztencia

```
bool TicTacToeDataAccess::saveGame(int gameIndex,
                                   const QVector<int> &saveGameData)
{
    QFile file("game" + QString::number(gameIndex) + ".sav");
    if (!file.open(QFile::WriteOnly)) return false;
    QTextStream stream(&file);
    for (int i = 0; i < 11; i++) stream << saveGameData[i] << endl;
    file.close();
    return true;
}
```

írás fájlba

```
bool TicTacToeDataAccess::loadGame(int gameIndex, QVector<int> &saveGameData)
{
    QFile file("game" + QString::number(gameIndex) + ".sav");
    if (!file.open(QFile::ReadOnly)) return false;
    QTextStream stream(&file);
    saveGameData.resize(11);
    for (int i = 0; i < 11; ++i) saveGameData[i]=stream.readLine().toInt();
    file.close();
    return true;
}
```

olvasás fájlból

1.Megoldás: perzisztencia

```
QVector<QString> TicTacToeDataAccess::savedGameList() const
{
    QVector<QString> result(5);

    for (int i = 0; i < 5; i++) {
        if (QFile::exists("game" + QString::number(i) + ".sav")) {
            QFileInfo info("game"+ QString::number(i) + ".sav");
            result[i] = "[" + QString::number(i + 1) + "] " +
            info.lastModified().toString("yyyy.MM.dd HH:mm:ss");
        }
    }
    return result;
}
```

visszaadja a mentéseket tartalmazó fájlok nevét azok mentési dátumával, de a sorozatban lehetnek „lyukak”

1.Megoldás: modell

```
bool TicTacToeModel::saveGame(int gameIndex)
{
    QVector<int> saveGameData;
    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);
}
```

a modell hívja a perzisztencia metódusát.

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

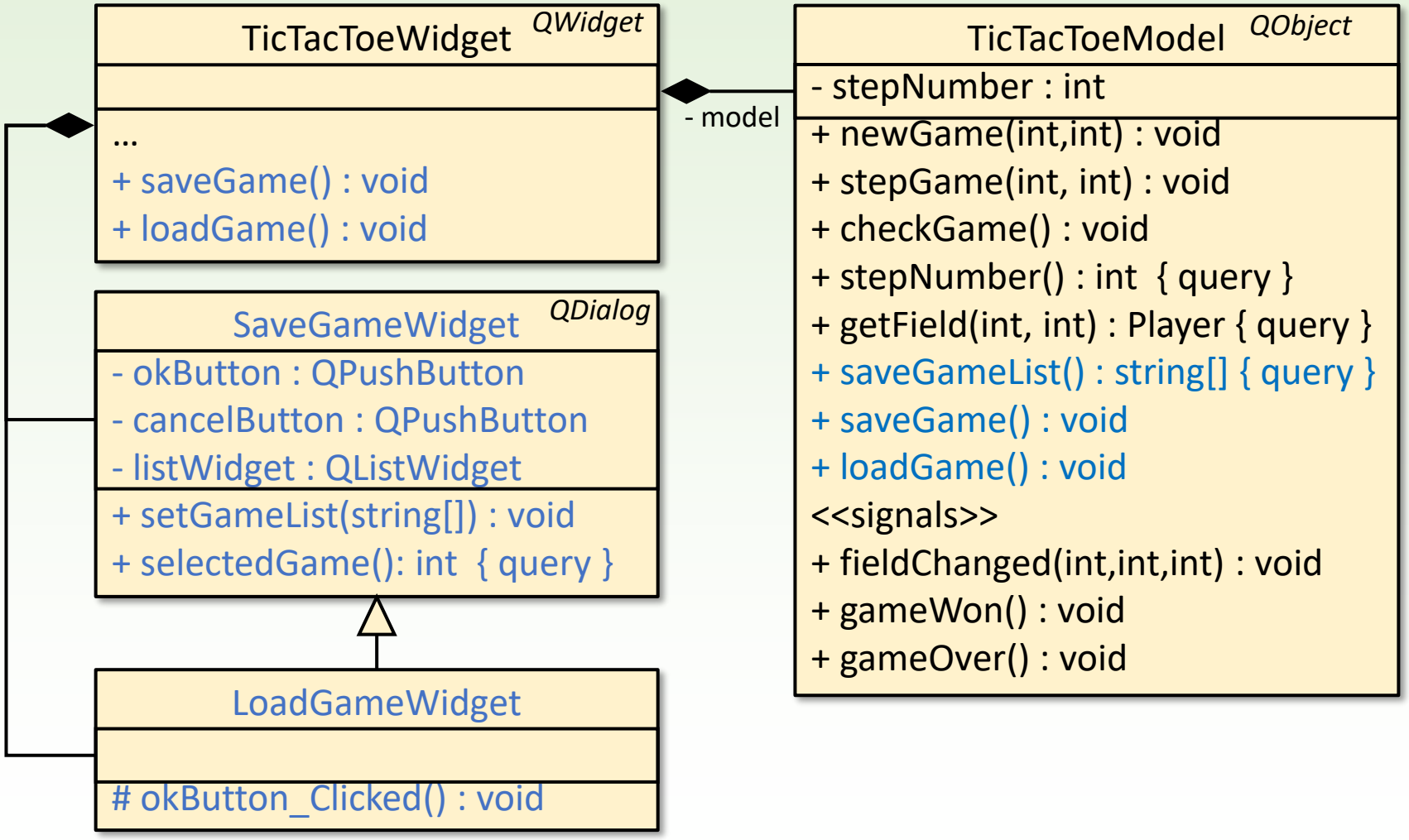
a modell hívja a perzisztencia metódusát.

1.Megoldás: modell

```
bool TicTacToeModel::loadGame(int gameIndex)
{
    QVector<int> saveGameData;
    if (!_dataAccess.loadGame(gameIndex, saveGameData)) return false;
    _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;
}
```

a modell hívja a perzisztencia metódusát

1.Megoldás: tervezés



1.Megoldás: nézet

```
void TicTacToeWidget::keyPressEvent(QKeyEvent *event)
{
    …
    // játék betöltése: Ctrl+L
    if (event->key() == Qt::Key_L
            && QApplication::keyboardModifiers() == Qt::ControlModifier) {
        if (_loadGameWidget == nullptr) {
            _loadGameWidget = new LoadGameWidget();
            connect(_loadGameWidget, SIGNAL(accepted()), this, SLOT(loadGame()));
        }
        _loadGameWidget->setGameList(_model->savedGameList());
        _loadGameWidget->open();
    }
    // játék mentése: Ctrl+S
    if (event->key() == Qt::Key_S
            && QApplication::keyboardModifiers() == Qt::ControlModifier) {
        if (_saveGameWidget == nullptr) {
            _saveGameWidget = new SaveGameWidget();
            connect(_saveGameWidget, SIGNAL(accepted()), this, SLOT(saveGame()));
        }
        _saveGameWidget->setGameList(_model->savedGameList());
        _saveGameWidget->open();
    }
}
```

dialógusablak nyitása

játék betöltése a dialógusban kiválasztott helyről

dialógusablak nyitása

játék mentése a dialógusban kiválasztott helyre

1.Megoldás: nézet

```
void TicTacToeWidget::loadGame()
{
     if (_model->loadGame(_loadGameWidget->selectedGame())) {
          update();
          QMessageBox::information(this, tr("JTic-Tac-Toe"),
            tr("Játék betöltve, következik: ") +
             ((_model->currentPlayer() == TicTacToeModel::PlayerX) ? "X" : "O")
            + "!");
     } else QMessageBox::warning(this, tr("Tic-Tac-Toe"),
                                       tr("A játék betöltése sikertelen!"));
}

void TicTacToeWidget::saveGame()
{
     if (_model->saveGame(_saveGameWidget->selectedGame())) {
          update();
          QMessageBox::information(this, tr("Tic-Tac-Toe"),
            tr("Játék sikeresen mentve!"));
     } else QMessageBox::warning(this, tr("Tic-Tac-Toe"),
               tr("A játék mentése sikertelen!"));
}
```

A betöltést a modell közvetítésével a perzisztencia végzi.

A mentést a modell közvetítésével l a perzisztencia végzi.

1.Megoldás: nézet

```
SaveGameWidget::SaveGameWidget(QWidget *parent) : QDialog(parent)
{
    setFixedSize(300, 200);
    setWindowTitle("Tic-Tac-Toe - Játék mentése");
    _listWidget = new QListWidget();
    _okButton = new QPushButton("Ok");
    _cancelButton = new QPushButton("Mégse");
    …
    connect(_okButton, SIGNAL(clicked()), this, SLOT(accept()));
    connect(_cancelButton, SIGNAL(clicked()), this, SLOT(reject()));
}

void SaveGameWidget::setGameList(QVector<QString> saveGameList)
{
    _listWidget->clear();
    // betöltjük az elemeket a listából
    for (int i = 0; i < 5; i++) {
        if (i < saveGameList.size() && !saveGameList[i].isEmpty())
            _listWidget->addItem(saveGameList[i]);
        else _listWidget->addItem("üres");
    }
}
```

Amelyik sorszámhoz nincs fájl, oda az „üres” szót írjuk.

1.Megoldás: nézet

```
LoadGameWidget::LoadGameWidget(QWidget *parent) : SaveGameWidget(parent)
{
    setWindowTitle("Tic-Tac-Toe - Játék betöltése");
    // ellenőrzést is végzünk az OK gomb lenyomására
    disconnect(_okButton, SIGNAL(clicked()), this, SLOT(accept()));
    connect(_okButton, SIGNAL(clicked()), this,  SLOT(okButton_Clicked()));
}

void LoadGameWidget::okButton_Clicked()
{
    if (_listWidget->currentItem()->text() == "üres") {
        QMessageBox::warning(this, tr("Tic-Tac-Toe"),
                                   tr("Nincs játék kiválasztva!"));
        return; // ha üres mezőt választott
    }
    accept(); // különben elfogadjuk a dialógust
}
```

Ne válasszunk a betöltéshez nem létező fájlt!

2.Megoldás

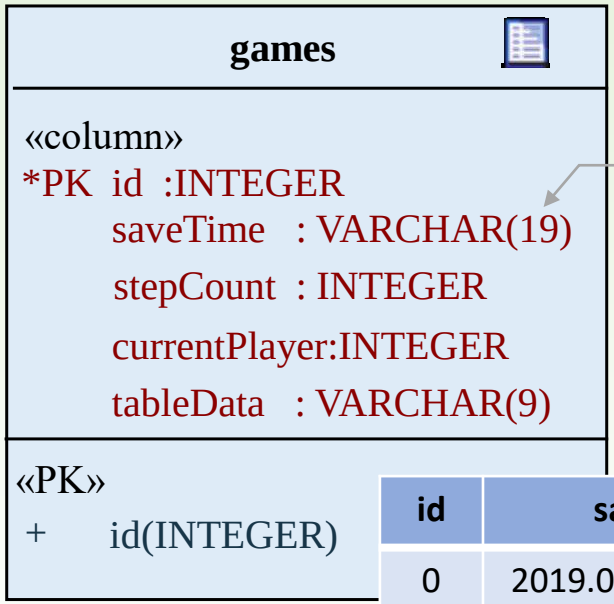
Módosítsuk az előző Tic-Tac-Toe programot úgy, hogy az adatok tárolása a **`game`** adatbázisnak a **`games`** táblájában történjen, ahol a *mezők* az alábbiak: id, saveTime, stepCount, currentPlayer, tableData.

- Továbbra is öt 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 a perzisztencia rétegen kell módosítanunk, a program többi része változatlan marad.

Adatbázisok használata Qt alatt

- A Qt-ben a *QtSql* module támogatja az SQL elérésű adatbázisok platform- és adatbázis független használatát.
- Felhasználjuk modul **`QSqlDatabase`**, és **`QSqlQuery`** osztályait
  - a **`QSqlDatabase`** segítségével megnyitjuk az adatbázist
  - a **`QSqlQuery`** közvetítésével SQL parancsokat használva tudunk olvasni az adatbázisból, illetve írni az adatbázisba

2.Megoldás: adatbázis



id	saveTime	stepCount	currentPlayer	tableData
0	2019.09.22 11:30:23	3	2	101000002
3	2019.09.29 16:38:04	6	1	121020102

2.Megoldás: perzisztencia

```
TicTacToeDataAccess::TicTacToeDataAccess()
{
    QSqlDatabase database = QSqlDatabase::addDatabase("QMYSQL");
    database.setDatabaseName("game");
    database.setHostName("localhost");
    database.setUserName("root");
    database.setPassword("root");

    database.open();
}

TicTacToeDataAccess::~TicTacToeDataAccess()
{
    QSqlDatabase::database().close();
}
```

megnyitja a kapcsolatot

bezárja a kapcsolatot

2.Megoldás: adatelérés

```
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;
}
```

Az aktuális (**game**) adatbázis **games** táblájának minden sorából veszi az **id** és **saveTime** értékeket.

A **query** a lekérdezés aktuális sorának lekért két értékét mutatja.

2.Megoldás: perzisztencia

```
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;
}
```

Az aktuális (**game**) adatbázis **games** táblájának **id** kulcsú sorából veszi a **stepCount**, **currentPlayer** és a **tableData** értékeket.

A **query** a lekérdezett adattábla sor elemeit tartalmazza.

2.Megoldás: perzisztencia

```
bool TicTacToeDataAccess::
     saveGame(int gameIndex, const QVector<int> &saveGameData)
{
     QSqlQuery query;
     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]);
     }
     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 + "'" );
}
```

Az aktuális (**game**) adatbázis **games** táblájának **gameIndex** kulcsú sorát törli, ha van ilyen.

Az aktuális (**game**) adatbázis **games** táblájába beszúr egy **gameIndex** kulcsú sort.

Függőségek

- A több rétegű architektúrában egy réteg felhasználja az alatta lévő réteg funkcionalitását, azaz a saját funkcionalitása függ az alatta lévő rétegtől.
- Ez a *függőség* (*dependency*, *coupling*) számos problémát okozhat (pl. túl sok függőség, hosszú függőségi láncok, ütközések, körkörös függőségek).
- A cél a minél kisebb függőség kiépítése (*loose coupling*) a rétegek között, jól definiált felületek (interfészek) mentén.

```
class Service {public:
    void Provide() { … }
}
class Client {
private:
    Service _service;
public:
    void Run() { … _service.Provide(); … }
}
```

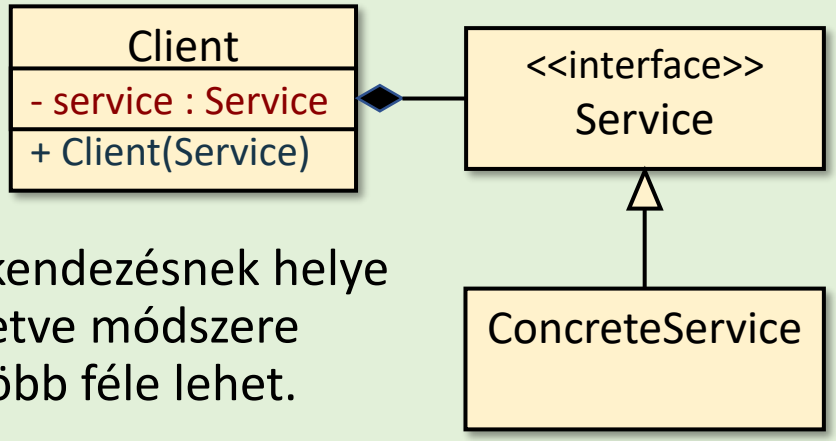
egy osztály, ami biztosít egy szolgáltatást

egy osztály, ami felhasználja a szolgáltatást

így függőség alakul ki

Függőségek megvalósítása

- A függőségeket úgy kell megvalósítanunk, hogy
  - a kliens a felhasznált szolgáltatás konkrét megvalósításától ne, csak annak felületétől (*interfészétől*) függjön (lásd SOLID elvek – *dependency inversion principle*).
  - a kliensnek az interfészt megvalósító objektumot futási időben adjuk át: ez a *függőség befecskendezés* (*dependency injection*)



  - a modulba történő befecskendezésnek helye (konstruktor, metódus), illetve módszere (példányosítás, másolás) több féle lehet.

Példa

```
class ServiceInterface {
public:
    void Provide() = 0;
}
class ConcreteService : public ServiceInterface {
public:
    void Provide() { … }
}
class Client {
private:
    ServiceInterface *_service;
public:
    Client(ServiceInterface *s) { _service = s; }
    void Run() { … _service->Provide(); … }
}
Client client(new ConcreteService());
```

egy osztály, ami biztosít egy szolgáltatást

egy osztály, ami megvalósítja a szolgáltatást

a függőség csak a felületre vonatkozik

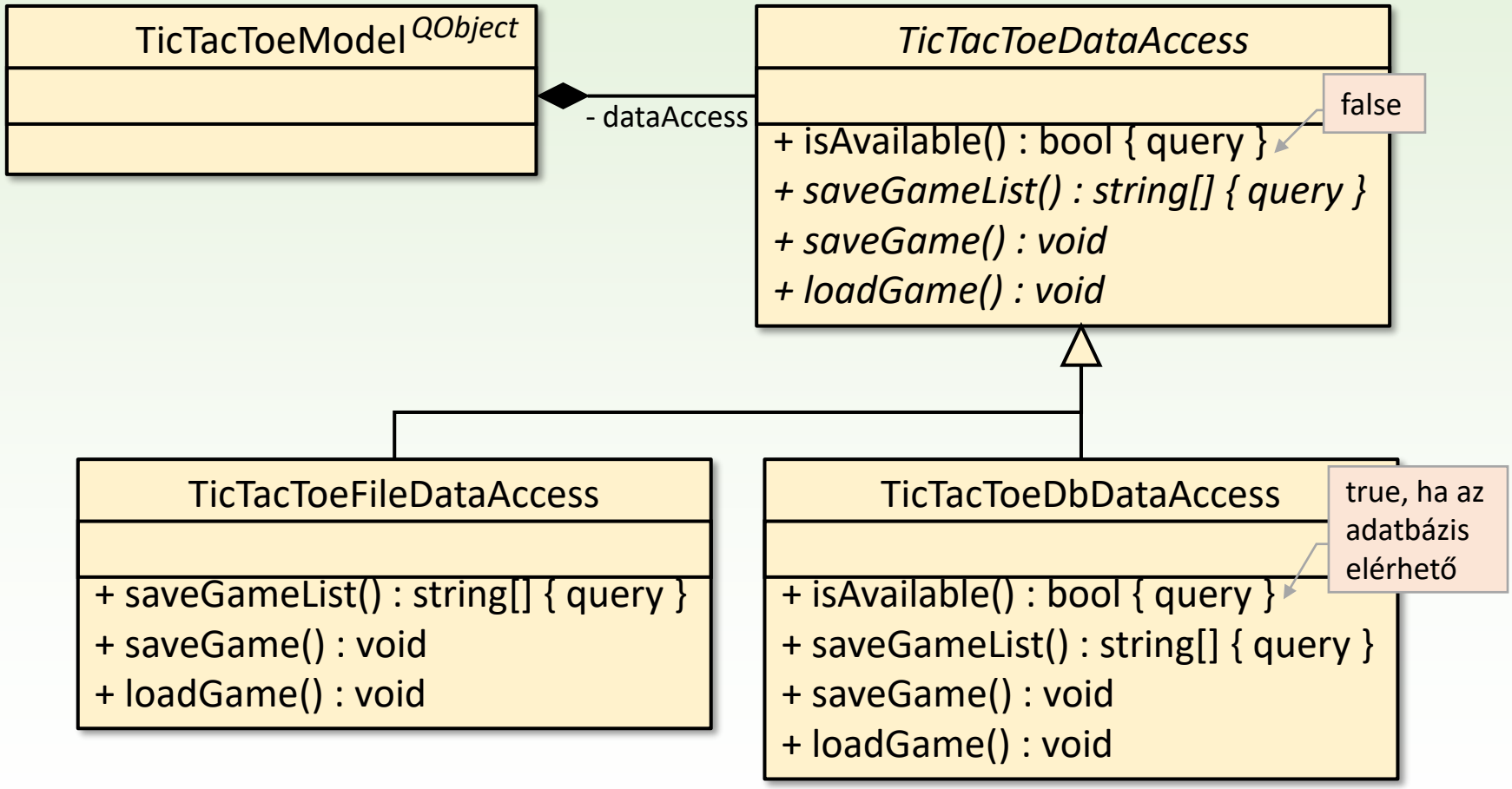
átadjuk a konkrét megvalósítást

3.Megoldás

Módosítsuk az előző Tic-Tac-Toe programot úgy, hogy az adatok tárolása alapértelmezetten az adatbázist használja mentésre, de amennyiben az nem elérhető, használjon fájl alapú adatkezelést.

- Az adatelérés osztályunk absztrakt lesz, és származtatjuk belőle a fájl (**`TicTacToeFileDataAccess`**) és adatbázis (**`TicTacToeDbDataAccess`**) alapú elérést.
- Az osztály kiegészül a rendelkezésre állás lekérdezésével (**`isAvailable`**).
- A nézet fogja megállapítani, hogy milyen adatelérést használjunk, és létrehozza a megfelelő perzisztenciát végző befecskendezendő objektumot.

3.Megoldás: tervezés



3.Megoldás: 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 létrehozzuk az adateléréssel
    …
}
```

Többrétegű alkalmazások tesztelése

- A függőség befecskendezés a fejlesztés során nagy szabadságot ad, mivel elég a felhasznált osztály interfészét megadni az attól függő osztály fejlesztéséhez.
  - Ekkor a függő osztály implementációját nem zavarja a konkrét megvalósítás hiánya.
  - A függő osztály tesztelése viszont csak úgy hajtható végre, ha a konkrét felhasznált osztályt is megvalósították, és ez lassíthatja a fejlesztést.
  - A függő osztály tesztelése során az is problémát jelenthet, ha a felhasznált osztály megvalósítása hibás, mivel ekkor a tőle függő osztály is hibás viselkedést produkál (noha a hiba másik osztályban található).

Mock objektumok

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

Példa

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

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

mock-olást megvalósító osztály, amely egy egyszerű megvalósítást biztosít

ezt használjuk, és így már tesztelhető a Client osztály

Feladat

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

- A modell függ az adateléréstől, de azt nem akarjuk tesztelni, ezért annak 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);
}
```

Tesztelés: 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;
}
```

Tesztelés: 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;
}
```