Eötvös Loránd Tudományegyetem
Informatikai Kar

Eseményvezérelt alkalmazások fejlesztése I

8. előadás

Általános szoftver architektúrák

Giachetta Roberto

http://people.inf.elte.hu/groberto

Általános szoftver architektúrák

Szoftverek architektúrája

- *Szoftver architektúrá*nak nevezzük a szoftver fejlesztése során meghozott *elsődleges tervezési döntések* halmazát
  - az architektúra létrehozása során mintákra hagyatkozunk, a szoftver teljes architektúráját definiáló mintákat nevezzük *architekturális mintá*knak (*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
    - más néven *kétrétegű* (*two-tier*) architektúra, ahol a két réteg egymásra épül, vertikálisan

Általános szoftver architektúrák

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

Általános szoftver architektúrák

Példa

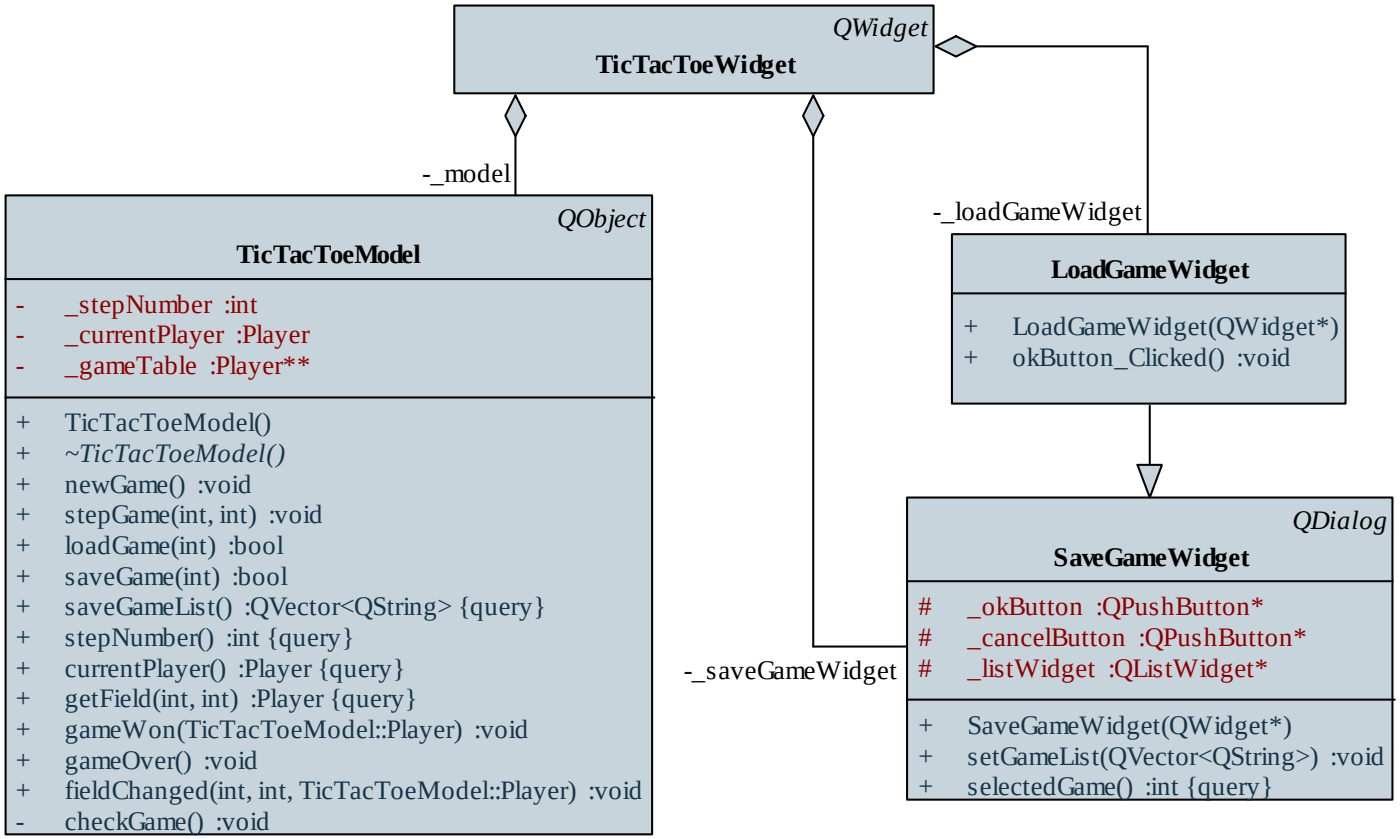
Feladat: Készítsünk egy Tic-Tac-Toe programot, amelyben két játékos küzdhet egymás ellen.

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

Általános szoftver architektúrák

Példa

Tervezés (architektúra):



Általános szoftver architektúrák

Példa

Megvalósítás (`tictactoemodel.cpp`*):*

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

    QTextStream stream(&file);
       // soronként egy adatot írunk ki
    stream << _stepNumber << endl;
    stream << (int)_currentPlayer << endl;
    …
    return true;
}
```

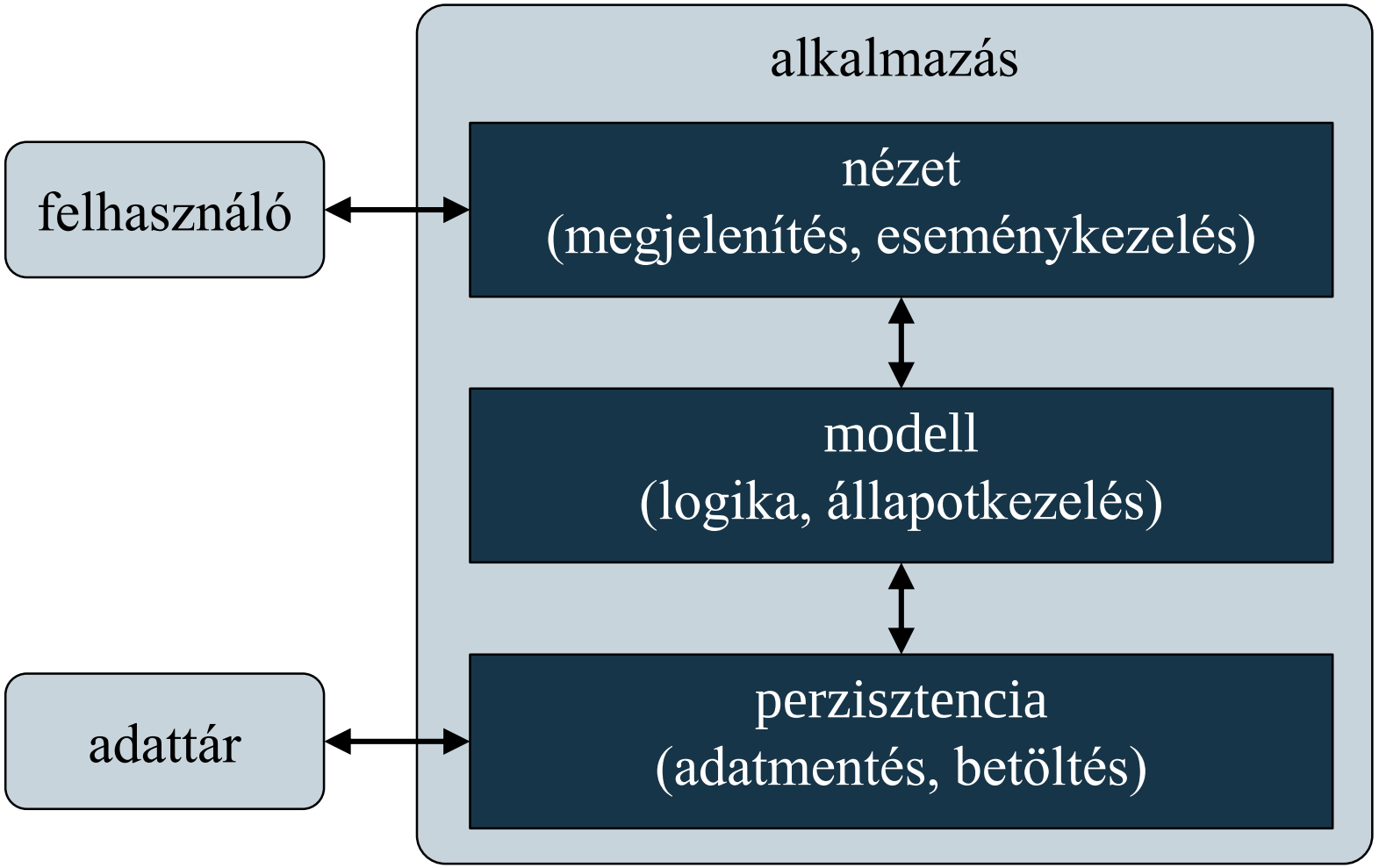
Általános szoftver architektúrák

A háromrétegű architektúra

- Igazából 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 responsibilty 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*)

Általános szoftver architektúrák

A háromrétegű architektúra



Általános szoftver architektúrák

Példa

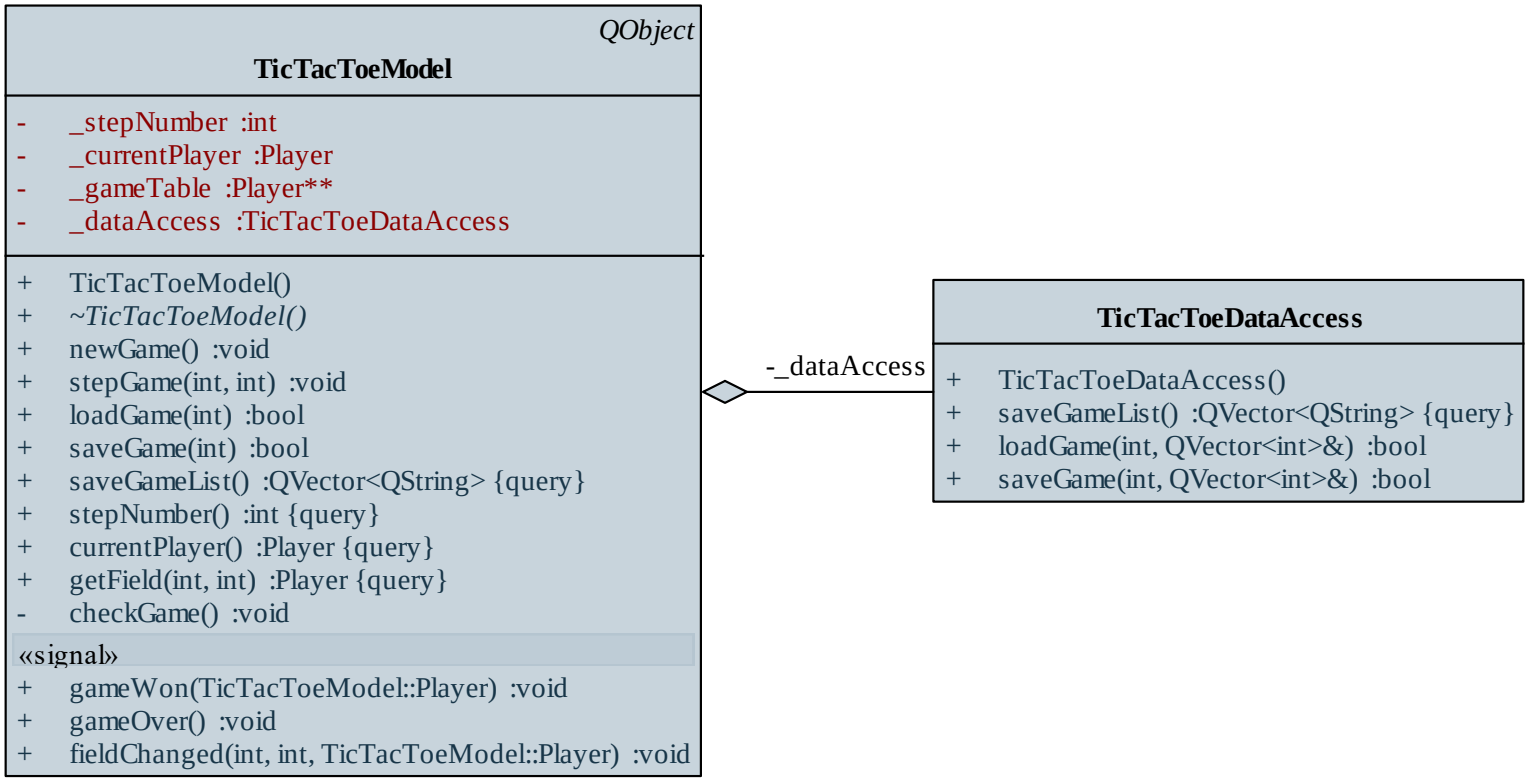
Feladat: Készítsünk egy Tic-Tac-Toe programot háromrétegű architektúrában.

- leválasztjuk az adatelérést a modellről egy új osztályba (`TicTacToeDataAccess`), amely biztosítja a három adatkezelési műveletet (`saveGame`, `loadGame`, `saveGameList`)
- az adatok modell és adatelérés közötti egyszerű kommunikációja érdekében az adatelérési réteg egészek vektorát fogja kezelni, amely 11 értéket tárol a korábbi sorrendnek megfelelően (lépésszám, játékos, mezők sorfolytonos sorrendben)

Általános szoftver architektúrák

Példa

Tervezés:



Általános szoftver architektúrák

Példa

Megvalósítás (`tictactoemodel.cpp`*):*

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

    // összerakjuk a megfelelő tartalmat
    saveGameData.push_back(_stepNumber);
    saveGameData.push_back((int)_currentPlayer);
    ...

    return _dataAccess.saveGame(gameIndex,
                                saveGameData);
       // az adatelérés végzi a tevékenységeket
}
```

Általános szoftver architektúrák

Példa

Feladat: Készítsünk egy Tic-Tac-Toe programot háromrétegű architektúrában.

- módosítsuk úgy az adatkezelést, hogy az adatok tárolása adatbázisban történjen, a **`game`** adatbázis **`games`** táblájában
- 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`**)

Általános szoftver architektúrák

Példa

Megvalósítás (`tictactoedataaccess.cpp`*):*

```
bool TicTacToeDataAccess::loadGame(int gameIndex,
                                   QVector<int> &saveGameData)
{
    QSqlQuery query;
    query.exec("select stepCount, currentPlayer,
                tableData from games where id = " +
                QString::number(gameIndex));
    …
    // betöltjük a mentés egyes elemeit
    saveGameData[0] = query.value(0).toInt();
    saveGameData[1] = query.value(1).toInt();
    …
}
```

Általános szoftver architektúrák

Függőségek

- Egy több rétegű architektúrában a rétegek (modulok) felhasználják az alattuk lévő réteg funkcionalitását, azaz a saját funkcionalitásuk függ az alattuk lévő rétegtől
- A *függőség*nek (*dependency*, *coupling*) több formája és szintje lehet
  - általában a cél a minél kisebb függőség elérése (*loose coupling*) a rétegek között, jól definiált felületek (interfészek) mentén
  - több réteg esetén a függőségek láncot alkotnak
  - függőség miatt számos probléma felmerülhet (pl. túl sok függőség, hosszú láncok, ütközések, körkörös függőségek)

Általános szoftver architektúrák

Függőségek

- Pl.:

```
class Service {
   // egy osztály, ami biztosít egy szolgáltatást
public:
   void Provide() { … }
}

class Client { // egy osztály, amely felhasználja
               // a szolgáltatást (kliens)
private:
   Service _service; // így függőség alakul ki
public:
   void Run() { … _service.Provide(); … }
}
```

Általános szoftver architektúrák

Függőségek kezelése

- A függőségeket úgy kell megvalósítanunk, hogy
  - a felhasznált osztály konkrét megvalósításától ne, csak felületétől (*interfész*étől) függjön (*dependency inversion principle*)
  - a megvalósítás a körülmények függvényében könnyen változtatható legyen
  - pl. az adatkezelést csak akkor végezhetjük adatbázisban, amennyiben az rendelkezésünkre áll
- Ennek megfelelően a függőségeket mindig általános formában (interfész, vagy absztrakt osztály) kell kezelnünk
  - ez érvényes minden modulra az architektúrában

Általános szoftver architektúrák

Függőségek kezelése

- Pl.:

```
class ServiceInterface {
   // egy osztály, ami biztosít egy szolgáltatás
   // felületet
public:
   void Provide() = 0;
}

class ConcreteService : public ServiceInterface {
   // egy osztály, ami megvalósítja a
   // szolgáltatást
public:
   void Provide() { … }
}
```

Általános szoftver architektúrák

Függőségek kezelése

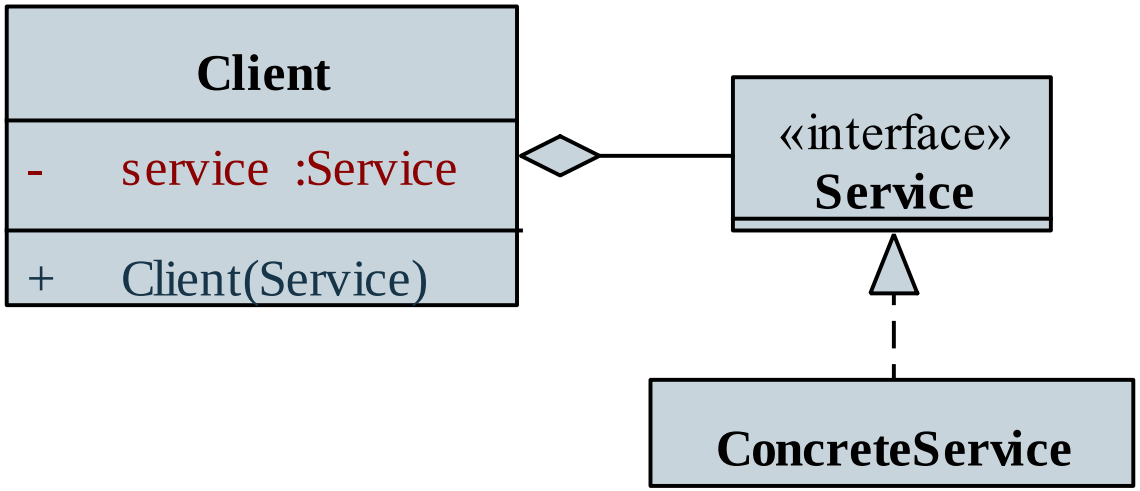
```
class Client {
private:
   ServiceInterface *_service;
   // a függőség csak a felületre vonatkozik
public:
   MyClass(ServiceInterface *s) { _service = s; }
      // valahol megadjuk, mi lesz a felhasznált
      // szolgáltatás
   void Run() { … _service->DoSomething(); … }
      // a megvalósítás fog végrehajtódni
}

Client client(new ConcreteService());
   // átadjuk a konkrét megvalósítást
```

Általános szoftver architektúrák

Függőségek kezelése

- A modulok tehát a függőségeknek csak az absztrakcióját látják, a konkrét megvalósítást külön adjuk át nekik, ezt nevezzük *függőség befecskendezés*nek (*dependency injection*)
  - a befecskendezés helye/módszere függvényében lehetnek különböző típusai (pl. konstruktor, metódus, interfész)



Általános szoftver architektúrák

Példa

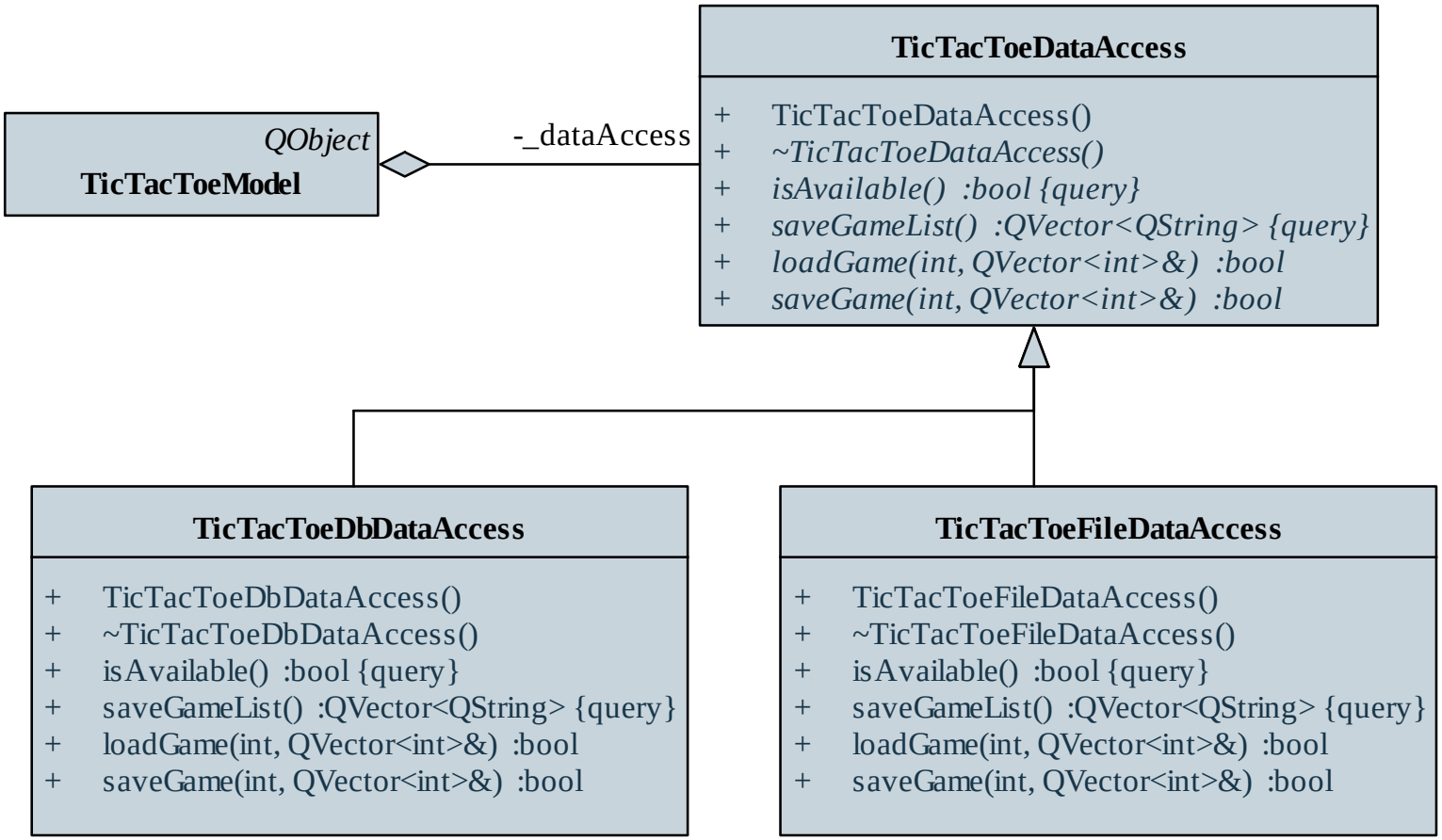
Feladat: Készítsünk egy Tic-Tac-Toe programot háromrétegű architektúrában.

- 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`)

Általános szoftver architektúrák

Példa

Tervezés:



Általános szoftver architektúrák

Példa

Megvalósítás (`tictactoewidget.cpp`*):*

```
TicTacToeWidget::TicTacToeWidget(QWidget *parent)
    : QWidget(parent) {
    …
    // az adatkezelést itt döntjük el
    _dataAccess = new TicTacToeDbDataAccess();
       // alapértelmezetten adatbázist használunk
    if (!_dataAccess->isAvailable()){
       // de ha az nem elérhető
       _dataAccess = new TicTacToeFileDataAccess();
       // átváltunk fájlra
    }
    …
}
```

Általános szoftver architektúrák

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
  - továbbá 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)

Általános szoftver architektúrák

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 objektum*oknak

Általános szoftver architektúrák

Mock objektumok

- Pl.:

```
class ServiceMock : public ServiceInterface {
   // mock-olást megvalósító osztály
public:
   void Provide() {
      qDebug() << "Running service.";
   } // amely egy egyszerű implementációt biztosít
}

Client client(new ServiceMock());
   // ezt felhasználjuk, így már tesztelhetjük az
   // osztályunkat
```

Általános szoftver architektúrák

Példa

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
- létrehozunk egy mock objektumot az adatelérésre (**`TicTacToeDataAccessMock`**), amely egyszerű funkciókat biztosít, és a konzolra (**`qDebug`**) üzen, ennek egy példányát felhasználjuk a tesztben

Általános szoftver architektúrák

Példa

Megvalósítás (`tictactoedataaccessmock.h`*):*

```
class TicTacToeDataAccessMock :
   public TicTacToeDataAccess
   // mock objektum, csak teszteléshez
{
   …
   bool saveGame(…) { // játék mentése
      qDebug() << "game saved to slot ("
         << gameIndex << ") with values: ";
      for (int i = 0; i < 11; i++)
         qDebug() << saveGameData[i] << " ";
      …
   }
}
```