

## Eseményvezérelt alkalmazások fejlesztése I

### 8. előadás

## Általános szoftver architektúrák

Giachetta Roberto

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

## 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
    - 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

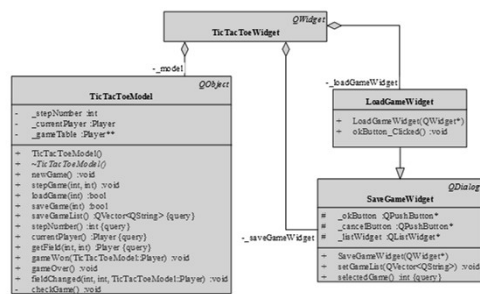
*Feladat:* Készítsünk egy Tic-Tac-Toe programot, amelyben két játékos küzdhet egymás ellen.

- lehetőséget adunk játékállás elmentésére (**Ctrl+L**) és betöltésre (**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 (tictactoeomodel.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 komponens 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)*

ELTE IK, Eseményvezérelt alkalmazások fejlesztése I 8:7

**Általános szoftver architektúrák**  
**A háromrétegű architektúra**

ELTE IK, Eseményvezérelt alkalmazások fejlesztése I 8:8

**Általános szoftver architektúrák**  
**Példa**

*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 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)

ELTE IK, Eseményvezérelt alkalmazások fejlesztése I 8:9

**Általános szoftver architektúrák**  
**Példa**

*Tervezés:*

ELTE IK, Eseményvezérelt alkalmazások fejlesztése I 8:10

**Általános szoftver architektúrák**  
**Példa**

*Megvalósítás (tictactoeomodel.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
}

```

ELTE IK, Eseményvezérelt alkalmazások fejlesztése I 8:11

**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 modul (QSqlDatabase, QSqlQuery)

ELTE IK, Eseményvezérelt alkalmazások fejlesztése I 8:12

Általános szoftver architektúrák	
Példa	
<p>Megvalósítás (tictactoeaccess.cpp):</p> <pre>bool TicTacToeDataAccess::loadGame(int gameIndex,     QVector&lt;int&gt; &amp;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();     ... }</pre>	
ELTE IK, Eseményvezérelt alkalmazások fejlesztése I	8:13

Általános szoftver architektúrák	
Függőségek	
<ul style="list-style-type: none"> <li>Egy több rétegű architektúrában a rétegek (modulok) felhasználják az alattuk lévő réteg funkcionalitását, azaz a saját funkcionalitásuk függ az alattuk lévő rétegtől</li> <li>A <i>függőségnek</i> (<i>dependency, coupling</i>) több formája és szintje lehet             <ul style="list-style-type: none"> <li>általában a cél a minél kisebb függőség elérése (<i>loose coupling</i>) a rétegek között, jól definiált felületek (interfészek) mentén</li> <li>több réteg esetén a függőségek láncot alkotnak</li> <li>függőség miatt számos probléma felmerülhet (pl. túl sok függőség, hosszú láncok, ütközések, körkörös függőségek)</li> </ul> </li> </ul>	
ELTE IK, Eseményvezérelt alkalmazások fejlesztése I	8:14

Általános szoftver architektúrák	
Függőségek	
<ul style="list-style-type: none"> <li>Pl.:             <pre>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(); ... } }</pre> </li> </ul>	
ELTE IK, Eseményvezérelt alkalmazások fejlesztése I	8:15

Általános szoftver architektúrák	
Függőségek kezelése	
<ul style="list-style-type: none"> <li>A függőségeket úgy kell megvalósítanunk, hogy             <ul style="list-style-type: none"> <li>a felhasznált osztály konkrét megvalósításától ne, csak felületétől (<i>interfésztől</i>) függjön (<i>dependency inversion principle</i>)</li> <li>a megvalósítás a körülmények függvényében könnyen változtatható legyen</li> <li>pl. az adatkezelést csak akkor végezhetjük adatbázisban, amennyiben az rendelkezésünkre áll</li> </ul> </li> <li>Ennek megfelelően a függőségeket mindig általános formában (interfész, vagy absztrakt osztály) kell kezelnünk             <ul style="list-style-type: none"> <li>ez érvényes minden modulra az architektúrában</li> </ul> </li> </ul>	
ELTE IK, Eseményvezérelt alkalmazások fejlesztése I	8:16

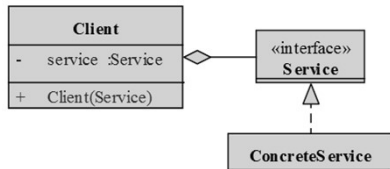
Általános szoftver architektúrák	
Függőségek kezelése	
<ul style="list-style-type: none"> <li>Pl.:             <pre>class ServiceInterface {     // egy osztály, ami biztosít egy szolgáltatás     // felületet public:     void Provide() = 0; }  class ConcreteService : public ServiceInterface {     // egy osztály, ami megvalósítja a     // szolgáltatást public:     void Provide() { ... } }</pre> </li> </ul>	
ELTE IK, Eseményvezérelt alkalmazások fejlesztése I	8:17

Általános szoftver architektúrák	
Függőségek kezelése	
<pre>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-&gt;DoSomething(); ... }     // a megvalósítás fog végrehajtódni }  Client client(new ConcreteService()); // átadjuk a konkrét megvalósítást</pre>	
ELTE IK, Eseményvezérelt alkalmazások fejlesztése I	8:18

## Általános szoftver architektúrák

### Függőségek kezelése

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



ELTE IK, Eseményvezérelt alkalmazások fejlesztése I

8:19

## 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`)

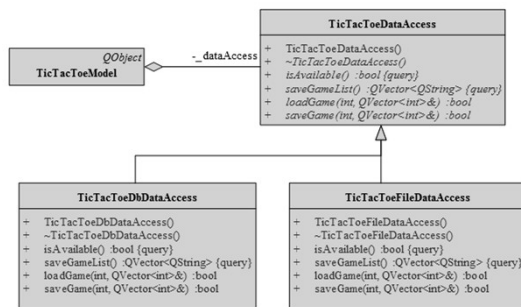
ELTE IK, Eseményvezérelt alkalmazások fejlesztése I

8:20

## Általános szoftver architektúrák

### Példa

*Tervezés:*



ELTE IK, Eseményvezérelt alkalmazások fejlesztése I

8:21

## 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
}
...
}
```

ELTE IK, Eseményvezérelt alkalmazások fejlesztése I

8:22

## 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 egységtesztnek 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)

ELTE IK, Eseményvezérelt alkalmazások fejlesztése I

8:23

## 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 objektumoknak*

ELTE IK, Eseményvezérelt alkalmazások fejlesztése I

8:24

## 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
```

ELTE IK, Eseményvezérelt alkalmazások fejlesztése I

8:25

## 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

ELTE IK, Eseményvezérelt alkalmazások fejlesztése I

8:26

## Általános szoftver architektúrák

### Példa

*Megvalósítás (tictactoeataccessmock.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] << " ";
        ...
    }
}
```

ELTE IK, Eseményvezérelt alkalmazások fejlesztése I

8:27