



Eötvös Loránd Tudományegyetem  
Informatikai Kar

# **Eseményvezérelt alkalmazások fejlesztése I**

---

## **8. előadás**

### **Adatbázis-kezelés modell/nézet architektúrában**

---

**Giachetta Roberto**

A jegyzet az ELTE Informatikai Karának  
2014. évi Jegyzetpályázatának támogatásával készült

# Adatbázis-kezelés modell/nézet architektúrában

## A modell/nézet architektúra

- Az összetettebb alkalmazásoknál célszerű a kétrétegű architektúrát bevezetni, amelyet a *modell/nézet* (*Model/View*) architektúrának nevezünk



- Adatkezelő alkalmazásoknál ez több szempontból is hasznos lehet, pl.:
  - az adatkezelést könnyen átalakíthatóvá teszi
  - az adatmegjelenítést egyedire szabhatjuk

# Adatbázis-kezelés modell/nézet architektúrában

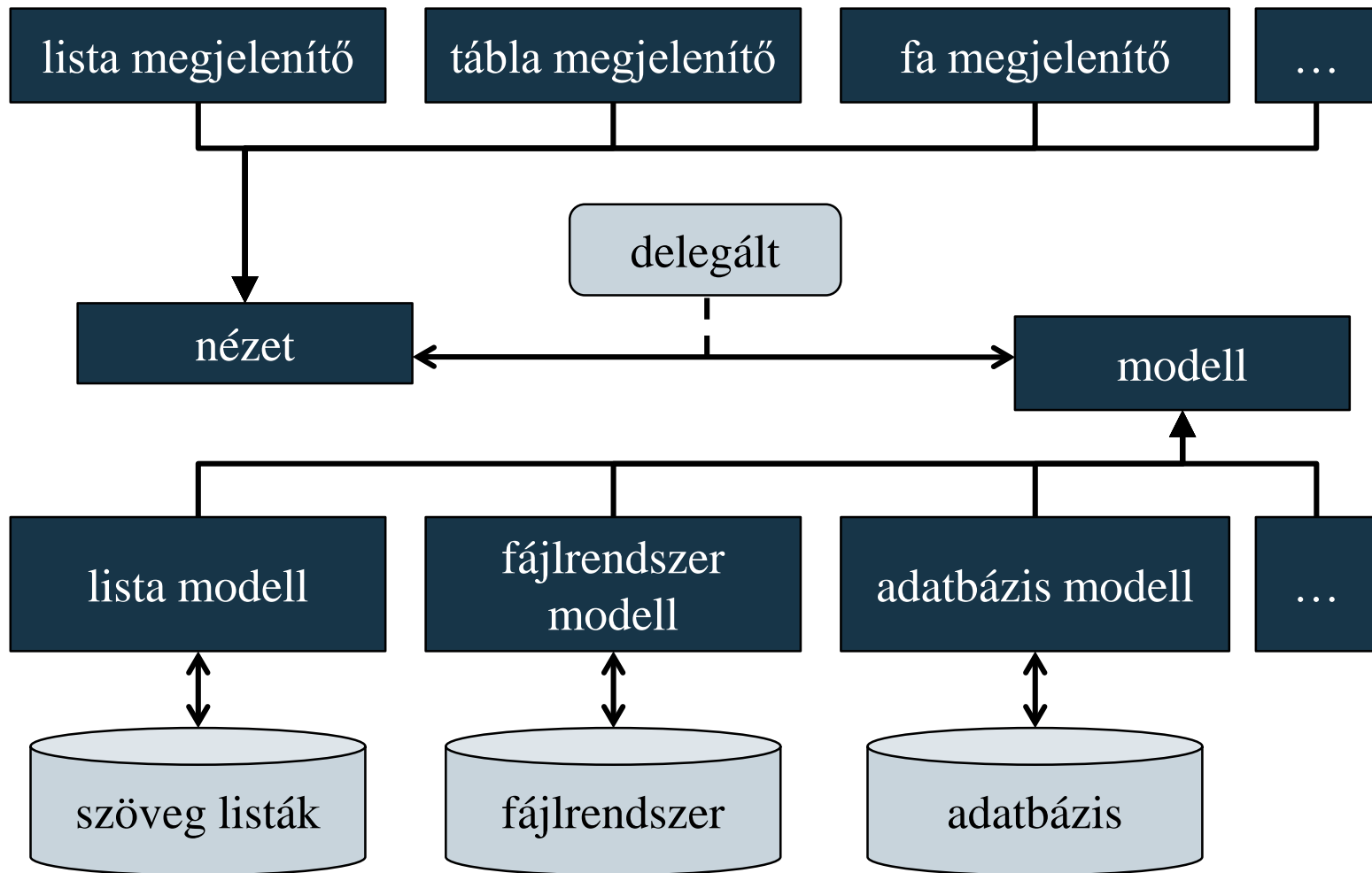
## A Qt koncepciója

---

- A Qt magába ágyazta a modell/nézet architektúrát adatkezelésre, így beépített elemek használatával is elérhető az alkalmazás rétegelt felépítése
  - a *modell* biztosít osztályokat a különböző adatforrások olvasására, írására, pl. listák, XML fájlok, adatbázisok, fájlrendszer, ...
  - a *nézet* különböző megjelenítő grafikus vezérlőket tartalmaz, pl. táblázat megjelenítő, lista megjelenítő, ...
  - a két réteg pontos összeillesztését a *delegált* (*item delegate*) típusok felügyelik, amelyek meghatározzák, milyen módon jelenjenek meg az adatok

# Adatbázis-kezelés modell/nézet architektúrában

## A Qt koncepciója



# Adatbázis-kezelés modell/nézet architektúrában

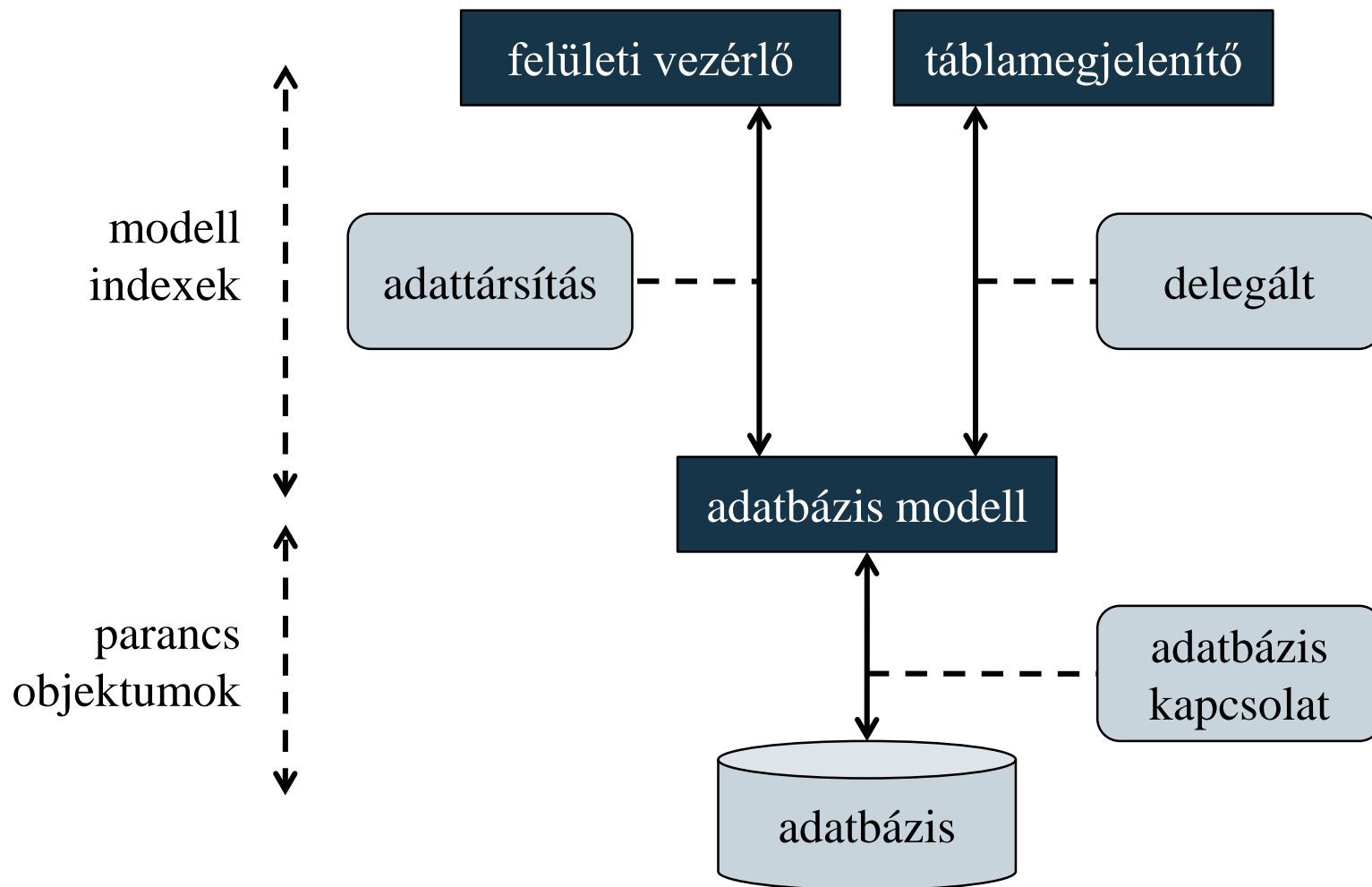
## Adattársítás és indexelés

---

- A felületen a nézetek mellett egyéb grafikus vezérlők is kezelhetik az adatokat, amennyiben megfelelő *adattársítást* (*data binding*) biztosítunk a vezérlő és a modell között
- Egy modell tetszőlegesen sok különböző felületi elemhez kapcsolható, különböző nézetek és adattársítások használatával
  - a tartalom szinkronizált, így az egyik nézetben végzett módosítások azonnal megjelennek a másik nézetben
- Az adatközlés a felület és a modell között *modell indexek* (**QModelIndex**) segítségével történik, amely az adatok lokalizálására szolgál
  - pl. sor/oszlop szám alapján táblázat esetén

# Adatbázis-kezelés modell/nézet architektúrában

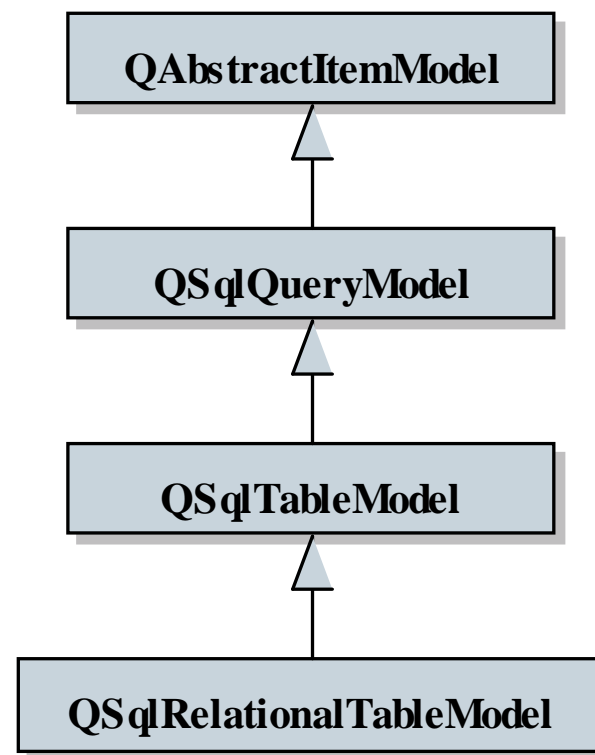
## Adattársítás és indexelés



# Adatbázis-kezelés modell/nézet architektúrában

## Adatbázis-kezelő modellek

- A modellek a `QAbstractItemModel` leszármazottai, ezek közül adatbázis-kezelésre 3 alkalmazható:
  - `QSqlQueryModel`: egy lekérdezés eredményének kezelésére (csak olvasható)
  - `QSqlTableModel`: egy tábla tartalmának kezelésére (írható és olvasható)
  - `QSqlRelationalTableModel`: idegen kulcsokat tartalmazó tábla kezelésére (további táblákból begyűjtött adatokkal)



# Adatbázis-kezelés modell/nézet architektúrában

## Lekérdezés modellek

---

- A `QSqlQueryModel` típust lekérdezések megjelenítésére, olvasásra használhatjuk
  - a `setQuery(<lekérdezés>)` metódussal beállíthatunk tetszőleges lekérdezést (akár több táblát felhasználva)
  - a `setHeaderData(<oszlop>, <megjelenés>, <elnevezés>)` művelettel szabályozhatjuk az oszlopok tulajdonságait
  - a sorok számát a `rowCount()`, az oszlopok számát a `columnCount()` metódussal kérdezhetjük le
  - az adatokat soronként (`record(<sor>)`), vagy indexek segítségével (`index(<sor>, <oszlop>)`) érhetjük el

# Adatbázis-kezelés modell/nézet architektúrában

## Lekérdezés modellek megjelenítése

- Modellek megjelenítéséhez a `QAbstractItemView` leszármazottait kell használnunk, táblázatos megjelenítéshez a `QTableView` típust
  - a `setModel(<modell>)` művelettel állítjuk be a modellt
  - pl.:

```
QSqlQueryModel model; // modell
model.setQuery("select ..."); // lekérdezés
model.setHeaderData(0, Qt::Horizontal, "Id");
    // oszlop fejlécének beállítása

...
QTableView view; // nézet
view.setModel(model); // modell beállítása
view.show(); // megjelenítés
```

# Adatbázis-kezelés modell/nézet architektúrában

## Példa

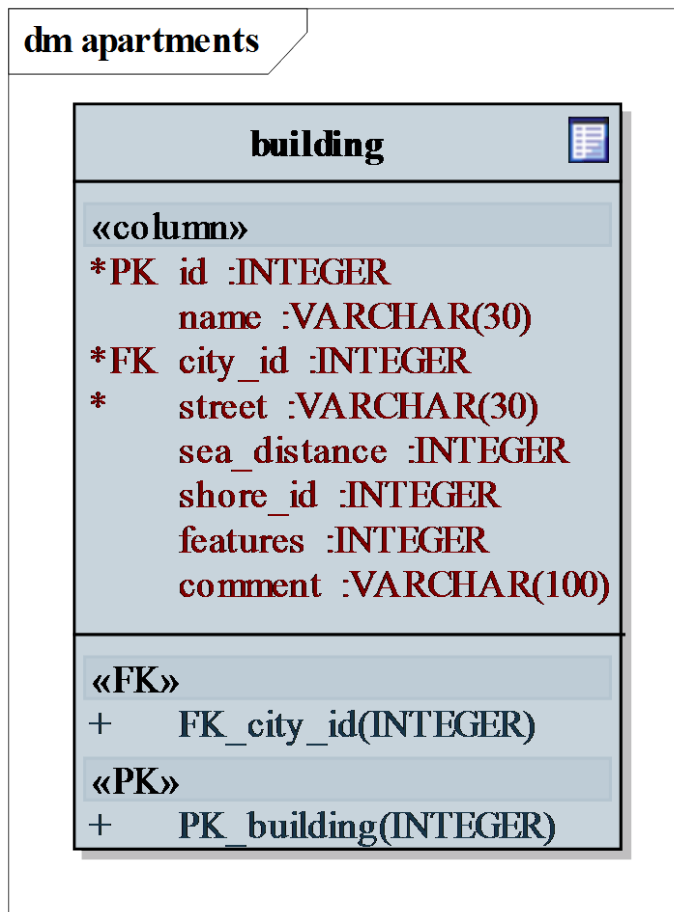
*Feladat:* Készítsük el az apartman adatbázis épületeinek (**buildings**) grafikus megjelenítését.

- az alkalmazáshoz nem kell egyetlen új osztályt se definiálnunk, a létező típusok felhasználásával megoldható a feladat
- az ablakban egy táblamegjelenítőben jelenjen meg a tábla teljes tartalma, ehhez egy **QTableView** példányt alkalmazunk
- az adatok betöltését egy lekérdező modellel végezzük (**QSqlQueryModel**), amely megkapja a megfelelő lekérdezést, és lefuttatja a lekérdező műveleteket (**QSqlQuery**)

# Adatbázis-kezelés modell/nézet architektúrában

## Példa

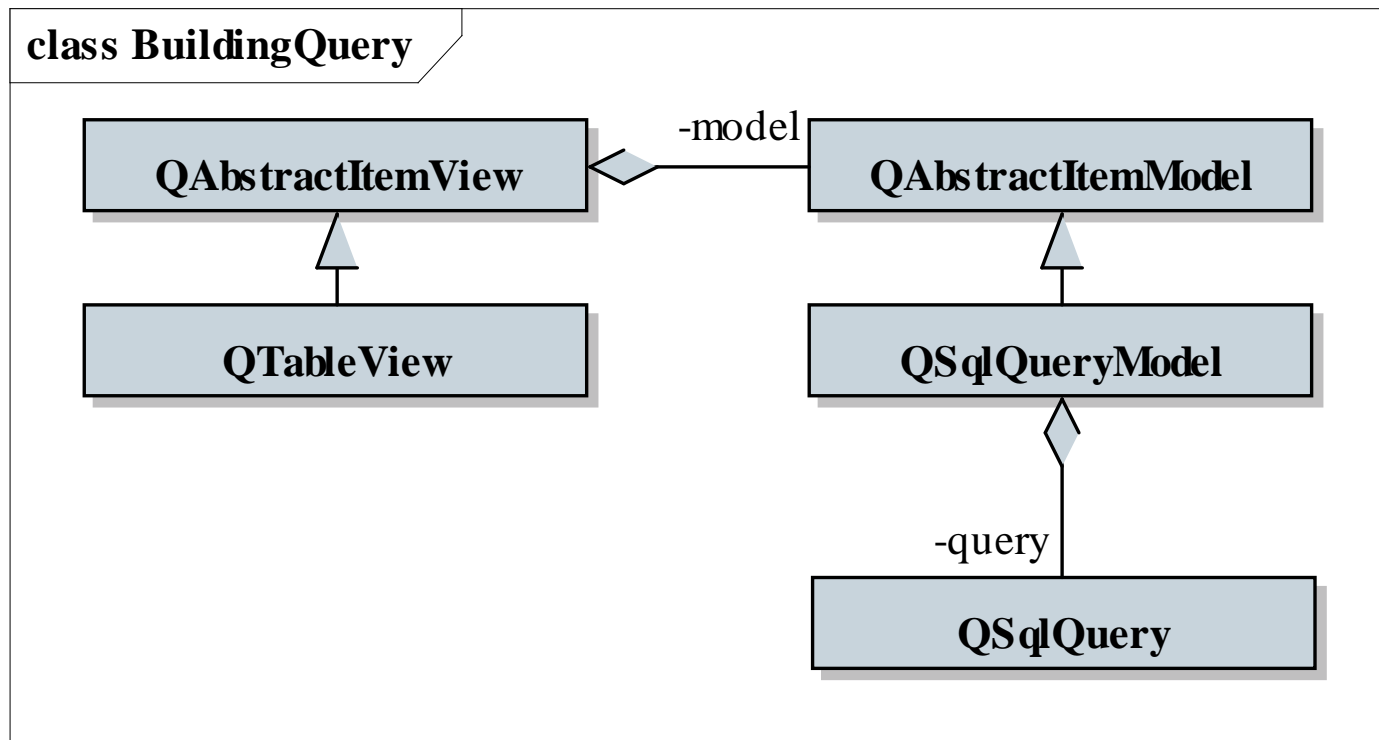
*Tervezés (adatbázis):*



# Adatbázis-kezelés modell/nézet architektúrában

## Példa

*Tervezés (alkalmazás):*



# Adatbázis-kezelés modell/nézet architektúrában

## Példa

*Megvalósítás (main.cpp):*

```
int main(int argc, char *argv[])
{
    QApplication a(argc, argv);

    QSqlDatabase db =
        QSqlDatabase::addDatabase("MYSQL");
    // adatbázis-kapcsolat létrehozása
    ...
    QSqlQueryModel* model = new QSqlQueryModel();
    // lekérdezési modell
    model->setQuery("select * from building");
    // lekérdezés beállítása
```

# Adatbázis-kezelés modell/nézet architektúrában

## Példa

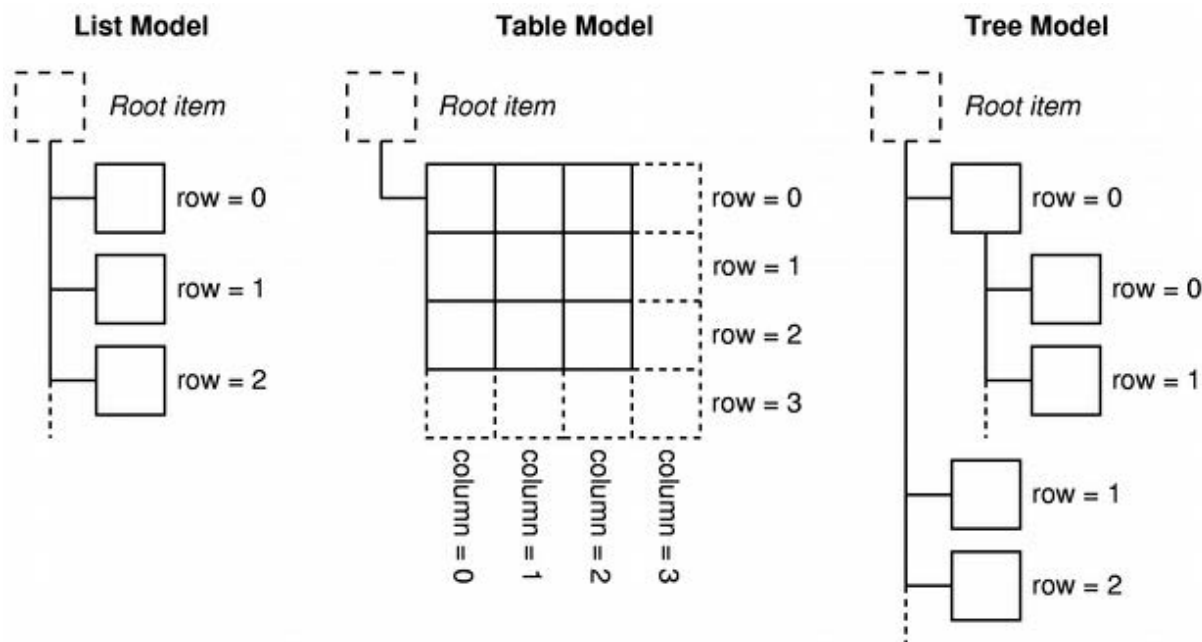
*Megvalósítás (main.cpp):*

```
model->setHeaderData(0, Qt::Horizontal,  
                    trUtf8("Azonosító"));  
    // fejlécek beállítása  
...  
QTableView* tableView = new QTableView();  
    // táblamegjelenítő  
tableView->setModel(model);  
    // modell beállítása a megjelenítőhöz  
tableView->show();  
  
return a.exec();  
}
```

# Adatbázis-kezelés modell/nézet architektúrában

## Indexek használata

- A modellen belüli adatok kezelését indexek (**QModelIndex**) segítségével tehetjük meg
  - minden elemnek a modellünkben saját indexe van, saját címmel, amely a modell felépítésétől függ



# Adatbázis-kezelés modell/nézet architektúrában

## Indexek használata

- az indexhez tartozó adatot a `data()` metódussal kérhetjük le
- az index sorral (`row()`) és oszloppal (`column()`) rendelkezik, fák esetén az indexnek lehetnek gyerek (`child(<sorszám>)`), illetve szülő (`parent()`) indexei is
- Az indexeket a nézetben is használhatjuk
  - a kiválasztás módját a `setSelectionBehavior(<viselkedés>)` és `setSelectionMode(<mód>)` műveletekkel szabályozzuk
  - a `setCurrentIndex(<index>)` a kijelölést állítja
  - az `edit(<index>)` művelettel szerkeszthetővé tehetünk egy elemet, az `update(<index>)` frissíti az adott tartalmat

# Adatbázis-kezelés modell/nézet architektúrában

## Szerkesztő modellek

---

- Egy tábla lekérdezését és szerkesztését a `QSqlTableModel` osztály biztosítja
  - a `setTable(<táblanév>)` művelettel állíthatunk be egy táblát adatforrásnak, a `select()` művelet szolgál az adatok lekérdezésére
  - adatot lekérdezni a `data(<index>)`, beállítani a `setData(<index>, <adat>)` metódussal tudunk
  - lehetőségünk van tetszőlegesen rendezni az adatokat a `setSort(<oszlop>, <rendezési mód>)` művelettel
  - az `insertRow(<sor>)` beszúr egy üres sort a megadott helyre, a `removeRow(<sor>)` töröl egy sort

# Adatbázis-kezelés modell/nézet architektúrában

## Szerkesztő modellek

- Pl.:

```
QSqlTableModel *model; // modell
QTableView *view; // nézet
...
model->setTable("myTable"); // tábla beállítása
model->setSort(0, Qt::AscendingOrder); // rendezés
model->select(); // adatok betöltése
...
int row = view->currentIndex().row();
    // kijelölt sor lekérdezése
model->insertRow(row); // új sor beszúrása
QModelIndex index = model->index(row, 0);
    // index lekérdezése (a sor első oszlopához)
model->setData(index, 100); // adat beállítása
```

# Adatbázis-kezelés modell/nézet architektúrában

## Példa

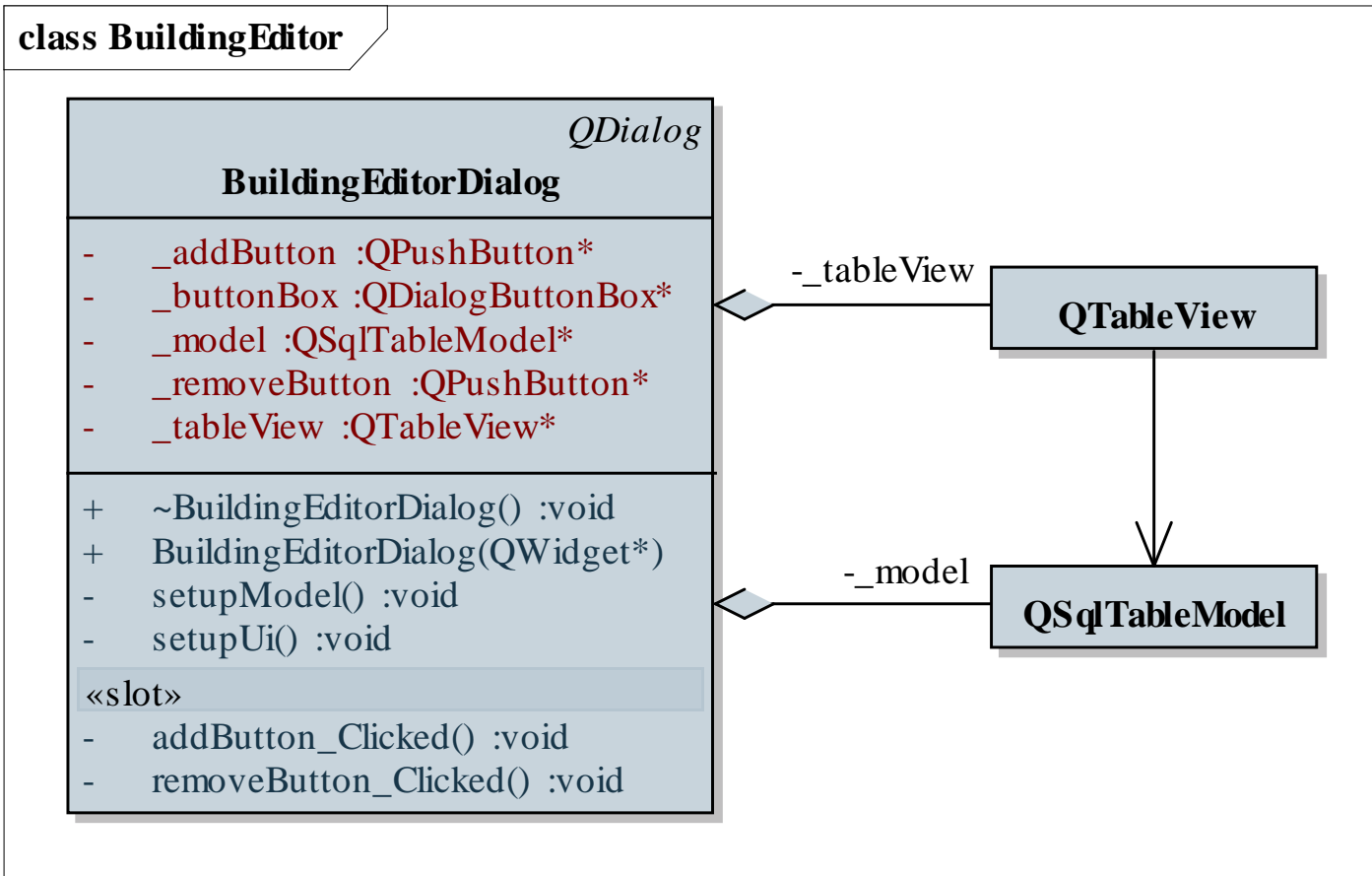
*Feladat:* Készítsünk alkalmazást, amely alkalmas az épületek szerkesztésére, új épület létrehozására, törlésére.

- a táblaszerkesztést egy ablakba (`TableModelDialog`) helyezzük, amelyhez felvesszük a hozzáadás és törlés gombjait, a táblakezeléshez egy `QSqlTableModel`, a megjelenítéshez egy `QTableView` példányt használunk
- beszúráskor lekérdezzük a kijelölt sor indexét, behelyezünk egy sort a helyére, átállítjuk a kijelölést (az indexen keresztül), majd szerkesztésre váltunk
- törléskor töröljük a kijelölt sort (amennyiben van kijelölés), és áthelyezzük a kijelölést

# Adatbázis-kezelés modell/nézet architektúrában

## Példa

*Tervezés:*



# Adatbázis-kezelés modell/nézet architektúrában

## Példa

*Megvalósítás (main.cpp):*

```
int main(int argc, char *argv[])
{
    QApplication a(argc, argv);
    QSqlDatabase db =
        QSqlDatabase::addDatabase("QMYSQL");
    ...
    if (db.open()) { // kapcsolat megnyitása
        BuildingEditorDialog *w =
            new BuildingEditorDialog();
        w->show();
        // ablak megnyitása
    }
    ...
}
```

# Adatbázis-kezelés modell/nézet architektúrában

## Példa

*Megvalósítás* (buildingeditordialog.cpp):

```
void BuildingEditorDialog::setupModel() {
    _model = new QSqlTableModel(this);
    // táblamodell létrehozása
    _model->setTable("building");
    // tábla beállítása
    model->setSort(1, Qt::AscendingOrder);
    // rendezési sorrend
    ...
    model->setHeaderData(0, Qt::Horizontal,
                        trUtf8("Azonosító"));
    ...
    model->select(); // adatok begyűjtése
    ...
}
```

# Adatbázis-kezelés modell/nézet architektúrában

## Példa

*Megvalósítás* (buildingeditordialog.cpp):

```
void BuildingEditorDialog::setupUi()  
{  
    ...  
    tableView = new QTableView(this);  
    tableView->setModel(model);  
    // modell hozzákapcsolása a megjelenítőhöz  
    tableView->setSelectionBehavior(  
        QAbstractItemView::SelectItems);  
    // kijelölés módja  
    tableView->resizeColumnsToContents();  
    // oszlopok automatikus méretezése  
    ...  
}
```

# Adatbázis-kezelés modell/nézet architektúrában

## Szerkesztési stratégia

---

- A szerkesztési stratégia függvényében kétféle megközelítést alkalmazhatunk az adatkezelésben:
  - *szinkron (állandó kapcsolatú) modell*: az adatbázis és a modell tartalma folyamatosan (legalábbis rekordváltásonként) egyezik, ez az automatikus szerkesztési stratégia
  - *aszinkron (bontott kapcsolatú) modell*: az adatbázis és a modell tartalma különbözhet, és csak meghatározott pontokon egyezik meg (**select**, **submitAll**, **revertAll**), ez a manuális szerkesztési stratégia
- A gyakorlatban az aszinkron modell az elterjedtebb, mivel nem igényel állandóan az adatbázis műveletek futtatását

# Adatbázis-kezelés modell/nézet architektúrában

## Szerkesztési stratégia

---

- A szerkeszthető modell lehetőséget ad a szerkesztési stratégia beállítására
  - a modell a módosításokat első lépésben csak a memóriában végzi el, utána menti vissza azokat az adatbázisba
  - a módosítás fennállását az `isDirty(<index>)` metódussal kérhetünk le, ez igazat ad, amennyiben az adat eltér az adatbázisban tárolttól
  - egy sort, vagy adatot menteni a `submit()`, a teljes tartalmat menteni a `submitAll()` utasítással tudunk
  - lehetőségünk van változtatások visszavonására is `revert()` és `revertAll()` metódusokkal

- a `setEditStrategy(<stratégia>)` függvényével definiálhatjuk a visszamentés módját, ez a következő lehetnek:
  - `OnFieldChange`: amint váltjuk a mezőt, automatikusan meghívja a `submit()` utasítást
  - `OnRowChange`: amint váltjuk a sort, automatikusan meghívja a `submit()` utasítást
  - `OnManualSubmit`: nem történik változtatás, amíg meg nem hívjuk a mentés (`submitAll()`) vagy visszavonás (`revertAll()`) műveletét
- a mentő műveletek hamissal térnek vissza sikertelen mentéskor, ekkor a `lastError()` tartalmazza a hibát

# Adatbázis-kezelés modell/nézet architektúrában

## Tranzakciók

- Lehetőségünk van az adatok konzisztencióját *tranzakciók* segítségével biztosítani (ha az adatbázis-kezelő támogatja)
  - a `transaction()` utasítás indítja a tranzakciót, amelyet a `commit()` utasítással véglegesíthetünk, a `rollback()` utasítással visszavonhatunk
  - amennyiben valamelyik utasítás hibásnak bizonyul, visszaállíthatjuk az adatbázis konzisztens állapotát, ezért célszerű használni a `submitAll()` utasítás esetén, pl.:

```
db.transaction(); // tranzakció indítása
if (model->submitAll()) // módosítások mentése
    db.commit(); // ha sikeres, véglegesítünk
else
    db.rollback(); // ha sikertelen, visszavonjuk
```

# Adatbázis-kezelés modell/nézet architektúrában

## Kapcsolt táblák kezelése

---

- Adatbázisbeli relációk segítségével kapcsolt adatokat a `QSqlRelationalTableModel` segítségével kezelhetünk
  - a `setRelation(<oszlop>, <reláció>)` metódussal beállíthatunk relációt egy adott oszlopra
  - a reláció típusa `QSqlRelation`, megadja a tábla nevét, a forrás (társított), valamint a cél (megjelenített) oszlopot
  - pl.:

```
QSqlRelationalTableModel model; // modell
model.setTable("myTable"); // tábla beállítása
model.setRelation(2,
    QSqlRelation("otherTable", 0, 1));
// reláció megadása
```

# Adatbázis-kezelés modell/nézet architektúrában

## Kapcsolt táblák kezelése

---

- A relációval kapcsolt tábla egyúttal egy külön modellt is létrehoz az alkalmazásban, amelyet külön lekérdezhetünk és szerkeszthetünk
  - a `relationModel(<oszlop>)` metódus visszaadja a csatolt táblához tartozó modellt
  - pl.:

```
model.setRelation(2,  
    QSqlRelation("otherTable",0,1));  
    // reláció megadása  
QSqlTableModel *otherModel =  
    model.relationModel(2);  
    // relációval kapcsolat tábla lekérdezése  
otherModel->data(...); // adat lekérdezése
```

# Adatbázis-kezelés modell/nézet architektúrában

## Kapcsolt táblák megjelenítése

---

- A társított adatokat nem csak szövegesen, hanem legördülő menü segítségével is megjeleníthetjük
  - a nézet `setItemDelegate(<delegált>)` metódusa segítségével állíthatunk be speciális delegáltat
  - a delegáltat az alapértelmezetről egy `QSqlRelationalDelegate` példányra kell lecserélnünk
  - pl.:

```
QTableView view; // nézet
view.setModel(model); // modell beállítása
view.setItemDelegate(new
    QSqlRelationalDelegate());
// delegált beállítása
```

# Adatbázis-kezelés modell/nézet architektúrában

## Példa

---

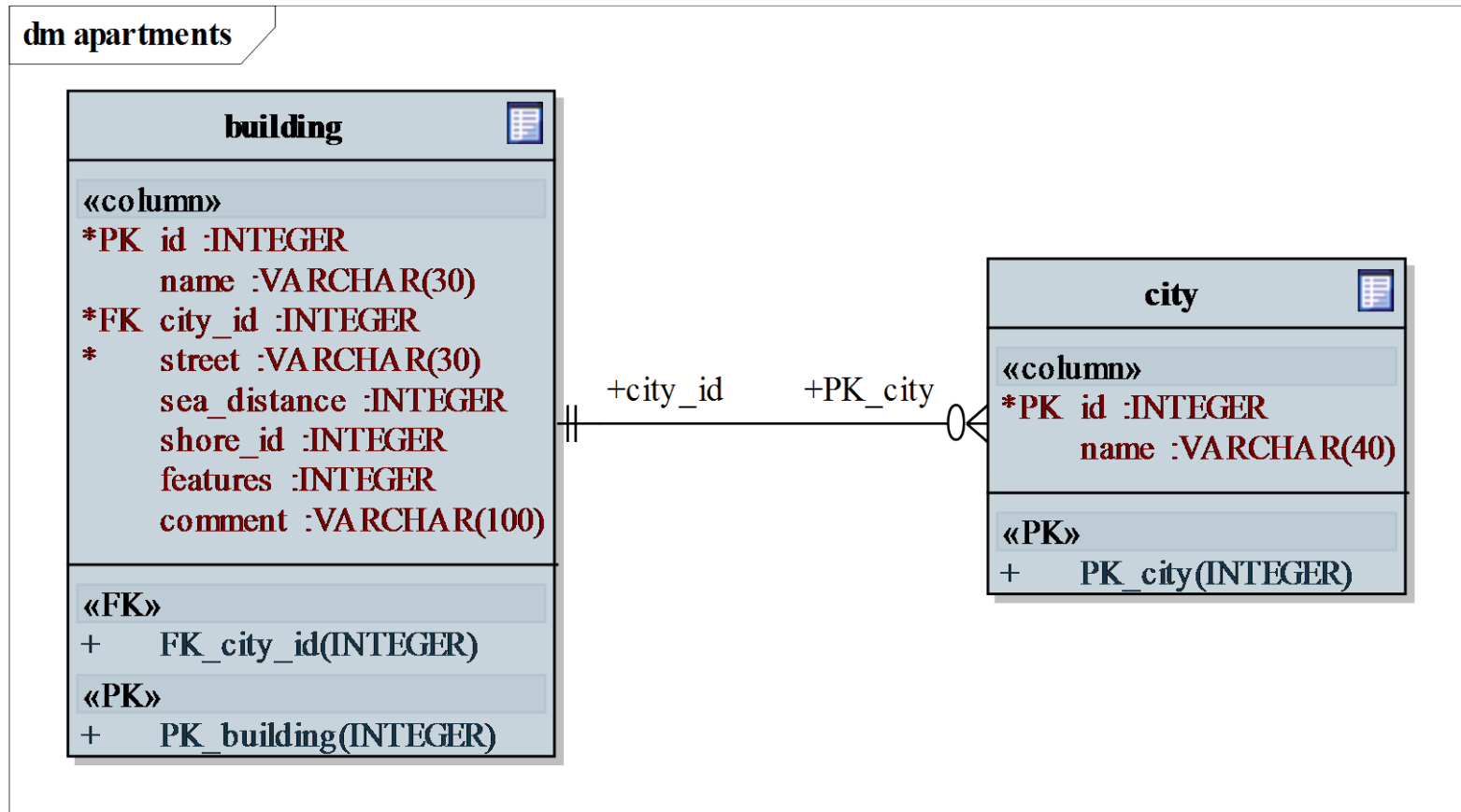
*Feladat:* Módosítsuk az épületek szerkesztését úgy, hogy a városokat hozzácsatoljuk az megjelenítéshez

- ehhez relációs adatmodellt kell használnunk, amely létrehozza a relációt a városok táblával (**city**), az épületek táblabeli azonosítót (**city\_id**) kötve az azonosítóhoz (**id**), és helyette megjelenítve a nevet (**name**)
- a megjelenítéshez lecseréljük a delegáltat is, így legördülő menü fog megjelenni
- az adatok mentését manuálisan valósítjuk meg tranzakciók segítségével egy külön gombbal

# Adatbázis-kezelés modell/nézet architektúrában

## Példa

*Tervezés (adatbázis):*

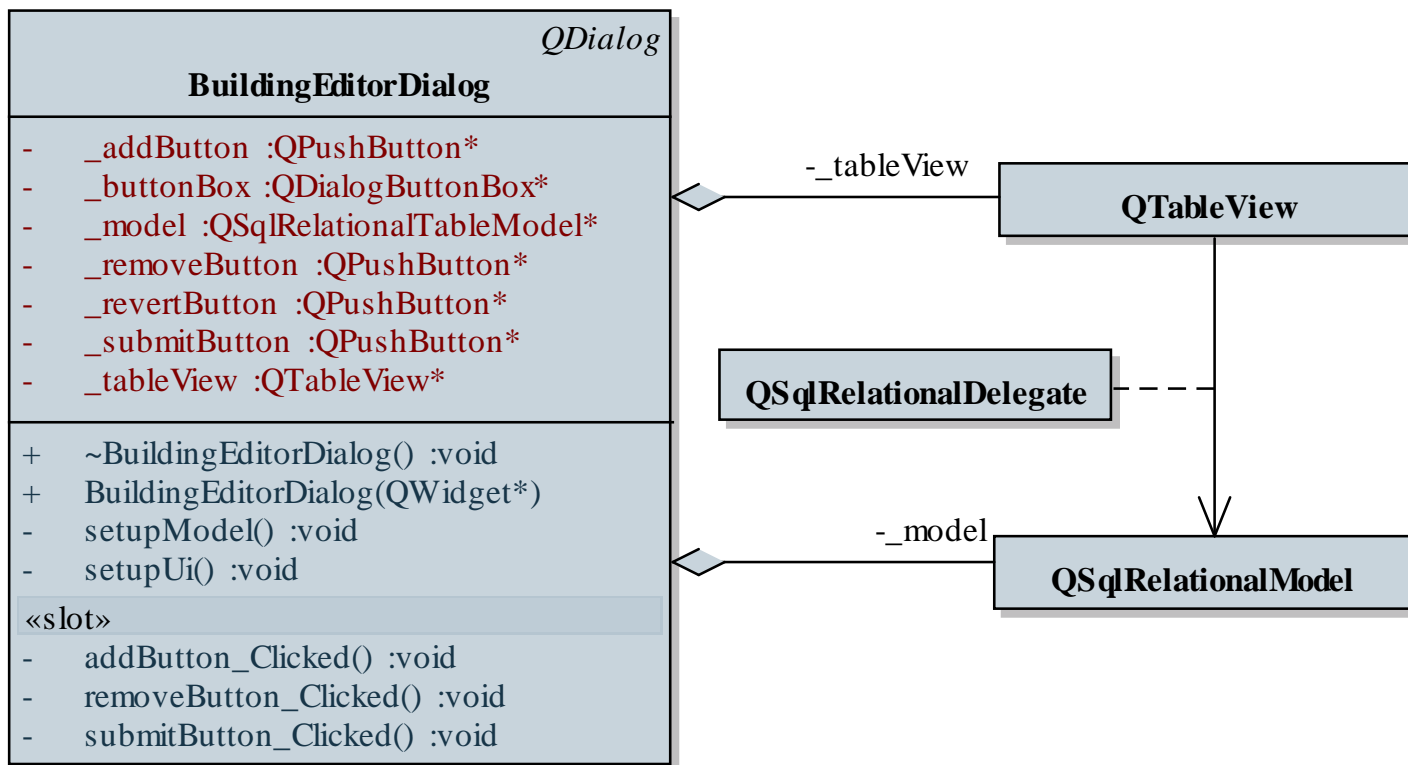


# Adatbázis-kezelés modell/nézet architektúrában

## Példa

*Tervezés (alkalmazás):*

class **BuildingEditorWithCities**



# Adatbázis-kezelés modell/nézet architektúrában

## Példa

*Megvalósítás* (buildingeditordialog.cpp):

```
...
_model = new QSqlRelationalTableModel(this);
_model->setRelation(2,
    QSqlRelation("city", "id", "name"));
    // reláció beállítása egy oszlophoz
...
_tableView = new QTableView(this);
...
_tableView->setItemDelegate(
    new QSqlRelationalDelegate());
    // megjelenítés módjának definiálása
```

# Adatbázis-kezelés modell/nézet architektúrában

## Példa

*Megvalósítás* (`buildingeditordialog.cpp`):

```
...  
if (model->submitAll()) { // mentés  
    _model->database().commit();  
}  
else { // amennyiben sikertelen volt  
    _model->database().rollback(); // visszavonjuk  
    QMessageBox::warning(this,  
        trUtf8("Hiba történt a mentéskor!"),  
        trUtf8("Az adatbázis a következő hibát  
            jelezte: %1").  
            arg(model->lastError().text()));  
    // jelezzük a hibát  
}
```