



Varga Péter

Qt strandkönyv

Átfogó bevezetés a Qt keretrendszer használatába



SZÉCHENYI TERV

Varga Péter:

Qt strandkönyv

Elektronikus kiadás

Kiadó: E-Közigazgatási Szabad Szoftver Kompetencia Központ, 2013

ISBN 978-963-08-8255-2

Szerzői jog

Ez a könyv a Creative Commons Attribution-ShareAlike 3.0 Unported (CC-BY-SA 3.0) licenc szerint szabadon terjeszthető és módosítható. További információk: <http://creativecommons.org/licenses/by-sa/3.0/>

A dokumentumban található összes védjegy a jogos tulajdonosait illeti meg.

Kiadva: 2013. december 17.

Tartalom

BEVEZETÉS	6
MIÉRT PONT QT?	7
NÉHÁNY EGYÉB TUDNIVALÓ	8
1. PÁR SZÓ AZ ESEMÉNYVEZÉRELT PROGRAMOKRÓL	9
2. ELSŐ PROGRAMUNK. IGEN, A „HELLÓ VILÁG!”, DE KICSIT BŐVEBBEN	10
2.1. A grafikus objektumok és a Property Editor	10
2.2. A signals and slots mechanizmus	10
2.3. Elég a kattintgatásból!	11
3. SAJÁT SIGNAL ÉS SLOT MEGVALÓSÍTÁSA – MEG EGY KIS QDEBUG ÉS QMAP	13
3.1. Először slot-ot készítünk	13
3.2. A QMap és a köszönések	15
3.3. Most pedig signal-t írunk	16
4. AKINEK NINCS ESZE	18
4.1. A QListWidget színre lép, és megismerkedünk az elrendezések fontosságával	18
4.2. A QListWidget elemeinek manipulációja	20
5. MENTSÜK, AMI MENTHETŐ!	24
5.1. QStringList a feladatok átadására	25
5.2. Menük és QAction-ök	25
5.3. A FileOperator osztályunk – QFileDialog, QMessageBox és egyéb veszedelmek	25
5.4. Iterátor à la STL és à la Java – na és a foreach	32
6. A FELADATOKHOZ KATEGÓRIÁKAT RENDELÜNK	34
6.1. A Task osztály	34
6.2. A QList<T> sablonosztály, és némi örület a mutatókkal. Megismerkedünk a QSharedPointer osztállyal	34
6.3. A ToDoList-et felkészítjük a kategóriák használatára, de még nem használunk kategóriákat	36
6.4. Kategóriák költöznek a ToDoList-be	39
7. MÓDOSÍTHATÓ FELADATOK, FELAJÁNLOTT KATEGÓRIÁK ÉS ALAPÉRTELMEZETTEN BETÖLTÖTT FELADATLISTA-FÁJL	41
7.1. A már megadott feladatok módosítása	41
7.2. Kategóriák felajánlása a QCompleter osztály használatával. Első pillantás a modellekre	43
7.3. Automatikus fájlbetöltés a program indulásakor – a QSettings osztály	46

8. KERESÉS, GRAFIKON ÉS TÖBBNYELVŰSÉG	49
8.1. A keresőfül.....	49
8.2. Grafikussá válva grafikont rajzolunk, és a tetejébe még iterálgatunk is	50
8.3. Többnyelvű alkalmazás és a QT Linguist	56
9. IKONOK, ABLAKOK, AUTOMATIKUS MENTÉS.....	59
9.1. Ikonok és QAction-ök a QToolBar-on	59
9.2. Módos és egyéb párbeszédablakok, a QTextBrowser és újabb adalékok az erőforrásfájlok természetéhez.....	61
9.3. Automatikus mentés a QTimer osztály használatával	64
10. KOMOLYAN HASZNÁLNI KEZDJÜK A MODEL–VIEW MINTA LEHETŐSÉGEIT	69
10.1. Elem-alapú és modell-alapú widget-ek.....	69
10.2. Újratervezés.....	70
10.3. Mit tartunk meg az előző változatból?.....	70
10.4. A főablak kialakítása	71
10.5. A modell, meg az osztály, ami kezeli.....	72
10.6. Kényelem mindenekfelett.....	74
10.7. Bevetjük a QItemSelectionModel osztályt	75
11. ADATMENTÉS ÉS –VISSZATÖLTÉS MODELL HASZNÁLATAKOR	82
11.1. A Beállítások ablak.....	82
11.2. A FileOperator osztály reinkarnációja.....	84
11.3. Utolsó lehetőség a mentésre – a QMessageBox osztály további lehetőségei és a Q_PROPERTY makró.....	89
11.4. Automatikus mentés és megnyitás	93
12. A MODELL, A DELEGÁLT, MEG A PROXYMODELL	94
12.1. Delegált megvalósítása a QStyledItemDelegate osztály használatával.....	94
12.2. A keresőfül és a QSortFilterProxyModel osztály	97
13. HÁLÓZATI SZINKRONIZÁCIÓ: A TODOList FELHŐCSKÉJE	98
13.1. A webszerver előkészítése	98
13.2. Kiegészítjük a beállítás-ablakot	100
13.3. Feltöltés a WebSynchronizer osztállyal – a QNetworkAccessManager és egyéb hálózati örökök.....	100
13.4. Letöltés a WebSynchronizer osztállyal – a QNetworkReply osztály és a fájlok kicsomagolása	107
14. TÖBBSZÁLÚ PROGRAMOT ÍRUNK.....	109
14.1. Tanulmányprogram a szálak tanulmányozására	109
14.2. Szálakat építünk a ToDoList-be	113
15. XML–FÁJLBA MENTJÜK A FELADATLISTÁT.....	116

16. A ToDoList, mint SQL-kliens	120
16.1. Gyomlálunk	120
16.2. Első QSqlTableModel-ünk	121
16.3. Törlés és visszavonása – barkácsoljunk kézzel QSqlRecord osztályú objektumot! ..	127
16.4. Keresés és kiegészítés a QSqlQueryModel osztállyal	129
17. Egyszerű Qt-program csomagolása Linux-rendszerekhez	132
17.1. Telepítőcsomag készítése Debian alapú rendszerekre	133
18. Kirándulás a Qt Quick-szigetekre (meg a QML tengerébe)	134
18.1. QML-programot írunk	134
18.2. Oda-vissza Qt és Qt Quick között: így beszélgetnek egymással a hibrid programok részei	139
BÚCSÚ	145
MILYEN OSZTÁLYOKRÓL, MAKRÓKRÓL ÉS EGYÉB SZÉPSÉGEKRŐL VAN SZÓ A KÖNYVBEN?	146

BEVEZETÉS

A Qt strandkönyv a Qt keretrendszerről szól. Nincs benne románc, feszültség és dráma, azaz végső soron nem is igazi strandkönyv, de egy kicsit mégis: habkönnyű és fájdalommentes ismerkedést kínál a keretrendszerrel, annak főbb lehetőségeivel.

Szerzőt, szerkesztőt, lektort az a cél vezérelte, hogy a programozó vagy programozópalánta anélkül ismerkedhessen meg a Qt-vel (vagy Qt-tal, merthogy a Qt kiejtése körülbelül „kjút”, azaz cuki, aranyos, édes), hogy úgy istenigazából neki kéne állnia a tanulásnak. Nincs hát szó arról, hogy az Olvasó referenciaművet böngéssze éppen. Már csak azért sem, mert maga a Qt óriási, és elég gyorsan változik ahhoz, hogy mire elkészülne egy igazán átfogó és szakmailag korrekt mű, már lennének olyan pontok, ahol elavultnak számítana.

Manapság jó sok szakkönyvön az olvasható, hogy mindenkinek csak a javára szolgál, ha elolvassa. Itt is szerepelhetne az, hogy egyaránt hasznos a telefonjával bajlódó kisiskolástól kezdve mindenkinek, akinek valami elektromos kerül a kezébe, legyen az akár egy hajszárító. De azért igyekszünk ennél korrektebbek lenni.

A könyv ideális olvasója az a programozni tanuló ember, aki túl van a C++-szal való ismerkedés első élményein (vagy megrázkódtatásain – nézőpont kérdése a dolog). Írt már pár túlterhelt függvényt, nem ijed meg, ha karakterláncokat kell tömbben tárolnia és a tömböt rendeznie. Elkészült az első néhány objektumával, és reggelire példányosít magának vajás kenyeret. És, ami a C++-szal foglalkozók esetében elengedhetetlen: látott már mutatót, érti a memóriaszivárgás fogalmát – de nem kell vérprofi mutatózsonglőrnek lennie.

Mi az, amit még nem kell tudnia? Nem kell ismernie a C++ szabványos könyvtárait, és nem kell tudnia eseményvezérelt programokat írnia. Az egyikre azért nem lesz szüksége, mert úgy adódott, hogy a Qt eszközkészlete pont elég lesz nekünk, a másakra meg azért nem, mert majd most úgyis megtanulja. (Egyébként még egy csomó mindent nem kell tudnia, de nem soroljuk fel mindet.)

Kívánjuk, hogy a következő oldalak szolgáljanak az Olvasónak épülésére, egyben kikapcsolódásul.

Gárdony, 2013. augusztus-október

Varga Péter

MIÉRT PONT QT?

Ma, amikor programozási nyelvek garmadájából válogathatunk, ugyan mégis mért választaná az ember a Qt keretrendszert? Lássunk néhány érvet!

C + + alapok

A Qt történetesen a világ egyik legismertebb és legelismertebb nyelvét egészíti ki. A használt kiegészítésekből a Meta Object Compiler (becenevén: moc) a fordítási folyamat során szabványos C + + kódot állít elő. Ezt aztán akár a GCC, akár a Visual Studio, akár a MinGW képes lefordítani.

Más nyelvből

Az a helyzet, hogy a Qt elemeinek használata nem csak C + +-ból lehetséges. Az elemkészlet használható Java, Python, Perl, Ruby és más nyelvekből.¹ Ebben a könyvben maradunk a C + +-nál, de ez ne tántorítson el senkit a más nyelvekkel való használatától.

Multiplatform fejlesztés

Nem is. Inkább multiplatform fordítás és futtatás. A fejlesztést végezhetjük kedvenc asztali operációs rendszerünkön (az IDE fut Windowson, Linuxon és OS X-en), de a forrás többféle rendszerre is lefordítható. Android és iPhone támogatás a Qt 5.2-ben érkezik – persze lehet, hogy mire e sorokat a nagyközönség is olvasni fogja, már az „érkezett” forma a helyes: a Qt 5.2-t 2013 őszére várjuk.

Jó IDE

A Qt Creator természetesen végez automatikus kiegészítést, van környezetérzékeny ságója. Kezel projekteket, támogat rengeteg verziókövetőt. Mobileszközök esetében a Qt Creator elkészíti a telepítőcsomagot és telepíti a fejlesztő gépéhez csatlakoztatott eszközre.

Sokféle licencelés

E könyv írásakor a Qt programunk licencelése lehet kereskedelmi, ha ezért vagy azért nem szándékszunk közzé adni művünk forráskódját. Ha pedig szándékszunk, választhatunk az LGPL 2.1 és a GPL 3.0 licenc közül, azaz megvan a módunk arra, hogy korszerű szabadszoftveres licencet használjunk, akár azért, mert ezt követelő kódot építünk a termékünkbe, akár azért, mert úgy keltünk fel reggel.

Közösségi támogatás és túlélés

A Qt jelenlegi tulajdonosa, a Digia vezeti a támogatókat – akár az Olvasót is – tömörítő -Qt Projectet. A KDE Free Qt Foundation nevű alapítvány pedig olyan jogok birtokában van, amelyek a Digia felvásárlása, csődbemenetele, vagy más céggel való összeolvadása esetén lehetővé teszik az alapítvány számára a Qt Free Edition szabadszoftveres licenc alatti közreadását. Azaz elvileg nem érheti ilyen okból csúnya baj a Qt-re alapozott hosszú távú üzletünket, és nem kell attól félnünk, hogy egy menedzser irodájában meghozott döntés miatt egyszer csak kihúzzák a talpunk alól kedvenc keretrendszerünket.

1 <http://qt-project.org/wiki/Category:LanguageBindings>

NÉHÁNY EGYÉB TUDNIVALÓ

Egyezzünk meg két dologban

Ebben a könyvben a forráskód nyelve teljes egészében az angol, még akkor is, ha a könyv maga magyar nyelvű. Elismerve, hogy megvannak az érvek a magyar nyelvű változó-, függvény- és osztálynevek mellett is, rámutatunk, hogy használatuk hibás gyakorlat kialakulásához vezet. Manapság a programozó jó eséllyel nemzetközi projektekben is részt vesz, forráskódját más nyelven beszélőknek is olvasniuk, módosítaniuk, használniuk kell. Különösen igaz ez akkor, ha a kezdő programozó szabad szoftveres projektben való részvétellel igyekszik megszerezni azt a munkatapasztalatot, referenciát, amelynek birtokában a siker lényegesen nagyobb esélyével mehet el állásinterjúra.

A programjaink eleinte magyarul szólnak a felhasználóhoz, lévén így fény derül pár dologra (igen, az ékezetek terén), ami amúgy sötét homályban maradna.

A szakszavakat csak akkor fordítjuk magyarra, ha a magyar változat használata a magyar szaknyelvben bevett, kikristályosodott. Azért is vagyunk óvatosak a fordítással, mert az angol nyelvű internetes szakirodalom történetesen gazdagabb még a magyarnál is. Kifejezetten zavaró, ha a magyar nyelvű szakkifejezés angol megfelelője nem jut eszünkbe, és ennek folyományaként nem tudunk jól keresni az interneten. A jó magyar szakkifejezés persze megkönnyíti az ismeretlen fogalom megértését, főleg annak, akinek (még) nem az angol a második anyanyelve.

A könyvben lévő forráskódok

A forráskódok a könyv egészéhez hasonlóan CC-BY-SA licencűek, azaz a szerző feltüntetése mellett azonos licenccel közreadhatók, származékos mű készülhet belőlük.² Már amennyiben a könyv végigolvasása után valaki még rászorulna ilyesmire.

A kódokat a legritkább esetben közöljük teljes egészében, mindazonáltal letölthetők a <http://szabadszoftver.kormany.hu/qtstrandkonyv> webhelyről.

Mire lesz szükségünk?

A Qt Library-ra, és a Qt Creator-ra. Mindkettő letölthető Windows, Linux és Mac rendszerre a qt-project.org webhelyről, bár Linuxon jó eséllyel a disztribúciónk csomagkezelőjével érdemesebb telepíteniük.

² Részletesebben: <http://creativecommons.org/licenses/by-sa/3.0/>

1. PÁR SZÓ AZ ESEMÉNYVEZÉRELT PROGRAMOKRÓL

Aki már írt grafikus felülettel bíró programot, bátran lapozzon előre a következő fejezetig – nem lesz messze. A többiek olvassanak tovább.

Amikor az ember programozni tanul, alighanem tudomást szerez arról, hogy a programok szekvenciálisan, azaz az utasítások sorrendjében futnak. Van természetesen néhány struktúra, nyelvi elem, ami bonyolítja a helyzetet, nevezetesen az elágazások és a ciklusok. Ilyenkor ugye a sok programsor közül pár kimarad, vagy éppen többször is lefut. Tovább bonyolítják a helyzetet az eljárások és a függvények. De azért valamilyen értelemben mégis szekvencia marad a szekvencia: mindig megmutatható, hogy épp hol tart a program, és lehet tudni, hogy mi lesz a következő dolga.

Nos, más a helyzet egy eseményvezérelt programban.

Itt ugyanis van egy fő ciklus, ami arra vár, hogy bekövetkezzen valamilyen esemény. Az esemény egyfelől lehet hardveres eredetű: billentyűnyomás, egérmozdulat, vagy -kattintás, egyéb periféria által kiváltott megszakítás, másfelől a program egyes részei is kiválthatnak eseményeket, például a fájlmentésért felelős rész szól, hogy betelt a háttértár. A program bezárása egyenértékű a fő ciklusból való kilépéssel.

A program a főcikluson kívül lényegében már „csak” eseménykezelőkből áll, a fő ciklus velük végezteti el a bekövetkezett események kezelést. Az eseménykezelők a mi esetünkben bizonyos objektumok tagfüggvényei lesznek. A tagfüggvények belseje természetesen megint csak szekvenciális, ugyanakkor lehetőségünk van a tagfüggvényekből eseményt kiváltani.

Azaz az eseményvezérelt program a mi értelmezésünkben egy vagy több objektum, egy vagy több tagfüggvénnyel, amelyek akkor futnak le, ha az általuk feldolgozandó esemény bekövetkezik.

Akkor nézzük mindezt most a Qt-n belül. Amikor létrehozunk egy grafikus alkalmazást, akkor a generált `main.cpp` fájl körülbelül ilyen lesz:

```
#include <QApplication>
#include "mainwindow.h"

int main(int argc, char *argv[])
{
    QApplication a(argc, argv);
    MainWindow w;
    w.show();

    return a.exec();
}
```

Az említett fő ciklus jelen esetben a `QApplication` osztályból példányosított `a` nevű objektum. A példányosítása után létrehozuk és meg is jelenítetjük a főablakot, majd meghívjuk a fő ciklus `exec` tagfüggvényét. Ez utóbbi tettünk hatására lendül munkába a fő ciklus, innentől végzi a dolgát.

2. ELSŐ PROGRAMUNK. IGEN, A „HELLÓ VILÁG!”, DE KICSIT BŐVEBBEN

2.1. A grafikus objektumok és a Property Editor

Indítsuk el a Qt Creator-t, felül kattintsunk a **Develop** fülre, majd az oldal közepe táján a **Create Project** lehetőségre. Azon belül válasszuk az **Applications** és a **Qt GUI Application** pontokat, végül pedig a **Choose** gombot. Adjunk nevet a készülő műnek, és pár **Next** után az első adandó alkalommal kattintsunk a **Finish** gombra. Létrejön a projektünk, benne a **main.cpp**, illetőleg, ha nem babráltunk a beállításokkal, a **mainwindow.cpp** és a **mainwindow.h** fájl.

Aki türelmetlen, már fordíthatja és futtathatja is a gépén lévő legújabb alkalmazást, akár úgy, hogy a bal oldali sáv alsó része felé található lejátszás-gombra kattint, akár a **Ctrl+R** billentyűkombináció használatával. Megjelenik egy **MainWindow** címsorú ablak, amit lehet mozgatni, átméretezni, meg még be is zárható. Tegyük is meg: annyira azért még nem remek ez a program, hogy órákig ellegyünk vele.

A projekt fájljait megjelenítő oszlopban válasszuk a **Forms** lehetőséget, azon belül pedig az ott árválkodó **mainwindow.ui** fájlt. Ekkor a képernyő átalakul. A fájllista helyén a használható grafikus objektumok listáját találjuk, középen pedig, a forráskódszerkesztő helyén maga a **MainWindow** csücsül, bár most épp a **Type Here** felirat olvasható rajta.

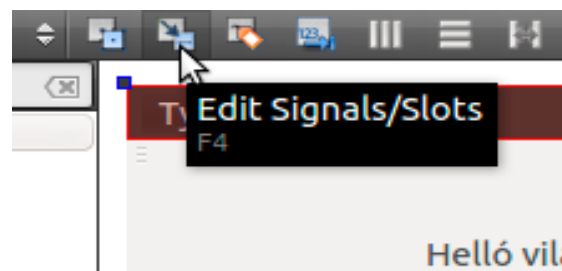
Bal oldalról, az objektumok listájából válasszuk a **Label**-t. Ha nem találjuk, akkor a fenti, **Filter** mezőben elkezdve gépelni a nevét, alighanem előkerül. Húzzuk rá a középen lévő ablakra. A felirata (**TextEdit**) rákattintva is megváltoztatható, de most mégis inkább a képernyő jobb alsó sarkában lévő **Property Editor** (tulajdonságszerkesztő) részen fogjuk módosítani.

A **Property Editor** arra való, hogy egy grafikus objektumot könnyen áttekinthessen és módosíthasson a fejlesztő. Jelen sorok szerzője a Delphi nevű programozási környezet első kiadásában, még Windows 3.1 alatt látott először ilyet (1995), azaz megkockáztathatjuk, hogy a koncepció nem új. Keressük meg a **text** tulajdonságot, és módosítsuk például „Helló világ!”-ra. Ha azt látjuk, hogy a felirat kilóg a **QLabel** osztályú objektumunkból (igen, minden Qt-osztály neve Q-val kezdődik), akkor módosítsuk a méretet akár az egerünkkel, akár a **Property Editor**-ban, a **geometry** részen. Futtassuk most a művünket, és dicsekedjünk el valakinek, például az aranyhörcsögünknek. Megvan az első Qt-programunk!

2.2. A signals and slots mechanizmus

Sem a grafikus fejlesztő, sem a **Property Editor** nem teszi egyedivé a Qt-t. A signals and slots mechanizmus azonban igen: a dolog teljes egészében Qt-specifikus.

Az előző projektünkben kiindulva húzzunk ki egy **QPushButton**-t a főablakra, majd állítsuk át a szöveget „Kilépés”-re (most is a **text** tulajdonság a barátunk). Fent, a menüsor alatt keressük meg a jobbra lefelé mutató nyilat ábrázoló ikont, melynek a súgója „**Edit Signals/Slots**”. Kattintsunk rá. Azon felül, hogy bal oldalt

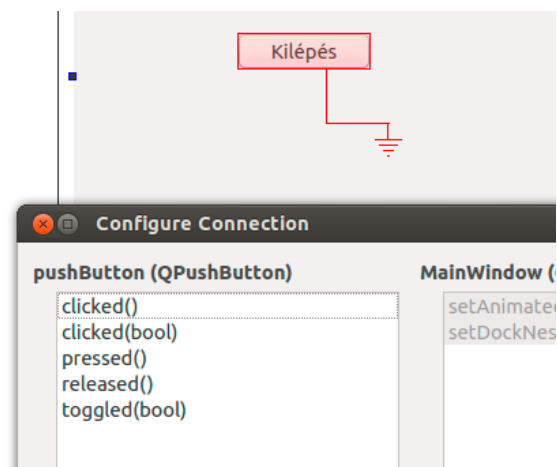


1. ábra: A Signals and Slots ikon

kiszűrjük az objektumok listája, nem sokat látunk, de ha megpróbáljuk a gombot odébbhúzni, akkor nem fog menni, helyette nyilat rajzolunk.

A nyilat engedjük el valahol az ablak területe fölött, és megjelenik egy új dialógusablak. Itt lehetőségünk nyílik a `QPushButton` valamely signal-ját (jelét) a `MainWindow` valamelyik slot³-jához kötni.

A `QPushButton` signal-jai közül választuk a `clicked()`-et, a `MainWindow`-nál azonban nem látszik az, amit keresünk. Tegyük pipát a **Show signals and slots inherited from QWidget** (a `QWidget`-től örökölt jelek és csapdák megjelenítése) mellé, és ekkor megjelenik a `close()` slot. Jelöljük ki, és kattintsunk az OK gombra. Futtassuk a programot, és kattintsunk a gombra. Olé!



2. ábra: Signal-t kötünk slot-hoz

Mit is végeztünk?

A Qt-ban az objektumok képesek magukból signal-okat kiadni, „kisugározni”, szakzsargonban emittálni. Ezek olyanok, mint amikor valaki ordít az erdőben: korántsem biztos, hogy bárki is hallja a zajt. Egy jól nevelt Qt-objektum (és majd figyelünk, hogy az általunk írtak ilyenek legyenek), csak akkor kiabál, ha történt vele valami fontos – például megváltozott az állapota. A nyomógombunkkal most pont ilyesmi történt: rákattintottunk.

A Qt-ban az objektumoknak lehetnek slot-jaik is, azaz olyan speciális tagfüggvényeik, melyek képesek hallgatózni: figyelni arra, ha egy másik objektum kiabál.

Amikor a signal-t a slot-hoz kötjük, tulajdonképp elmondjuk a slot-nak, hogy melyik objektum melyik kiabálására kell figyelnie. Figyeljünk fel arra a tényre, hogy a kiabáló objektumnak fogalma sincs róla, hogy valaki figyel-e rá: ő csak kiabál. A kapcsolattól, azért, hogy történjen valami, lényegében a slot objektuma felel.

Amikor lenyomtuk a gombot, az elkiáltotta magát, hogy „Rám kattintottak!”, és a `MainWindow close()` slot-ja ezt meghallva sürgősen bezárta az ablakot.

Hozzunk létre egy másik gombot (csak akkor fog menni, ha a fenti ikonsoron a most lenyomottól balra lévő, **Edit Widgets** ikonra kattintunk előbb), legyen a szöveg rajta „Csitt!” és ezt kössük a `QLabel`-hez, a signal-ok közül válasszuk megint a `clicked()`-et, a slot-ok közül meg azt az egyet, amit lehet: a `clear()`-t. Próbáljuk ki ezt a művünket is.

2.3. Elég a kattintgatásból!

Ez persze nem igaz, de a következő feladat megvalósításához már kénytelenek leszünk a billentyűzetet is használni. Hozzunk létre még egy gombot, és erre írjuk ki azt, hogy „Köszönj másképp!” A feladat az, hogy erre kattintva változtassuk meg a `QLabel` szövegét. Igen ám, de hát hogyan? A jó kis **Configure Connection** ablak most nem segít. Miért?

- 3 A slot szó többek közt bevágást, csapdát jelent. Ez utóbbi jelentése körülbelül megfelel a jelenlegi szerepének, de nem pontos abban a tekintetben, hogy a slot olyasmit jelöl, amibe beleillik az, amit bele akarunk tenni.

Ha megnézzük a `QLabel` osztály dokumentációját (akár a beépített súgóban, `F1`-et nyomva az objektumon állva, akár az interneten, a qt-project.org webhelyen), látni fogjuk, hogy a slot-jai között van olyan, hogy `setText()`, azaz szöveg beállítása. A visszatérés típusát (`void`) meglátva (mármint azt, hogy egyáltalán van neki ilyen), joggal gyanakszunk arra, hogy *a slot-ok tulajdonképp speciális tagfüggvények*. A tagfüggvényeknek meg ugye van deklarációjuk, benne paraméterlistájuk. És itt lesz a kutya elásva. *A signal-lal a küldő (sender) objektumnak pont ugyanolyan típusú adatot kell emittálnia, mint amilyet a kapó (receiver) slot-ja vár.*⁴

Esetünkben ugye meg az a helyzet, hogy a `QPushButton.clicked()` nem ad olyan adatot (`QString` típusút), amit a `QLabel.setText(const QString &)` várna. Akkor most mi lesz?

Kattintsunk jobb egérgombbal a nyomógombra, és válasszuk a `Go to slot...` lehetőséget a helyi menüből. A signal lehet most is a `clicked()`. Visszakerülünk a kódszerkesztőbe, és megíródott nekünk egy fél függvény, nevezetesen ott a függvénydeklaráció, csak a függvény törzsét kell megadnunk. Mielőtt ezt tennénk, nézzük meg a függvény prototípusát a `mainwindow.h` állományban (kattintgatunk, vagy `F4`-et nyomunk). Az osztálydefiníció jelen állapot szerint így néz ki:

```
class MainWindow : public QMainWindow
{
    Q_OBJECT

public:
    explicit MainWindow(QWidget *parent = 0);
    ~MainWindow();

private slots:
    void on_pushButton_3_clicked();

private:
    Ui::MainWindow *ui;
};
```

Látjuk, hogy a függvény prototípusa nem a szokványos `public/private` területen, hanem az egyszeri `C++`-programozó számára ismeretlen helyen, a `private slots` részen van. Most, hogy ezt megnéztük, mehetünk vissza függvénytörzset írni.

Ha figyelmesen szemléljük a `mainwindow.cpp` fájl tartalmát, észrevesszük, hogy van egy `ui` (user interface, felhasználói felület) nevű objektum. A főablak elemeit ezen az objektumon belül példányosítja a Qt Creator. Így aztán a harmadik gombunk lenyomásakor hívott slot-ot így kell megírunk:

```
void MainWindow::on_pushButton_3_clicked()
{
    ui->label->setText("Szia neked is, hörcsög!");
}
```

A szöveg persze így nem fog kiférni, húzzuk szélesebbre a `QLabel`-t. És, ha eddig nem tettük volna, most már tényleg mutassuk meg az aranyhörcsögünknek a programunkat.

A következő fejezetben írunk saját signal-t, meg saját slot-ot.

4 Ez a Qt 5 esetében nem teljesen igaz: ha a típusok között implicit átalakítás lehetséges, akkor használhatunk más típust is. Mi a könyvünkben e tekintetben a Qt 4 szigorúbb megközelítést használjuk.

3. SAJÁT SIGNAL ÉS SLOT MEGVALÓSÍTÁSA – MEG EGY KIS QDEBUG ÉS QMAP

Ebben a fejezetben egy olyan programcskát írunk meg, amelyik visszaköszön, még hozzá „értelmesen”. Nem óhajtjuk persze elvenni a mesterséges intelligenciával foglalkozók kenyerét, ezért magunkat visszafogva a következők megvalósítását tűzzük ki célul:

Lesz egy `MainWindow`-unk, rajta egy szövegbeviteli mezővel, egy `QLineEdit`-tel. Egy másik objektumból figyeljük, hogy mi került a szövegmezőbe, és ha épp felkiáltójelhez érünk, akkor feltételezzük, hogy vége a köszönésnek. Ha a köszönés „Szia!”, akkor visszaköszönünk, hogy „Szevasz!”, ha ellenben „Jó napot!”, akkor „Üdvözlöm!” lesz a válaszuk. Nem állítjuk, hogy ennek a bonyolult feladatnak a kivitelezéséhez valóban másik osztályt kell írunk, de ez ugye egy strandkönyv, és a strandkönyvek ritkán jelentik a valóság pontos leképezését.

3.1. Először slot-ot készítünk

Kezdjünk hát új projektet, a neve legyen `polite`. Tegyük ki a `MainWindow` ablakra egy `QLineEdit` objektumot, és a `Property Editor`-ban kereszteljük át az objektumot `greetingLine`-ra az `objectName` tulajdonság állításával. A Qt Creator bal oldali sávjában kattintsunk az `Edit` lehetőségre, majd a `Sources` felíratra, vagy a projekt nevére (Qt Creator verziótól függően) a jobb egérgombbal. A felbukkanó menüből válasszuk az `Add new...` lehetőséget, a megnyíló ablakból a `C++`-t, azon belül pedig a `C++ Class`-t. A `Choose` gombra kattintva új ablakot kapunk, itt adjuk a `Greeter` nevet az osztálynak, és állítsuk be a `QObject`-et alaposztályként (`Base Class`).

Mikor elkészült az osztályunk, keressük meg a `greeter.h` fájlt, és nézzük meg. Az osztálydeklaráció alatt látni fogunk egy `Q_OBJECT` makrót. Egy ujjal se nyúljunk hozzá, mert akkor ebben a fájlban fordításkor nem néz körül a moc (amivel a Miért pont Qt? részben ismerkedtünk meg futólag), és nem készít a Qt-specifikus nyelvi elemekből olyan kódot, amit már egy szabvány C++-fordító is megért.

Itt a fejlécfájlból hozzá is fogunk saját slot-unk megírásához: a `public slots` részen helyezzük el a `greetingLineTextEdited()` nevű, `void` visszatérési értékű függvény prototípusát. Igen ám, de mi lesz a függvényünkhöz tartozó signal? A `QLineEdit` osztály ságóját böngészve kettő jelöltet is találunk. Az egyik prototípusa: `void textChanged(const QString & text)`, a másiké pedig: `void textEdited(const QString & text)`. Átfutva a dokumentációjukat, rájövünk, hogy tulajdonképp mindegy, melyiket használjuk, ha mégis választanunk kell, akkor talán az utóbbit szeretnénk. Már most figyeljük meg, hogy a signal-nak a megváltozott szöveg *nem a visszatérési értéke*.

A `QString` típus, ha tetszik, a `QString` osztályú objektumok azok, ahol egy jól nevelt Qt-program a szöveges változóit tárolja, még hozzá Unicode kódolásban. Ha használni akarjuk – márpedig akarjuk –, akkor `#include`-olnunk kell, azaz helyezzük el a `greeter.h` fájl elején az

```
#include <QString>
```

sort.

Nos, pár oldallal korábban olyasmit állítottam, hogy a slot olyan típusú adatot kér, amelyet a signal emittál, azaz a `greetingLineTextEdited()` slot prototípusát megadó sor a következő:

```
void greetingLineTextEdited(const QString&);
```

Nyomjunk a soron jobb egérgombot, és a helyi menüből válasszuk a **Refactor**, azon belül az **Add Definition in greeter.cpp** lehetőséget, vagy ha ez a lassúság elviselhetetlen, akkor használjuk az **Alt+Enter** billentyűkombinációt, és nyomjunk **Enter**-t.

Átkerülünk a **greeter.cpp** fájlba, ott vár bennünket az üres függvénytörzs. Névtér beállítva. Szóval, ha a szöveg a **greetingLine** objektumban megváltozik, ez a függvény reagál majd. Mielőtt azon törnénk a fejünket, hogy pontosan mi legyen a dolga, nézzük meg, hogy tényleg működik-e a signal-slot mechanizmus. Adjuk meg, hogy a paraméter neve legyen **greeting**, majd írassuk ki, hogy mit kaptunk. A kiírást a **QDebug()** függvénnyel végezzük. Használatához a **greeter.cpp** fájl elején szükségünk van egy

```
#include <QDebug>
```

sorra. Figyeljünk, hogy mikor **QDebug()** és mikor **<QDebug>**, mikor kisbetűs, mikor nagybetűs.

A függvény egésze mostanra tehát a következő formát ölti:

```
void Greeter::greetingLineTextEdited(const QString &greeting)
{
    qDebug() << greeting;
}
```

Most az a rész következik, hogy a **greeter.h**-t használatba vesszük a **mainwindow.h**-ban, azaz elhelyezzük az

```
#include "greeter.h"
```

sort a fájl eleje felé. Aztán egy

```
private:
    Greeter *greeter;
```

sorpárossal létrehozunk magunknak egy mutatót, és a **cpp**-fájlban, a **MainWindow** konstruktorában pedig példányosítunk magunknak egy objektumot a remek kis osztályunkból. Ekkorra a **MainWindow** konstruktora az alábbi formát ölti:

```
MainWindow::MainWindow(QWidget *parent) :
    QMainWindow(parent),
    ui(new Ui::MainWindow)
{
    greeter = new Greeter(this);
    ui->setupUi(this);
}
```

Amikor a **Greeter** szó után kitesszük a nyitó zárójelet, sugót kapunk arról, hogy itt a szülőobjektum mutatóját kéne megadnunk, aminek alapértelmezett értéke a 0. Mi azért adunk meg **this**-t, azaz magára a **MainWindow**-ra mutató mutatót, mert ha így járunk el, akkor működésbe lép a Qt memóriefelügyelete, nevezetesen az, hogy mutató ide vagy oda, a szülőobjektum megsemmisülése együtt jár majd a gyermekobjektum megsemmisülésével. Ez annyira fontos dolog, hogy elmondjuk másként is: ha csak a **QObject** osztály leszármazottaival dolgozunk, és figyelünk arra, hogy az objektumnak legyen szülője, elfelejtkezhetünk mindarról a mizériáról, amit a C++-ban a lefoglalt memória felszabadítása, pontosabban annak elmaradása okozni szokott.

A jó hír olvasatán remélhetőleg nem csak bennünket, de az aranyhörcsögöt is az öröm hullámai járják át. Ha öt esetleg mégsem, akkor adjunk neki pár centi sárgarépát.

Most már végre futtassuk a programunkat! Fordul, de működni például nem működik: a `qDebug()` hallgat. Na igen. Ki mondta meg a slot-nak, hogy melyik signal-ra figyeljen?!

Az `ui->setupUi(this)` sor után (vagyis, amikor a `greeter` mutató már jó helyre mutat, és helyén van a `greetingLine` nevű objektum is az `ui` belsejében valahol), írjuk be a következő sort – ami ráadásul, pusztán az olvashatóság mián két sorba kerül most is, meg később is így érdemes majd beírni.

```
connect(ui->greetingLine, SIGNAL(textEdited(QString)),
greeter, SLOT(greetingLineTextEdited(QString)));
```

Látjuk, hogy nem egészen szokványos C++-utasítás a `connect`. Megadjuk benne, hogy melyik objektum fogja a signal-t emittálni, a `SIGNAL()` zárójelében megadjuk, hogy melyik ez a signal, megadjuk a slot-ot tartalmazó objektumot, majd a `SLOT()` zárójelében, hogy pontosan melyik slot-ot kell a signal-hoz kapcsolni. Írtuk már, hogy a Qt 5 nem ragaszkodik teljesen azonos típusúhoz a signal és a slot esetében, meg azt is, hogy ebben a könyvben maradtunk a Qt 4 szigorúságánál. Ennek megfelelően a fenti szintaxis a Qt 4-é. A Qt 5-ben egy kicsit más is lehet⁵, de ez is működik.

Ha most futtatjuk a programunkat, ahogy elkezdünk gépelni a `greetingLine`-ba, a `qDebug()` telepokolja a képernyőre. Remek!

3.2. A QMap és a köszönések

Szóval a slot működik, reagál arra, ha írunk valamit a `greetingLine` szerkesztősorába. Akkor jöhet a logika. A köszönéseket egy megfeleltetési típus, a `QMap` használatával tároljuk. Az első dolgunk, hogy a `greeter.h` fejlécállomány elején jelezzük, hogy a `<QMap>` fejlécre is szükség van. Aztán deklarálunk egy privát változót a köszönéseknek, mégpedig így:

```
private:
    QMap<QString, QString> greetings;
```

Mit is jelent mindez? Létrehozunk egy `greetings` nevű objektumot, amely kulcs-érték párokból áll⁶. Ennek az objektumnak mind a kulcsai, mind az értékei `QString` típusúak. A `greeter` osztály konstruktorában fel is töltjük a szótárunkat, mind a két ismert köszönéssel, illetve a rájuk adandó válaszokkal.

```
Greeter::Greeter(QObject *parent) :
    QObject(parent)
{
    greetings["Jó napot kívánok!"] = "Üdvözlöm!";
    greetings.insert("Szia!", "Szevasz!");
}
```

Pusztán a sokszínűség jegyében kétféle módszert is bemutatunk a kulcs-érték párok megadására: az elsőben a kulcs szógletes zárójelbe kerül, és az értéket az értékadás-operátorral adjuk meg (de, komolyan), míg a második esetben az `insert` tagfüggvény paramétereiként adjuk meg a kulcsot is, és az értéket is. Nem tűnik túl bátor következtetésnek, hogy ez utóbbi esetben a paraméterek sorrendjének szerepe van.

⁵ <http://woboq.com/blog/new-signals-slots-syntax-in-qt5.html>

⁶ Python nyelvben ennek a típusnak külön neve is van: szótárnak hívják.

A slot-unkat pedig a következőképp alakítjuk át:

```
void Greeter::greetingLineTextEdited(const QString &greeting)
{
    if(greeting.endsWith("!")){
        if(greetings.contains(greeting)){
            qDebug() << greetings.value(greeting);
        }else{
            qDebug() << "Én nem ismerem kendet.";
        }
    }
}
```

Az `endsWith()` tagfüggvény neve magáért beszél. Nekünk azért kell, hogy ne köszöngessünk egyfolytában, hanem udvariasan megvárjuk, amíg a tisztelt felhasználó befejezi a köszönését, és csak akkor reagálunk. A `contains()` tagfüggvénnyel megvizsgáljuk, hogy számunkra értelmes-e a köszönés (van-e ilyen kulcs a `greetings` objektumban), és ha igen, akkor a `value` tagfüggvénnyel előkotortatjuk a kulcsnak megfelelő értéket. Ha meg nincs, akkor elküldjük a delikvenst.

Akkor eddig minden rendben, a program köszönget, vagy elküld, ahogy arra kértük. Még egy utolsó adalék a `QMap` osztály használatához: a `value()` tagfüggvény képes arra, hogy ha valamit nem talál, akkor egy előre beállított értéket ad vissza. Ennek fényében a slot-unk törzse az alábbi két sorra csökkenthető, anélkül, hogy a működése bármennyiben is változna:

A Qt 4 és az UTF-8

A művünket Qt 4 alatt tesztelve arra jutunk, hogy a „Szia!” köszönésre szépen reagál, de ha udvariaskodva köszönünk, akkor bizony nem ismer meg. Ez azért szomorú, mert érthető, hogy az ember programja fiatalos, na de hogy bunkó is?! Kis kutatás után rájövünk, hogy a hibát az ékezetes betűink okozzák: ha a kulcsot „Jo napot kivanok!”-ra módosítjuk, a dolog működni kezd. Akkor most Unicode a `QString`-ben tárolt adat, vagy sem? Igen, az, de nekünk meg UTF-8 a beviteli módunk: a forrásfájlban azt használunk meg a kimenet is olyan. És erről még nem szóltunk a Qt-nak. A megoldás az, hogy a legmagasabb olyan szinten, ahol értelmét látjuk, kiadjuk a

```
QTextCodec::setCodecForCStrings(QTextCodec::codecForName("UTF-8"));
```

parancsot. Érdemes akár a `MainWindow` konstruktorában első sorként megadni. Ha használni akarjuk a parancsot, szükségünk lesz a `<QTextCodec>` fejlécfájlra. Még valami: mostantól ennek a parancsnak a használatát nem hangsúlyozzuk: mindenkinek jusson eszébe magától.

```
if(greeting.endsWith("!!"))
    qDebug() << greetings.value(greeting, „Én nem ismerem kendet.");
```

3.3. Most pedig signal-t írunk

Eddig ugye a visszaköszönést a `qDebug()` függvénnyel írtuk ki, ami most még elmegy, de egy kiforrott szoftverben talán elvetendő megoldásnak minősítenénk. Szívünk mélyén érezzük, hogy inkább valahol a főablakban kellene megjelenítenünk, és akinek most a már ismerős `QLabel` villan be, annak azt mondjuk, hogy legyünk nagyratörőbbek! Mert a `MainWindow` aljában ott csücsül az állapotsor, ami történetesen egy `QStatusBar` osztályú objektum, s mint ilyen, rendelkezik `showMessage()` tagfüggvénnyel. Megnézve az osztály dokumentációját,

látjuk, hogy ez a függvény ráadásul egy slot. Húú! Akkor már csak egy ugyanilyen paraméterlistájú signal-t kell írunk.

A slot prototípusa `void showMessage(const QString & message, int timeout = 0)`, azaz vár egy üzenetet, és ezen felül még arra is kíváncsi, hogy hány ezredmásodpercig kell kiírva lennie az üzenetünknek. Nos, ezek szerint ezt a két paramétert kell emittálnunk.

Elsőként a `greeter.h` állományban helyezzük el a `slots` sor alá az alábbi függvény-prototípust:

```
void answerGreeting(const QString &message, int timeout = 0);
```

Figyeljük meg, hogy a paraméterlistát úgy alkottuk meg, hogy egyszerűen lemásoltuk a slot paraméterlistáját.

Most pedig furcsa figyelmeztetésre ragadtatjuk magunkat: Óvakodjunk a signal megvalósításától! Nem, nem erről a konkrét signal-ról van szó. Inkább átfogalmazuk: *Soha, egyetlen signal-t se valósítsunk meg!* Komolyan.

A signal használata úgy történik, hogy kiadjuk az `emit` parancsot, mögé pedig a signal nevét és paramétereit. Esetünkben a slot-unk – újabban már mindössze két soros – törzsében a `QDebug()` függvényt használó sort cseréljük le az alábbival:

```
emit answerGreeting(greetings.value(greeting, „Én nem ismerem  
kendet."), 3000);
```

Már csak annyi a dolgunk, hogy a `MainWindow` konstruktorába az előző `connect` utasítás alá újabbat szúrunk be:

```
connect(greeter, SIGNAL(answerGreeting(QString,int)),  
        ui->statusBar, SLOT(showMessage(QString,int)));
```

Ha a programot futtatva, és a köszönések ötletgazdagságán merengve gondolataink közé befészkeljük magát a gyanú, hogy a signals and slots mechanizmus többek között arra is jó, hogy egy gyermekobjektumból szalonképes formában „írhassuk” a szülőobjektum, illetve egy másik gyerekobjektum tulajdonságait, hívassuk a tagfüggvényeiket – akkor legyünk büszkék magunkra!

Most mindenestre hátradőlhetünk, és vérmérsékletünk függvényében elfogyaszthatjuk a maradék sárgarépat, illetve odaadhatjuk a hörcsögnek.

4. AKINEK NINCS ESZE...

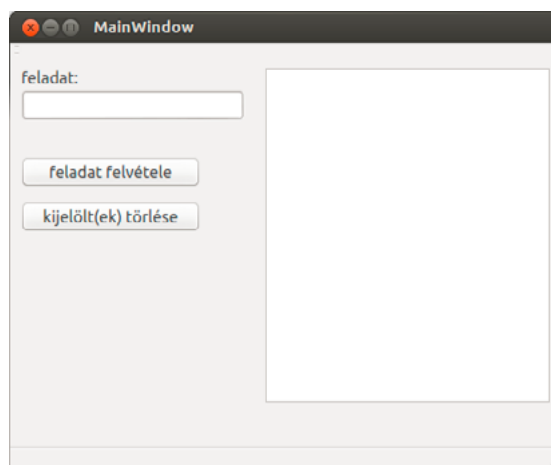
Ebben a fejezetben nekifogunk annak a műnek, melyre tulajdonképp egyáltalán nincs szükség, mert tonnaszám található ilyen már készen. Van ingyen és drágán, van helyben futó és internetes, van szép és csúnya. Szükség tehát nincs rá, cserébe viszont a könyv végéig ellesszünk vele. Tennivaló-listát nyilvántartó programot készítünk!

Miért pont ezt? Két jó okunk is van rá. Az egyik, hogy erre szépen felfűzhetjük minden leendő tudományunkat, még ha néha erőltetetté is válik a dolog, a másik pedig, hogy ez olyan alkalmazás, amellyel mindenki tisztában van: tudja, hogy mit várhat tőle.

Kezdjünk új projektet az előzőhöz hasonló módon. A projekt neve legyen – szárnyalj, fantázia! – ToDoList.

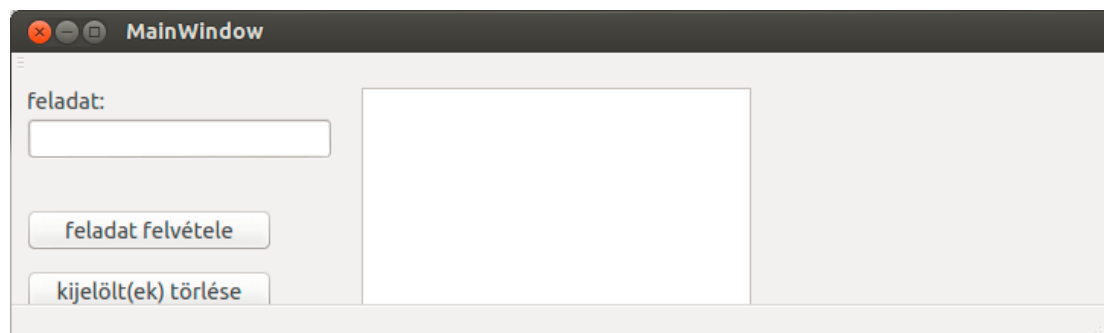
4.1. A QListWidget színre lép, és megismerkedünk az elrendezések fontosságával

Gyalogoljunk át a `mainwindow.ui` állományra. Megnyílik a grafikus szerkesztő. Helyezzünk el egy `QLineEdit` elemet és két `QPushButton`-t. A szerkesztő neve maradjon az alapértelmezett `lineEdit`, a két nyomógombé pedig `addButton` és `removeButton`. Tegyük ki még egy `QListWidget`-et is, a neve maradjon az alapértelmezett `listWidget`. Végül még egy `QLabel` is kell. Helyezzük el és feliratozzuk őket az ábrán látható módon:



3. ábra: a ToDoList első futása

Ha most egy kicsit átméretezzük az ablakot, az eredmény esztétikai értéke erősen korlátozott lesz.



4. ábra: A ToDoList főablaka rosszul tűri az átméretezést

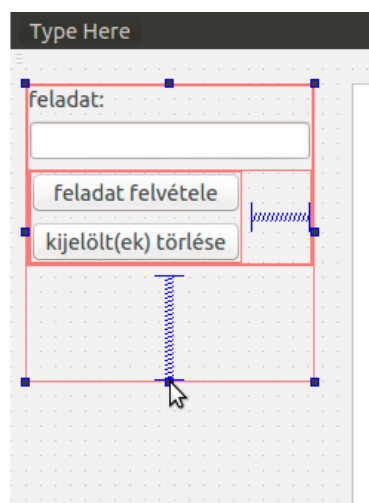
Próbáljunk segíteni a helyzeten. A megoldás az elrendezések (layout) használata, ha ugyanis efféleképp oldjuk meg az elemek elhelyezését, akkor szépen mozognak és átméreteződnek majd az ablak méretének változtatásával. Az elrendezések ki-be kapcsolgatása, állítgatása programból is történhet, de mi inkább kattintgatunk most. Alapvetően megy a dolog, hogy kijelöljük azokat az elemeket (widget), amelyeket *egymáshoz képest* egymás mellé vagy fölé rendeznénk, és a Qt Creator felső részén, a **Edit Signals/Slots** (lásd korábbi ábránkat) ikon mellett megkeressük a **Layout**

Out Horizontally (vízszintes elrendezés) és a **Layout Vertically** (függőleges elrendezés) gombokat. Az elrendezési hierarchiát jobb oldalt, az **Object Inspector**-ban láthatjuk. Ha valamit úgy rendeztünk el, ahogy nem is annyira jó, akkor itt kattintsunk jobb egérgombbal, és a helyi menüben keressük meg a **Layout out... / Break Layout** (elrendezés megszüntetése) menüpontot. Az egészel kísérletezni kell egy darabig, amíg rá nem érünk. Most előbb az ábrán nézzük meg a jelenlegi elrendezést, aztán próbáljuk elvégezni a következő, lépésről lépésre vezető útmutatót.

Először jelöljük ki a két nyomógombot – egyszerre –, és rendezzük őket függőlegesen. Aztán az elemek közül húzzunk a jobb oldalukra egy **Horizontal Spacer**-t (vízszintes távtartó), és jelöljük ki az előző párost, meg a **Spacer**-t, majd rendezzük őket vízszintesen. Ha megvagyunk, akkor a kijelöléshez vegyük hozzá a **lineEdit**-et, meg a „feladat:” feliratot, és rendezzük őket függőlegesen. Mikor ez is kész, tegyünk alájuk egy **Vertical Spacer**-t, és ezzel együtt kijelölve válasszuk a függőleges elrendezést. Fogjuk meg az alsó fogantyút (lásd az ábrát), és húzzuk a balra lévő **QListWidget** aláig.

Object	Class
▼ MainWindow	QMainWindow
▼ centralWidget	QWidget
addButton	QPushButton
label	QLabel
lineEdit	QLineEdit
listWidget	QListWidget
removeButton	QPushButton
menuBar	QMenuBar
mainToolBar	QToolBar
statusBar	QStatusBar

5. ábra: A jelenlegi elrendezés az Object Inspector-ban



6. ábra: A fogantyút húzzuk a QListWidget aláig

Vegyük hozzá a kijelöléshez a `QListWidget`-et, és rendezzünk vízszintesen (ha összeugrana, húzzuk ki újra). Ezek után kattintsunk a kijelölés *mellé*, a `MainWindow` területére, és kattintsunk akár a vízszintes, akár a függőleges elrendezésre. Az **Object Inspector** ablaka ekkor az ábrán látható hierarchiát mutatja.

Most futtatva a programot az egész hóbelevanc szépen újrendeződik és átméreteződik az ablak sarkának húzásával. Már csak annyi a baj a jó kis programunkkal, hogy nem jó még semmire, hiányzik belőle a lényeg. No, tegyük róla!

Object	Class
▼ MainWindow	QMainWindow
▼ centralWidget	QWidget
▼ horizontalLayout_2	QHBoxLayout
▼ verticalLayout_3	QVBoxLayout
▼ verticalLayout_2	QVBoxLayout
▼ horizontalLayout	QHBoxLayout
▼ verticalLayout	QVBoxLayout
addButton	QPushButton
removeButton	QPushButton
horizontalSpacer	Spacer
label	QLabel
lineEdit	QLineEdit
verticalSpacer	Spacer
listWidget	QListWidget
menuBar	QMenuBar
mainToolBar	QToolBar
statusBar	QStatusBar

7. ábra: A kész elrendezés hierarchiája az Object Inspector-ban

4.2. A `QListWidget` elemeinek manipulációja

A hozzáadás megvalósításával kezdjük. Kattintsunk jobb egérgombbal a felső nyomógombon, és válasszuk a **Go to slot** menüpontot a helyi menüből. Megnyílik a kódszerkesztőben a függvényünk, és már írhatjuk is a törzsét.

```
void MainWindow::on_addButton_clicked()
{
    ui->listWidget->addItem(ui->lineEdit->text());
}
```

A `QListWidget` osztályú objektumokban az `addItem()` tagfüggvénnyel helyezünk el újakat. A nyitó zárójel kiírásakor (ha az automatikus kiegészítést használtuk, akkor az Enter lenyomásakor) eszközsúgóként megjelenik a paraméterlista, aminek az elején az „1 of 2” azt jelzi, hogy van másik is. A lefelé mutató kurzorral meg is tudjuk nézni: a másik paraméterlista (remek dolog a túlterhelés, igaz?) egyetlen `QString`-ből áll. Ilyen meg ugye van nekünk, benne a `lineEdit` objektumban. Kinyerjük a `text` tagfüggvénnyel⁷, és kész. Vagy nem?

A program működik, de elég buta: felveszi a `listWidget`-re az üres feladatokat is. Gondolhatnánk arra, hogy megnézzük előtte, hogy üres-e a `lineEdit.text()` visszatérési értéke, de a szóközt még így is felveszi. A `text` tagfüggvény ugyebár `QString` típusú objektumot ad vissza. A `QString` dokumentációját böngészve meg találunk egy jó kis `simplified()` tagfüggvényt, ami azon felül, hogy a sor elejéről és végéről radiózza a szóközt, tabulátort, soremelést és hasonlókat (angol összefoglaló nevükön: whitespace), még a sor közben is csak egyet-egyet hagy meg olyan helyeken, ahova többet is sikerült elhelyeznie az embernek. A függvényünk törzsét alakítsuk át így:

7 Azt még nem is említettük, hogy az objektum mutatójának neve után nem kell nyílat (`->`) gépelnünk, elég egy pont is, mintha nem is mutató volna, hanem igazi objektum. A Qt Creator intelligensen cseréli, ha ennek szükségét látja.

```
QString txt = ui->lineEdit->text().simplified();
if(!txt.isEmpty())
    ui->listWidget->addItem(txt);
```

Azaz mostanra csak az a feladat kerül be a `listWidget`-be, ami nem üres. Mit lehet még tupírozni? Például azt, hogy egyúttal ürüljön ki a szerkesztősor. Ezzel gyorsan megvagyunk, írjunk még egy sort a fenti három alá:

```
ui->lineEdit->clear();
```

És ha már ilyen szépen kiürítjük, vissza is adhatnánk neki a fókuszt. Íme a függvénytörzs ötödik sora:

```
ui->lineEdit->setFocus();
```

Persze az se volna épp rossz, ha nem kellene kattintani, hanem elég volna Enter-t nyomni, amikor megfogalmaztuk valamely bokros teendőnket. A `QLineEdit` osztály dokumentációjában hamar rálelünk a `returnPressed()` signal-ra, amihez írhatnánk egy, a fentivel megegyező slot-függvényt, de az ugye a kód karbantarthatóságának nem tesz jót. A valós tennivalókat elkülöníthetnénk külön függvénybe, és megoldhatnánk, hogy mindkét slot ezt a függvényt hívja, egyszerű burkoló (wrapper) szerepre kárhóztatva a jó kis slot-jainkat. De még az aranyhörcsögünk is csalódottabban rágcsálná az almáját, ha a Qt-ban nem volna valamilyen elegánsabb megoldása a problémának. És van is: mégpedig az, hogy a `connect` utasítás nem csak egy signal-t és egy slot-ot tud összekötni, hanem két signal-t is. A `MainWindow` konstruktorának törzsében jelenleg egyetlen sor szerepel, ez alá írjuk be, hogy:

```
connect(ui->lineEdit, SIGNAL(returnPressed()),
        ui->addButton, SIGNAL(clicked()));
```

Szabad fordításban: ha a `lineEdit`-ben Enter-t nyomnak, csináld azt, mintha rákattintottak volna az `addButton`-ra – bármi legyen is a teendő.

És mi a helyzet, ha valaki a gyorsbillentyűk (keyboard shortcut) nagy rajongója? Kedvezzünk neki is! Kattintsunk az `addButton`-ra, és a `Property Editor`-ban a `text` tulajdonság értékét változtassuk meg: írjunk az első „f” betű elé egy & jelet (angolul „ampersand”, ha keresni akarunk az interneten – mondjuk máskor is „ampersand”-nak hívják). Az f betű mostantól alá van húzva, és az `ALT+F` billentyűkombinációval is tudunk „kattintani” a gombon. Ez remekül ment, nyilván nem okoz gondot a másik gomb hasonló beállítása. De hogy navigálunk vissza a `lineEdit`-re? Persze megadhatnánk neki is billentyűkombinációt, de miként oldjuk meg azt, hogy a felhasználó előtt mindez ne maradjon titokban? Hova írjuk ki neki?

A megoldás az lesz, hogy egy `QLabel` osztályú elemet, egy címkét használunk – nálunk ilyen a `label` –, ennek a feliratában helyezzük el az & jelet. Persze ne az „f” betű elé tegyük, bármelyik másik viszont jó – legyen mondjuk az „e”. A címkék statikus jellemükből kifolyólag nem sok mindent tudnak csinálni, még arra is képtelenek, hogy az `Alt+E` billentyűkombináció lenyomása után a fókuszt elfogadják. Azonban igen önzetlen természetűek, és örömmel adják oda az így kapott fókuszt a pajtásuknak. Már csak meg kell mondanunk nekik, hogy melyik elem a pajtásuk.

A pajtás angolul „buddy”, és ennek tudatában sejteni kezdjük, hogy a `QLabel` osztály dokumentációjában lévő `setBuddy()` tagfüggvény mire lehet jó. Mi azonban most nem programból oldjuk meg a buddy beállítását. Két lehetőségünk is van: az egyik, hogy az `Edit Signals/Slots` és az elrendezések ikonjainak közelében megkeressük az `Edit Buddies` ikont, és az

egérrel nyilat húzunk a `label` és a `lineEdit` objektum közé. A másik, hogy a `Property Editor`-ban adjuk meg a `buddy` tulajdonság értékéül szolgáló objektum nevét. A programot futtatva a billentyűkombináció hatására valóban a `lineEdit` kapja meg a fókuszt.

Akkor már csak a törlés gomb munkára fogása van hátra. Ez persze rejt magában néhány további érdekességet, úgyhogy amennyiben a hörcsög már álmos, küldjük aludni. Mi meg még maradunk.

A szokásos módon állítsuk elő a törlés gomb lenyomására reagáló slot-függvényt. Aztán ugorjunk neki a `QListWidget` osztály dokumentációjának. Hamar rájövünk, hogy alighanem a `takeItem()` tagfüggvény lesz a barátunk. Látjuk, hogy a visszatérési értéke `QListWidgetItem` osztályú objektumra mutató mutató, és a neve is zavaró. Az a helyzet, hogy ez a tagfüggvény nem törli az objektumot (a listaelemet), hanem csak *kiveszi a listából*. A bemeneti értéke az objektum sorának a száma. Ha a slot törzsébe egyelőre csak annyit helyezünk el, hogy

```
qDebug() << ui->listWidget->currentRow();
```

akkor a gomb lenyomásakor látjuk, hogy a `listWidget` sorai 0-tól számozódnak (ne felejtjük el használatba venni a `<QDebug>` fejlécfájlt).

Ha most ezt a sort így egészítjük ki:

```
qDebug() << ui->listWidget->takeItem(ui->listWidget->currentRow());
```

akkor a `listWidget`-ből eltűnik a sor, és a `qDebug()` függvény kinyomtatja a kivett `QListWidgetItem` osztályú objektumra mutató címet. Ha ezt összevetjük a `takeItem()` dokumentációjában olvasott „Items removed from a list widget will not be managed by Qt, and will need to be deleted manually.” (a `listWidget`-ből eltávolított listaelemeket a Qt nem kezeli, kézzel kell őket törölni) kitétellel, akkor sejteni kezdjük, hogy van még dolgunk. De menjünk biztosra, írjuk át a slot-unk törzsét efféleképp:

```
QListWidgetItem *lwi = ui->listWidget->takeItem(ui->listWidget->currentRow());
QDebug() << lwi;
ui->listWidget->addItem(lwi);
```

Indítsuk el a programunkat, vegyünk fel két-három listaelemet, majd az elsőt töröljük. Egy pillanatra eltűnik (nem is látjuk), a `qDebug()` kiírja a címét, majd visszakerül – a lista végére. Kell ennél jobb bizonyíték arra, hogy nem is töröltük, csak *kivettük*?

Ha viszont ez a helyzet, akkor egyértelmű, hogy mi legyen a függvény törzsé:

```
delete ui->listWidget->takeItem(ui->listWidget->currentRow());
```

és kész.

Vagyis dehogy: ezzel az eljárással csak egy elemet tudunk törölni egyszerre. Az igaz, hogy egyelőre csak egy listaelemet tudunk egyszerre kijelölni, azaz ezen kéne segítenünk először. Talán nem lepődünk meg, ha kiderül, hogy a `listWidget` tulajdonságain kell állítanunk, egészen konkrétan a `selectionMode` értékéül kell `MultiSelection`-t megadni. A többszörös kijelölés megy, de a gomb lenyomására csak az egyik kijelölt elem törlődik. Hmmm.

A `QListWidget` osztály dokumentációjában keresgélve felleljük a `selectedItems()` tagfüggvényt, s ha már így esett, írhatunk egy remek ciklust. A `selectedItems()` visszatérési értéke egy `QListWidgetItem` osztályú mutatókat tartalmazó `QList`, ezt kell bejárjunk.

Üritsük ki a slot-unk törzsét: kezdjük előlről. A `QList` osztályú objektumokat úgy tudjuk deklarálni, hogy megadjuk a lista elemeinek típusát is, ezek ezúttal ugye `QListWidgetItem` osztályú objektumokra mutató mutatók lesznek. Az objektumunkat rögtön fel is töltjük. A fenti két műveletet egy sorban végezzük el:

```
QList<QListWidgetItem*> toBeDeleted = ui->listWidget->selectedItems();
```

A lista egy egyszerű for-ciklussal bejárható, és minden egyes tagra hívható a `takeItem()` tagfüggvény, ami persze az elem sorának a számát várja bemenetként, nekünk olyanunk meg nincsen. Szerencsére a `QListWidget` osztályban megvalósították a `row()` tagfüggvényt, ami sorszámot ad vissza, ha megadjuk a `QListWidgetItem` osztályú objektum mutatóját, és mutatóink történetesen vannak is, benne a `toBeDeleted` listában. Azaz a slot törzsének maradék sorai a következők:

```
for(int i = 0; i < toBeDeleted.size(); i++)
    delete ui->listWidget->takeItem(ui->listWidget->row(
toBeDeleted.at(i)));
```

Mindez persze nem olyan elegáns, viszont jól átlátható. A következő megoldás nem olyan egyértelmű, de bizonyos szemszögből szebb mint az előző. Ebben a megoldásban a slot törzse egyetlen sor:

```
qDeleteAll(ui->listWidget->selectedItems());
```

A `qDeleteAll()` egy nem kifejezetten agyondokumentált függvény, de kis keresgéléssel kiderül, hogy valójában mutatókat töröl. Tekintve, hogy a `selectedItems()` tagfüggvény a kijelölt elemek `QListWidgetItem*` típusú mutatóinak listáját adja vissza, a `qDeleteAll()` függvény remek szolgálatot tesz nekünk.

Levezetésül még állítsuk be a tab-sorrendet, azaz azt, hogy a grafikus elemek között milyen sorrendben ugrálunk végig a Tab billentyű nyomogatásával. Az ikon már megint az [Edit Signals/Slots](#) mellett van, keressük meg, és pár kattintással állítsuk be a megfelelő sorrendet.

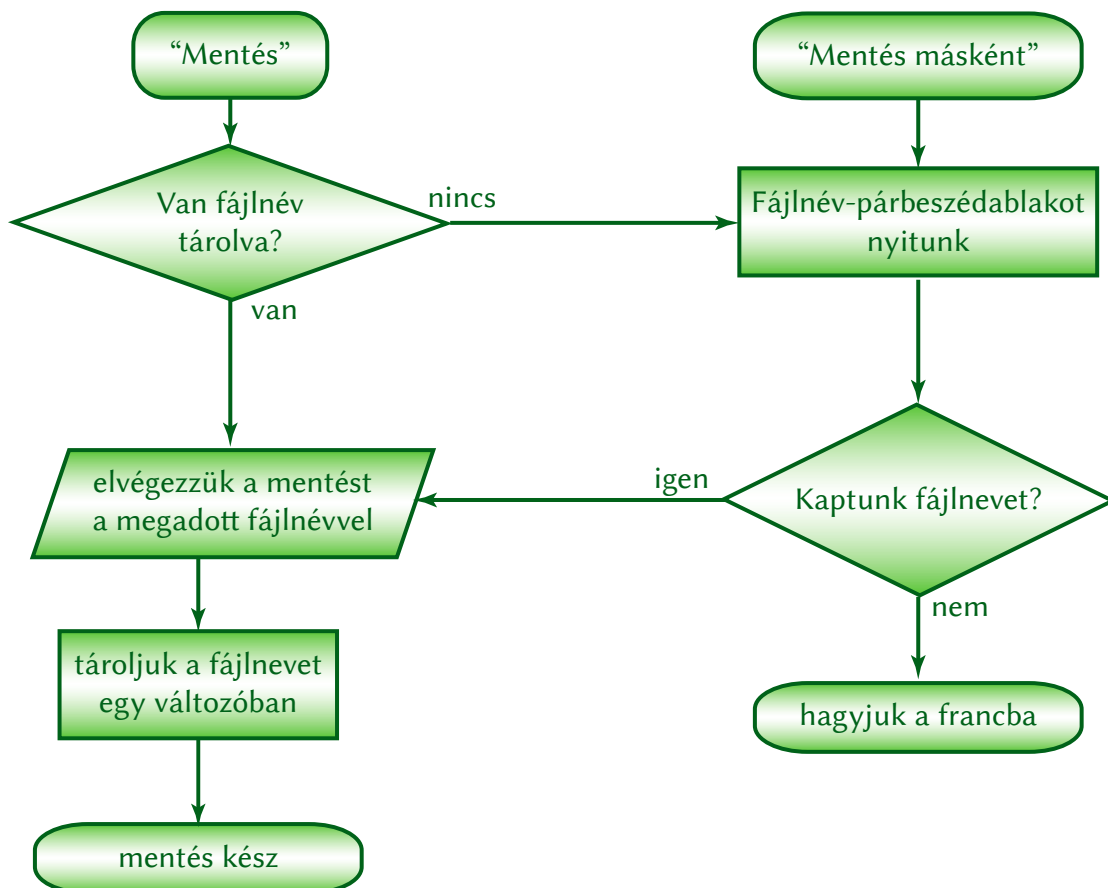
Ezzel a `ToDoList` első változata kész is. Remélhetőleg a hörcsögünk mostanra kialudta magát, és a motoszkálásával nem hagy bennünket aludni. Pedig ránk fér, nemdebár?

5. MENTSÜK, AMI MENTHETŐ!

Már a múltkor is motoszkált a fejünkben valami halvány gyanú, hogy ugyan mi értelme az olyan listának, amit minden alkalommal újra kell írni, de akkor el voltunk foglalva holmi pajtásokkal, tab-sorrendekkel, meg mutatólisták törlésével. Most azonban semmi sem állhatja útját annak, hogy teendőinket megörökítsük. Kőbe vésésről persze szó sincs, megelégszünk azzal is, ha operációs rendszerünk háttértárán elhelyezhetjük őket egy fájlban.

A mentési műveleteknek ezer éve két menüpont a kiindulása: a „Mentés” és a „Mentés másként...”. A két menüpont működése szorosan összefügg, és nem is triviális, hogy pontosan mely pontokon. Úgyhogy bevetjük azt az eszközt, amit az első programozás-óráink óta nem használtunk: folyamatábrát készítünk. Voilà:

Ezt a sok műveletet (és a betöltésről még nem is beszéltünk) jó volna elkülöníteni, úgyhogy külön osztályt valósítunk meg nekik. Úgy kezdünk hozzá, hogy a fájl-fában a **Sources** részre vagy a projekt nevére kattintunk jobb egérgombbal. A helyi menüből az **Add New...** pontot választjuk, majd a párbeszédablakban a **C++**, azon belül a **C++ Class** lehetőséget adjuk meg. Az osztály neve legyen **FileOperator**, az alaposztálya pedig a **QWidget**⁸.



8. ábra: A mentési folyamat

8 Majdnem gond nélkül lehetne QObject is, de egy esetben kelleni fog neki a QWidget őse. A QWidget annyival több egy sima QObject-nél, hogy elmondani is nehéz: ez az osztály őse minden olyan osztálynak, amelyekből a grafikus felhasználói felület összeáll.

És, még a komoly munka megkezdése előtt használatba is vesszük az osztályt: a `mainwindow.h` fájlban megadjuk a fejlécek között a `fileoperator.h`-t, és deklarálunk az objektumunknak egy privát mutatót `fileOperator` néven. A `MainWindow` objektum konstruktorában pedig létrehozuk az objektumot, nem feledve magát a `MainWindow`-t (`this`) megadni szülőobjektumként.

5.1. QStringList a feladatok átadására

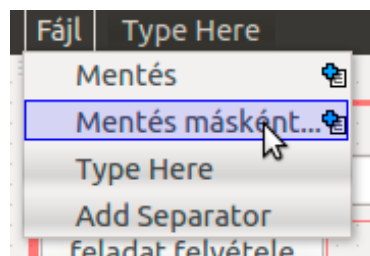
A feladatok jelenleg a `listWidget` belsejében laknak, méghozzá úgy, hogy a `QListWidgetItem` típusú objektumok `text()` tagfüggvénye tudja őket visszaadni. Deklaráljunk hát egy `QStringList` visszatérési értékű, `createThingsToDoList()` nevű privát függvényt, és írjuk meg a törzsét is. Ne felejtkezzünk meg a `<QStringList>` fejléc használatáról.

```
QStringList MainWindow::createThingsToDoList()
{
    QStringList list;
    for(int i = 0; i < ui->listWidget->count(); i++)
        list << ui->listWidget->item(i)->text();
    return list;
}
```

5.2. Menük és QAction-ök

Aztán, mielőtt a `FileOperator` osztályt hivatalni kezdenénk, elkészítjük a menüt. Nyissuk meg a grafikus szerkesztőben a `mainwindow.ui` fájlt, és az ablakon a `Type Here` (gépelj ide) részre kattintva alakítsuk ki az ábrán látható menüt.

A menü neve és a menüpontok megjelennek az **Object Inspector** hierarchiájában. Kezdjük azzal, hogy átnevezzük őket: az objektumok ugyanis nem kaphatnak ékezetes karaktereket tartalmazó nevet, és az ékezetes karaktereket a Qt Creator alávonással (underscore) helyettesíti, ami távol áll a számunkra optimálisról. Közben vegyük észre, hogy a menüben belül a „Mentés” és a „Mentés másként...” osztály `QAction` lett.



9. ábra: A Fájl menü

Ha az átkeresztelgetésekkel elkészültünk, a főablak rajza alatti **Action Editor** részen a „Mentés” kaphat egy `Ctrl+S` billentyűkombinációt. Ugyanitt tudunk a két „action” számára slot-ot létrehozni a helyi menüből az immáron megszokott `Go to slot...` menüpont választásával. A párbeszédablakból kiválasztható események közül mindkét esetben a `triggered()` használatát javasoljuk. S ha sikeresen létrejött a két slot (ha akarjuk, letesztelhetjük a működésüket egy-egy `qDebug()`-üzenettel), visszatérhetünk a `FileOperator` osztály megvalósításához.

5.3. A FileOperator osztályunk – QFileDialog, QMessageBox és egyéb veszedelmek

Mire is lesz szükségünk? Kelleni fog egy `QString` osztályú privát fájlnev – ötletesen a `fileName` nevet választjuk. Kelleni fog egy `save()` és egy `saveAs()` tagfüggvény – mindkettő

publikus, ezek lesznek a folyamatábrán is látható belépési pontok. Mindkettő függvény paramétere a feladatokat tároló `QStringList` osztályú objektum lesz. Mindkét osztály használatát jeleznünk kell a fejlécek között a `fileoperator.h` fájlban. Kelleni fog továbbá egy `performSaveOperation()` privát tagfüggvény, amelyiket a folyamatábrán a rombusz és az alatta lévő téglalap jelképez. Ez a függvény végzi majd a munka dandárját, és két helyről is hívhatjuk, de csak az egyik esetben (mentés) van tárolt fájlnevünk, a másik esetben (mentés másként) még nincs, vagy nem azt akarjuk használni, így innen jó volna átadni neki. Úgy érezzük, hogy túlzás volna túlterhelni ezt a tagfüggvényt, azaz abban maradunk, hogy inkább mindkét esetben átadjuk majd neki a fájlnevet, és ennek öröme fel is tüntetjük a paraméterlistájában. A változókat és a tagfüggvényeket helyezzük el a `fileoperator.h` állományban, aztán kezdjük hozzá a `save()` tagfüggvény megvalósításához.

A `save()` törzse teljes négy sor (figyeljük meg, hogy a folyamatábrától egy hangyányit eltérünk, amennyiben fájlnev hiányában a másik belépési pontra irányítjuk át a műveletet – persze a belépési pont és a párbeszédablak megnyitása a valóságban egybecsúszik, szóval nem is biztos, hogy eltértünk):

```
void FileOperator::save(QString thingsToDo)
{
    if(fileName.isEmpty())
        saveAs(thingsToDo);
    else
        performSaveOperation(fileName, thingsToDo);
}
```

Akkor hát a `saveAs()` tagfüggvénnyel folytatjuk a mókát. Szükségünk lesz a `QFileDialog` osztály `getSaveFileName()` statikus tagfüggvényére, azaz helyezzük el a `<QFileDialog>`-ot a betöltendő fejlécek között.

Folytassuk a `performSaveOperation()` tagfüggvény megvalósításával, de egyelőre csak egy olyan változattal, ami csak kiírja, hogy hova mentene, és beállítja a `fileName` változó új értékét. A teljes függvény így néz ki:

```
void FileOperator::performSaveOperation(QString fn, QStringList list)
{
    qDebug() << "Saving to:" << fn;
    fileName = fn;
}
```

Ha zavar bennünket, hogy a fordítás során figyelmeztetést kapunk a nem használt `list` változó miatt, akkor a függvénytörzs első soraként adjuk meg, hogy:

```
Q_UNUSED(list)
```

A makrónak a tesztelési időszakon túl akkor van nagy értelme, amikor egy beépített függvény, ha kell, ha nem átad nekünk valamit, amit kénytelenek vagyunk el is venni, de egyébként nem igazán van rá szükségünk.

Mostanra már a hörcsög szerint is csak a `saveAs()` tagfüggvény megvalósítása lehet hátra. Használni szeretnénk pár fejlécfájlt: a `<QDir>`, a `<QFileInfo>`, és a `<QFileDialog>` fejlécről van szó.

A `QFileDialog` osztályból egy statikus függvényt, a `getSaveFileName()` nevűt használjuk arra, hogy megjelenítsük a célfájl kiválasztására szolgáló ablakot. A függvény

paraméterlistájának első tagja a szülőobjektum, aminek `QWidget` osztályúnak kell lennie – hát ezért döntöttünk annak idején úgy, hogy a `FileOperator` osztályunk őse a `QWidget` és nem a `QObject`.

A függvény további paraméterei között szerepel az a könyvtár is, ahol a párbeszédablak megnyílik. Igen ám, de hol is nyíljon meg? Hát, ha ez az első mentés, akkor legyen mondjuk a felhasználó saját mappája, ha meg már volt, akkor ott, ahova az utolsó fájlt mentettük. A saját mappa elérési útja platformfüggetlenül megtudható a `QDir` osztály statikus tagfüggvényének használatával. Az utolsó fájl teljes elérési útjából pedig mindenféle karakterlánc-varázslattal ki tudjuk nyerni a tartalmazó mappát, de most lusták vagyunk, és különben is a `QFileInfo` osztály épp erre jó. Azaz elsőként végezzünk az elérési út beállításával:

```
QString path = QDir::homePath();
if(fileName != ""){
    QFileInfo info1(fileName);
    path = info1.absolutePath();
}
```

Oké, eddig megvagyunk. Megnyithatjuk a párbeszédablakot, aminek a visszatérési értéke a teljes elérési utat is tartalmazó fájlnev. A példában szándékosan helyezünk el minden paramétert új sorban, mert annyi van belőlük, hogy az több a soknál – és még nem is adtuk meg mindet. Természetesen csak azért használunk többféle lehetséges kiterjesztést, mert ez egy strandkönyv, és minden strandkönyvben így szokás, meg kicsit azért is, mert így jobban be tudjuk mutatni a `QFileDialog::getSaveFileName()` tagfüggvény lehetőségeit. A paraméterek sorban: szülőobjektum, ablak címe, az a mappa, ahol az ablak megnyílik, és a fájl típusok. Utóbbiakat két pontosvesszővel választjuk el.

```
QString fn = QFileDialog::getSaveFileName(
    this,
    "Mentés másként...",
    path,
    "ToDoList-fájlok (*.tdolst);;Szövegfájlok (*.txt);;Minden fájl (*)"
);
```

Ha a felhasználó a mégse-gombra kattint, vagy ezzel egyenértékűen cselekszik, a függvény üres karakterlánccal tér vissza. Így a következő lépésünk annak ellenőrzése, hogy akkor végső soron van-e fájlnevünk. Ha nincs fájlnev, akkor egyszerűen befejezzük a függvényt és visszatérünk a főablakhoz. Ha van fájlnev, akkor megnézzük, hogy van-e kiterjesztése⁹ – megint egy `QFileInfo` osztályú objektumot használunk a feladat kivitelezésére. Ha nem volna, akkor odabiggyesztünk egyet. Mikor mindezzel megvagyunk, meghívjuk a félkész `performSaveOperation()` tagfüggvényt.

9 Logikusnak tűnhet, hogy a beállított fájl típusnak megfelelő kiterjesztés automatikusan kerüljön rá a fájl végére. A Qt párbeszédablaka itt nem segít, aminek okait főként a Qt multiplatform mivoltában kell keresnünk. Van operációs rendszer, ahol tulajdonképp a beépített mentési ablak nyílik meg, de van, ahol nincs ilyen, és ott a Qt adja az ablakot.

```

if(!fn.isEmpty()){
    QFileInfo info2(fn);
    QString ext = info2.suffix();
    if(ext != ".tdolst" && ext != ".txt")
        fn += ".tdolst";
    performSaveOperation(fn, thingsToDo);
}

```

A `saveAs()` tagfüggvény ezzel kész. Visszakutyagolhatunk a `mainwindow.cpp` fájlba, és a két slot-ban megadhatjuk a megfelelő függvény hívását:

```

void MainWindow::on_actionMentes_triggered()
{
    fileOperator->save(createThingsToDoList());
}

```

```

void MainWindow::on_actionMentesMaskent_triggered()
{
    fileOperator->saveAs(createThingsToDoList());
}

```

Ki is próbálhatjuk őket. Persze valós mentés nem fog történni, de figyelhetünk a megfelelő fájlnev képzésére, megjelenítésére, a második-harmadik-sokadik mentéskor a kétféle mentési mód megfelelő viselkedésére. Mennie kell.

A hörcsög pointere az asztalunkon lévő kekszekre mutat – vagy akkor ez most a keksz pointere? Még szerencse, hogy nem kutyánk van. Mondjuk egy pointer.

A `performSaveOperation()` tagfüggvény rendes megvalósítása irányába tereljük csapongó gondolatainkat. A feladatokat nyilván úgy volna érdemes mentenünk, hogy soronként egy feladatot mentünk, így a fájl más szoftverből is kezelhető marad. A fájlok írása-olvasása Qt-ban a `QFile` osztály használatával történik – így aztán szükségünk lesz a hasonló nevű fejlécfájlra.

A fájl megnyitásakor beállíthatunk néhány jelzőbitet (angolul flag), amelyeket bitenkénti vagy művelettel kötünk össze. Az így előálló szám pontos reprezentációja lesz annak, hogy mely kapcsolókat, jelzőbitekét adtuk meg. Persze a jelzőbiteknek megvan a szöveges konstans változatuk is, így nem kell fejben tartanunk, hogy az írásra megnyitás például a `0x0002`, az meg, hogy szövegfájlként nyissuk meg, és a soremelések az operációs rendszernek megfelelően konvertálódjanak (soremelés Unix-okon, soremelés és kocsivissza Windows-on), a `0x0010` jelzőbit. Ezeket a jelzőbitekét bitenkénti vaggal összekapcsolva `0x0012`-őt kellene megadnunk a következő példában. Szerencsére nem szorulunk arra, hogy a jelzőbitekét folyamatosan fejben tartsuk, erőforrásainkat nem kell kódolásukra és dekódolásukra pazarolnunk; látjuk majd, hogy szép olvasható formában is kifejezhetjük kívánságainkat. Még hozzátesszük, hogy azért értekeztünk ám ilyen hosszan a módszerről, mert a Qt előszeretettel használja több osztályában is.

Az adatok soronkénti kiírását könnyíti meg számunkra a `QTextStream` osztály. Ne felejtkezzünk el a fejlécéről.

Már csak egy dolgot kell átgondolnunk, mielőtt nekikezdenénk a `performSaveOperation()` tagfüggvényhez, nevezetesen azt, hogy figyelünk kéne rá, hogy végül sikerül-e a mentés. Gondolnánk-e vagy sem, a Qt legtöbb modulja nem kezel kivételeket, azaz aki arra gondolt, hogy most aztán jöhetnek a jó kis try-catch blokkok, bizony ki kell ábrándítanunk. És hogy miért nem kezeli a Qt a kivételeket? Azért, mert annyira multiplatformos. Használták olyan platformon is, ahol a platform számára ismeretlen volt a kivétel fogalma, a kivételkezelés

mechanizmusa – így kivételkezelést csak pár újabb modultól várhat az ember. Mi pedig maradunk a jó öreg változóállítgatós módszernél, de azért szólunk a felhasználónak, ha baj történt. Mondanivalónk közlésére a `QMessageBox` osztályból példányosítunk objektumot – persze csak a fejlécének használatba vételét követően.

No, akkor hát:

```
void FileOperator::performSaveOperation(QString fn, QStringList list)
{
    QFile file(fn);
    bool success = false;
    if (file.open(QFile::WriteOnly | QFile::Truncate | QFile::Text)) {
        QTextStream out(&file);
        for(int i = 0; i < list.count(); i++)
            out << list.at(i) << endl;
        if(file.error() == 0)
            success = true;
    }
    if(success)
        fileName = fn;
    else{
        QMessageBox mb;
        mb.setIcon(QMessageBox::Critical);
        mb.setText("Nem sikerült a mentés.");
        mb.setInformativeText("Próbálgj valami okosat tenni.");
        mb.exec();
    }
}
```

Minekutána létrejön a fájlnev felhasználásával a `file` objektum, és a sikeres műveletet jelző változónak beállítottuk a kezdeti hamis értéket, megpróbálkozunk a fájl írásra való megnyitásával. Ha sikerült, akkor egy `for`-ciklussal végigjárjuk a `QStringList` osztályú listát, és az egyes sorokat a fájlba írjuk. Az írással a program akkor is próbálkozik, ha időközben mondjuk elfogy a hely, vagy egyéb szörnyűség történik, de ez esetben a `file` objektum `FileError`-száma nullától eltérő lesz. A lehetséges hibakódokat a `QIODevice` osztály dokumentációja tartalmazza.

Ha a művelet sikeres, akkor beállítjuk a `fileName` változó értékét, ha nem, akkor megjelenítünk egy üzenetet arról, hogy pórul jártunk. A fájlt nem szükséges lezárunk, ugyanis a `QFile` osztály destruktora elvégzi ezt a műveletet, márpedig ahogy a hatókörből kilépünk, a stack-ben létrehozott objektumok destruktora meghívódik.

Akinek kedve van, megírhatja, hogy a program a főablak állapotosorán jelezze a fájlműveletek sikerét, illetve sikertelenségét – neki az „A QMap és a köszönések” című részben áttekintettek lesznek nagy segítségére. Meg kell adnia a signal-t, ami a sikerről tájékoztat, a `performSaveOperation()` tagfüggvényben el kell helyezni a megfelelő `emit`-parancsokat, illetve ki kell alakítani a kapcsolatot az állapotosor slot-ja és a `fileOperator` objektum signal-ja között.

Hasonló signal-slot mechanizmus kialakításával megoldható az is, hogy a főablak címsorába kiíródjék az épp használatban lévő fájl neve.

A mentéssel emígyen végeztünk is. Ha eddig nem adtuk oda a hörcsögnek a kekszet, most már igazán megkaphatja. Mi pedig folytassuk a betöltéssel, amit – követve az elmúlt húsz év divatját – megnyitásnak fogunk nevezni.

Alakítsuk ki a helyét a menüben, rendeljük hozzá a szokásos billentyűkombinációt, és állítsassuk elő a slot-ot, ami majd kezeli az eseményt.

Mit is kell tennie ennek a slot-nak? Először is törölnie kell a `listWidget` sorait. A következő lépés a fájl betöltése, és a benne lévő sorok megszerzése. A sorokat alighanem a `fileOperator` objektumtól fogjuk elkérni, megint csak `QStringList` osztályú objektum formájában. Persze, ha a betöltés sikertelen, a felhasználó meggondolja magát, vagy hasonló szörnyűség történik, esetleg örülnénk, ha mégsem törlöttük volna ki a jelenlegi listát. Érdeemes lesz tehát ezt a két feladatot felcserélnünk. Az utolsó teendő pedig a megszerzett feladatlista betöltése a `listWidget`-be.

Kezdjük a munkát a `FileOperator` osztály legújabb tagfüggvényeivel. Az `open()` függvény feladata lesz a kívánt fájlnev kiderítése, és ennek ismeretében a `performLoadOperation()` tagfüggvény hívása. Ez utóbbi `QStringList` osztályú objektumban adja vissza a fájlból beolvasott sorokat, amit aztán az `open()` szépen továbbad az ő hívójának. Az `open()` tagfüggvény egyébiránt igen hasonló lesz a `saveAs()` nevű társához:

```
QStringList FileOperator::open()
{
    QString path = QDir::homePath();
    if(fileName != ""){
        QFileInfo info(fileName);
        path = info.absolutePath();
    }
    QString fn = QFileDialog::getOpenFileName(
        this,
        "Fájl megnyitása...",
        path,
        "ToDoList-fájlok (*.tdolst);;Szövegfájlok (*.txt);;Minden fájl (*)"
    );
    QStringList thingsToDo;
    if(!fn.isEmpty())
        thingsToDo = performLoadOperation(fn);
    return thingsToDo;
}
```

Ezúttal is abból indulunk ki, hogy amennyiben már van betöltött fájlunk, feltételezzük, hogy a mostani megnyitásnál ugyanabból a mappából szándékszik a felhasználó egy másik fájlt megnyitni, mondjuk azért, mert az előző mégsem az volt, amit keresett. Ezúttal természetesen az erre a feladatra készült statikus tagfüggvényt használjuk a `QFileDialog` osztályból, aminek a paraméterezésében az előzőekhez képest semmi különbséget nem találunk. Ha kaptunk fájlnevet, megkíséreljük a fájl betöltését, de azt már másik tagfüggvényre bizzuk. Mindenképp visszaadjuk a frissen létrehozott `thingsToDo` objektumot, adott esetben akár üresen is.

A valós munkát végző `performLoadOperation()` tagfüggvény is sok ponton hasonlít a mentéskor használt párjára:

```

QStringList FileOperator::performLoadOperation(QString fn)
{
    QStringList thingsToDo;
    QFile file(fn);
    bool success = false;
    if (file.open(QIODevice::ReadOnly | QIODevice::Text)){
        QTextStream in(&file);
        while (!in.atEnd())
            thingsToDo << in.readLine();
        if(file.error() == 0)
            success = true;
    }
    if(success)
        fileName = fn;
    else{
        QMessageBox mb;
        mb.setIcon(QMessageBox::Critical);
        mb.setText("Nem sikerült a megnyitás.");
        mb.setInformativeText("Próbálj valami okosat tenni.");
        mb.exec();
    }
    return thingsToDo;
}

```

A fájl megnyitását ezúttal olvasásra és szövegfájlként végezzük – ez utóbbi hangsúlyozása megint a fájlban lévő sorvégek operációs rendszerenkénti más-más jelzése miatt jó ötlet. Az előző esethez hasonlóan figyelünk arra is, hogy megnyitható-e a fájl és arra is, hogy a beolvasás rendben lezajlott-e. Amennyiben igen, beállítjuk a `fileName` változót, hogy a módosítások mentése egyszerűen megoldható legyen a **Ctrl+S** billentyűkombináció lenyomásával. Ha meg valami gikszer volt, a felhasználóra hagyományozzuk a probléma elhárításának feladatát. Ebből a tagfüggvényből is ész nélkül visszaadjuk az üres feladatlistát is – majd kezeljük a helyzetet a főablak slot-jában.

Bár pár bekezdéssel előbb megbeszéltük, hogy a Megnyitás menüpont kiválasztásakor viszonylag sokrétűek az elvégzendő feladatok, maga a slot valószínűtlenül egyszerű:

```

void MainWindow::on_actionMegnyitas_triggered()
{
    QStringList thingsToDo = fileOperator->open();
    if(!thingsToDo.isEmpty()){
        ui->listWidget->clear();
        ui->listWidget->addItem(thingsToDo);
    }
}

```

A törzs első sora deklarálja és inicializálja a feladatlistát tároló karakterlánc-listát. Többször említettük már, hogy esetleg üres listánk lesz, így a további lépések végrehajtása előtt ellenőrizzük, hogy valóban így jártunk-e. Ha igen, akkor a tétlenség bölcs útját választjuk.

Ha azonban van listánk, egy elegáns hívással kitakarítjuk a `listWidget` objektumot. Utána pedig épp ciklusírásba fognánk, a ciklusmagban `addItem()` tagfüggvény-hívásokkal, mikor a `QListWidget` osztály dokumentációjában ráakadunk az `addItem()` tagfüggvényre, amely történetesen épp `QStringList` osztályú objektumot vár bemeneti értéként. Mint a mesében!

Menteni és megnyitni remekül tudunk, de mi a helyzet, ha új listát írnánk? Hozzunk létre neki menüpontot, állítsunk be billentyűkombinációt, majd keressük meg a **Go to slot...** lehetőséget. Eddig biztos. Mi a teendők még? Hát először is a **listWidget** kiürítése. Ez viszonylag gyorsan megvan: a slot törzsében helyezzük el a

```
while(ui->listWidget->count()>0)
{
    delete ui->listWidget->takeItem(0);
}
```

blokkot. A felületes szemlélő azt gondolná, hogy el is készültünk a feladatunkkal, de – szokás szerint – más a helyzet. Ha ugyanis történt már fájlművelet a program indítása óta, akkor a **fileName** változó értéke nem üres karakterlánc, ami azt eredményezi, hogy a Mentés menüpont választásával úgy felülírjuk a fájlunkat, hogy csak na. Persze a **fileName** változó a **fileOperator** objektumunkban lapul, és ráadásul privát, ami jól is van így. Készítsük el a **FileOperator::newList()** tagfüggvényt, aminek a törzse egyetlen sor lesz majd:

```
fileName = „”;
```

Ha elkészültünk, a fenti slot törzsének utolsó utasításaként adjuk ki a tagfüggvényt hívó

```
fileOperator->newList();
```

utasítást.

Némi tesztelgetés után végre felkelünk egy kicsit, s elégedetten einstandot¹⁰ jelentünk be az aranyhörcsög maradék kekszeire. Aztán visszajövünk még pár percre, utána meg úgy is vége a fejzetnek.

5.4. Iterátor à la STL és à la Java – na és a foreach

Az imént örvendeztünk azon, hogy milyen remek dolog is egy hirtelen jött **addItem()** tagfüggvény. Ugyan hová is lettünk volna nélküle? Hát lássuk, méghozzá négyszer. Az alábbiakban ugyanis képesek leszünk négyféleképp is végigjárni a **thingsToDo** nevű, **QStringList** osztályú objektumot. A közölt kódrészlet mind a négy esetben az **on_actionMegnyitas_triggered()** slot törzsének utolsó, az **addItem()** függvényt hívó sorát cseréli le.

Az első változat egyszerű **for**-ciklus:

```
for(int i = 0; i < thingsToDo.size(); i++)
    ui->listWidget->addItem(thingsToDo.at(i));
```

A **thingsToDo.at(i)** kifejezés majdnem egyenértékű a **thingsToDo[i]** változattal. Értékadásra azonban csak az utóbbi használható.

A második változat úgynevezett STL¹¹-féle iterátorral dolgozik:

```
QStringList::const_iterator ci;
for (ci = thingsToDo.constBegin(); ci != thingsToDo.constEnd(); ++ci)
    ui->listWidget->addItem(*ci);
```

A harmadik változat a Qt 4-ben megjelent Java-féle iterátor segítségével járja be a

¹⁰ <http://hu.wikipedia.org/wiki/Einstand>

¹¹ Az STL a C++ Standard Template Library (szabványos sablonkönyvtár) név rövidítése

thingsToDo objektumot:

```
QStringListIterator ji(thingsToDo);  
while (ji.hasNext())  
    ui->listWidget->addItem(ji.next());
```

Figyeljük meg, hogy egyrészt új osztályból példányosítunk objektumot, s így jutunk iterátorhoz, (ugye nem felejtkezünk el a fejlécről?), másrészt, hogy ez az iterátor az előzőtől eltérően nem egy adott elemre mutat, hanem *elemek közé*. Így már van értelme annak, hogy a következő (next) elemből képezzünk elemet a listWidget számára.

Az STL és a Java típusú iterátorok előnyeiről, hátrányairól igazán remek összefoglaló található a <http://doc.qt.digia.com/qq/qq12-qt4-iterators.html> weboldalon. Kár volna kihagyni.

Végül, de utolsósorban említjük meg a több nyelvből is ismert **foreach** kulcsszót. Qt-ban a következőképp használható:

```
foreach(QString thing, thingsToDo)  
    ui->listWidget->addItem(thing);
```

Látjuk, hogy ez a módszer azoknak való, akik szeretik a dolgokat a nevükön nevezni. A *teendők* listájából minden körben – a ciklus minden ismétlődésekor – kiveszünk egy konkrét *teendőt*, és azzal ügködünk valamit. Nem indexszel, nem hivatkozva, nem egy lista következő elemeként aposztrofálva, hanem pusztán a neve alapján gondolhatunk rá. Ez a módszer úgy segíti gondolkodásunkat, hogy kiiktat a belőle egy absztrakciós szintet – ami azért néha nagyon jól jön.

Alighanem még a hörcsögnek is feltűnt, hogy az eredeti megoldásnál most mind a négy módszer többet problémázik, de ez kivételes eset. A legritkább esetben lesz olyan mázlink, hogy megússzuk egy gyűjteményes adattípus elemeinek bejárását.

6. A FELADATOKHOZ KATEGÓRIÁKAT RENDELÜNK

Időközben annyi teendőnk lett, hogy nem győzzük őket áttekinteni, és az a fényes ötletünk támadt, hogy mindegyiket besoroljuk valamilyen kategóriába. A programnak igyekszünk meg hagyni az eddigi szerkezetét, azaz azt, hogy egy osztály kezeli a fájlt, és a főablak dolgoz a tartalom átvétele és megjelenítése. Ezentúl azonban nem `QStringList` osztályú objektumot mozgattunk a két programrészünk között, ugyanis a feladatok már nem `QString`-ek. Létrehozunk egy új osztályt, és ebből példányosítjuk azokat az objektumokat, amelyek az egyes teendőket és a hozzájuk tartozó kategóriát tárolják.

Állapodjunk meg abban, hogy mostantól egy feladat (task) egy teendőből (job) és a hozzá tartozó kategóriából (category) áll.

6.1. A Task osztály

A Task osztályt egy teljesen új projektben írjuk meg, mert utána még lesz vele kis bajunk, és ezzel jobb, ha az eddigi kódunktól leválasztva ismerkedünk meg és leszünk úrrá rajta. Úgyhogy kezdjük új projektet a szokásos módon. Hozzunk létre egy új C++ osztályt Task néven. Ősosztályként adjuk meg a `QObject`-et.

Kis Qt-stilisztika következik: Qt-ban úgy szokás osztályokat megvalósítani, hogy a tagváltozó neve elé valamilyen előtétet teszünk, és amit a tagváltozó nevéül szánunk, az valójában a kiolvasó (getter), azaz a tagváltozó értékét visszaadó függvény lesz. Az előtétek lehetnek arra utalók, hogy ez egy tagváltozó, tehát jó előtét például az `"m_"`. A beállító, értékadó (setter) függvények neve pedig a `set` szóból és a tagváltozó nevének nagybetűs változatából áll.

Akkor ezt most élesben. A Task osztálynak két privát tagváltozója lesz, az `"m_job"` és az `"m_category"`, mindkettő `QString` típusú. Lesz két kiolvasó függvénye, a `job()` és a `category()`. Mindkettő visszatérési értéke `QString` típusú, és egyetlen sorból álló törzsük a hasonló nevű változó értékét adja vissza. Lesz két beállító függvénye is az osztálynak, a `void` visszatérési értékű `setJob()` és a `setCategory()`. Mindkettő paramétere `QString`, és egyetlen értékadó utasítás szerepel bennük, amely a hasonló nevű tagváltozó értékéül a paraméterként kapott karakterláncot adja meg. Természetesen mind a négy tagfüggvény publikus.

A `mainwindow.h` fájl elején, a fejlécek között helyezzük el a `task.h` állományt.

6.2. A `QList<T>` sablonosztály, és némi örület a mutatókkal. Megismerkedünk a `QSharedPointer` osztállyal

A `QList<T>` tehát egy sablonosztály. Sablonosztály? Az meg mi a nyavalya? Valami olyasmi, amelyben megadhatjuk, hogy a létrehozott példány milyen típusú (igen, ez a nagy T) adatokat kezeljen. A `QList<T>` tehát olyan osztály, amelynek példányai T típusú objektumokat tárolnak.

A `mainwindow.h` fájlban adjuk meg a `<QList>` fejléctet, majd deklaráljuk a privát `tasks` listát. Így:

```
QList<Task> tasks;
```

Adjunk meg továbbá egy `createTasks()` tagfüggvényt is. A megvalósítása a következő:

```

QStringList jobs = (QStringList() << "job1" << "job2" << "job3");
QStringList categories = (QStringList() << "cat1" << "cat2" <<
"cat3");

for(int i = 0; i < jobs.size(); i++){
    Task t;
    t.setJob(jobs.at(i));
    t.setCategory(categories.at(i));
    tasks.append(t);
}

```

A blokk elején ékes példáját látjuk annak, ahogy az ember `QStringList` típusú objektumot állít elő és tölt fel (sőt, értékül is ad) egyetlen sorban. A recept a következő: végy egy üres `QStringList`-et, adj át neki `QString`-eket a `<<` operátorral, és tedd zárójelbe az egész hóbelevancot, hogy a műveletek az értékadó műveleti jel munkába lépése előtt végbe is menjenek.

A maradék részben, azaz a ciklusban `Task` típusú objektumokat hozunk létre. Beléjük pakoljuk a megfelelő teendőket és kategóriákat (eszünkbe jut, hogy jó volna egy paraméterezhető túlterhelt konstruktort is írni az osztálynak), majd a `QList<Task>` osztályú `tasks` objektumhoz fűzzük őket.

Akkor most fordítsuk le.

Brr! Hogyaszongya: error: 'QObject::QObject(const QObject&)' is private. Míg a legtöbb esetben csak a szemünket dörzsöljük ilyen remek hibaüzenet láttán, a szemfülesek így kiáltanak fel: „Ojjé, hiszen ez egy privát másolókonstruktor! Ezek szerint a `QObject` osztály leszármazottai nem másolhatók!”

Ördögük van. No de korábban keljen fel az, aki rajtunk (így) akar kifogni. Deklaráljuk úgy a `QList`-et, hogy ne `Task` osztályú objektumokat, hanem a rájuk mutató mutatókat tudja tárolni:

```

QList<Task*> tasks;

```

És persze a remek tagfüggvényünket is átírjuk, legalábbis a ciklus belsejében lévő első három sort.

```

Task *t = new Task();
t->setJob(jobs.at(i));
t->setCategory(categories.at(i));

```

A `t` immáron mutatóvá avanszált (vagy degradálódott: nézőpont kérdése), ennek megfelelően a beállító tagfüggvényeket már nem ponttal, hanem nyíllal kötjük. És végre lefordul az osztályunk!

A hörcsög extázisban rágja a dióbelet. Először még mi is, de aztán kellemetlen gyanú fészkei magát kicsi szívünkbe. Mi lesz a sok objektumunkkal, ha a `QList` osztályú `tasks` objektum elhalálozik, akármiért is? Gondolkodjunk: törlődik egy rakás mutató. És maguk az objektumok? De nem ám! Na *ezt* hívják memóriaszivárgásnak.

Többek között ilyen esetekre találták ki a smart pointereket (okos mutatókat). A smart pointer¹² attól olyan smart, hogy automatikus forrásfelszabadítást végez, azaz ha már sehol

12 A Qt smart pointereiről remek összefoglalás olvasható itt <http://blog.qt.digia.com/blog/2009/08/25/count-with-me-how-many-smart-pointer-classes-does-qt-have/> és itt <http://www.macieira.org/blog/2012/07/continue-using-qpointer/>

nem hivatkoznak az objektumra, akkor megsemmisíti azt. A Qt sok smart pointere közül mi a `QSharedPointer` osztályt használjuk fel.

A `mainwindow.h` állományban adjuk meg a `<QSharedPointer>` fejléct is, a `tasks` deklarációját pedig alakítsuk át ilyenképpen:

```
QList<QSharedPointer<Task> > tasks;
```

Ne felejtjük ott a csillagot, és figyeljünk arra is, hogy a két záró `>` karakter között legyen szóköz.

A `mainwindow.cpp` fájlban a `Task *t = new Task();` sort pedig váltsuk fel ezzel:

```
QSharedPointer<Task> t(new Task());
```

Nincs egyenlőségjel.

Az így létrehozott mutatót úgy használjuk a továbbiakban, mintha a világ legközönségesebb mutatója volna. A hivatkozott objektum tagfüggvényeit is a legközönségesebb -> nyilakkal hívjuk a pont helyett, így a tagfüggvény maradék része nem is változik.

A nehezén immár túl vagyunk, gyorsan ellenőrizzük le, hogy tényleg használható lesz-e így az osztályunk. Kezdjük azzal, hogy megírjuk a túlterhelt, paramétrezhető konstruktort, amit az imént hiányoltunk:

```
Task::Task(QString job, QString category, QObject *parent) :  
    QObject(parent)  
{  
    m_job = job;  
    m_category = category;  
}
```

Aztán ezt használatba is vesszük, átírjuk a `createTasks()` tagfüggvény törzsét ilyenre:

```
QSharedPointer<Task> t(new Task(jobs.at(i), categories.at(i)));  
tasks.append(t);
```

A főablakra húzzunk ki egy `QPushButton`-t és két `QListWidget`-et. Hagyjuk meg az eredeti neveiket. A szokásos kattintgatásokkal állítsuk elő a nyomógomb slot-ját, és benne alakítsuk ki az alábbi törzset:

```
createTasks();  
for(int i = 0; i < tasks.size(); i++){  
    ui->listWidget->addItem(tasks.at(i)->job());  
    ui->listWidget_2->addItem(tasks.at(i)->category());  
}
```

Vegyük észre azt, amit elvileg persze tudunk, nevezetesen, hogy a `tasks.at(i)` mutatót ad vissza, minek folyományaként a `Task` osztály tagfüggvényeit a -> operátorral hívjuk.

Tanulmányprogramocskánk ezzel elkészült (a forrása természetesen ennek a programnak is elérhető a könyv weboldalán), visszatérhetünk a `ToDoList`-hez.

6.3. A `ToDoList`-et felkészítjük a kategóriák használatára, de még nem használunk kategóriákat

Most, hogy elolvastuk ezt a remek címet, és kicsit összezavarodtunk, talán világossá válik minden, ha megtudjuk, hogy a konkrét kategóriák helyett egyelőre egy helyőrzőt adunk csak meg. Azzal kezdjük a munkát, hogy a `ToDoList` projekt mappájába átmásoljuk a `task.cpp` és

a `task.h` állományt. A projekt betöltését követően a fájlválasztó rész **Sources** feliratán jobb egérgombot nyomva az **Add Existing Files** lehetőséget választva a projekt részévé tesszük őket. A `task.h` állományt a `<QList>` és a `<QSharedPointer>` fejlécekkel együtt felvesszük mind a `fileoperator.h`, mind a `mainwindow.h` állomány elejére. Alakítsuk továbbá vissza a feladattörölést úgy, ahogy először elkészítettük: hogy csak egyetlen elemet töröl egyszerre a gomb. A gomb feliratának átalakítása a legkisebb, bár a leglátványosabb változás. A hozzá tartozó slot törzse legyen ismét:

```
delete ui->listWidget->takeItem(ui->listWidget->currentRow());
```

Természetesen a `listWidget` tulajdonságai közül a `selectionMode`-ot is vissza kell állítanunk `SingleSelection`-ra a tulajdonságszerkesztőben.

És akkor most átülünk a hörcsög mellé és miközben ő abba a tévedésbe esve, hogy kaja áll a házhöz, előjön, elfilóztatunk kicsit a programunkon. Az adatok eddig egyszerű karakterláncok voltak, de ez már a múlté, és így a múlté válik az is, hogy a főablakon belül többnyire a `listWidget` tárolta őket, és csak közvetlen azelőtt emeltük át őket egy látható megjelenési formával nem rendelkező objektumba – a `QStringList` osztályú `thingsToDo`-ba –, hogy átadtuk volna az egészet a `fileOperator` objektumnak mentés végett.

Most azonban ez nem járható út: az előző tanulmányprogramunkhoz hasonlóan önálló objektumra van szükségünk az adatok tárolására. Ugyanezt az objektumot kell majd átadnunk a `fileOperator` objektumnak mentés végett, ráadásul megnyitáskor is ilyen objektumot kapunk majd vissza tőle.

Otthagytva a hörcsögöt kezdjük a munkát azzal, hogy a `mainwindow.h` állományban privátként deklaráljuk azt a változót, amelyik majd a feladatokat tárolja. Puskázhatunk az előző programocskából, de akár innen is átírhatjuk:

```
QList<QSharedPointer<Task>> > tasks;
```

A `createTasks()` tagfüggvény mehet a kukába, a deklaráció csakúgy, mint a megvalósítás. Az `addButton` kivételével az összes slot törzsét alakítsuk megjegyzéssé.

Lássuk, mit alakítunk az `addButton` slot-ján! Úgy döntünk, hogy a szerkesztősor tartalmát két helyre tesszük. Az egyik – mint eddig – a `listWidget` lesz, a másik pedig maga a `tasks` lista. Ez alá a sor alá helyezzük el az a két sort, amelyik a listához is hozzáfűzi a feladatot:

```
QSharedPointer<Task> t(new Task(txt, "default category"));
tasks.append(t);
```

Az első sor ugyebár előállít egy `Task` osztályú objektumra mutató mutatót. Ha még emlékszünk, írunk egy túlterhelt konstruktort is, így a szerkesztőléc tartalmát (a `txt` változó) és a kategória helyőrzőjét már az objektum (`t`) létrehozásakor megadjuk. Az objektum mutatóját pedig utolsó elemként hozzábiggyesztjük a `tasks` listához. (Ugye nem fogjuk elkövetni azt a kezdő hibát, hogy a `{}` kapcsolósárójel-párat elfelejtjük a ciklusmag köré kitenni?)

Ha kipróbáljuk művünket, és eddig minden oké, akkor lássunk neki a törlésnek. Vagy mégis inkább a törlés megvalósításának. A `QList` osztály dokumentációjában hamar megtalálja az ember a `takeAt()` tagfüggvényt, ami a `listWidget` `takeItem()` függvénye miatt ismerősnek tűnik, és valóban, ugyanúgy kiveszi a listából az indexnek megfelelő valamit, visszaadva a hívónak. Nekünk ez a valamint egy mutató, azaz a `delete` utasítással törölhetjük is a mutató jelezte objektumot. Ha azonban immáron a Qt-dokumentáció gyakorlott olvasójává fejlődünk,

már rég észrevettük a `takeAt()` ismertetése alatt is megbúvó „lásd továbbá” (See also) részt, benne pedig a `removeAt()` tagfüggvényt. Ez lesz a mi barátunk!

A `QList::removeAt()` tagfüggvény ugyanazt a számot várja paraméterül, mint a `listWidget::takeItem()`, legalábbis a mi esetünkben. Így a `removeButton` slot-jának törzse a következő formát ölti:

```
int index = ui->listWidget->currentRow();
delete ui->listWidget->takeItem(index);
tasks.removeAt(index);
```

A Mentés, illetve a Mentés másként menüpontoknak megfelelő slot-on csak annyi változtatni valónk van, hogy a függvényhívások paramétere mostantól a `tasks` lista.

A Megnyitás menüpont slot-ja azonban ennél egy fél fokkal bonyolultabb lesz. Először is, mostantól a `fileOperator` objektum `open()` tagfüggvénye olyan objektumot ad vissza, amit a `tasks` objektumunk értékül kaphat. Bár ezt a hangyányi változást a `FileOperator` osztály még nem tudja. Meg van még pár dolog, amit tán nem tud... na, majd mindjárt felvilágosítjuk!

A következő változás, hogy sajnos le kell mondanunk arról az `addItem()` tagfüggvényről, amin akkorát lelkendeztünk pár oldalnak előtte. Marad helyette a mezei `addItem()`, amit csak a példa kedvéért használtunk az iterátorokkal való ismerkedésünkkel. S ha már iterátor, legyen mondjuk Java-féle. Ne feledjük, hogy az iterátor sem `Task` objektumot, hanem `Task*` mutatót (egészen pontosan `QSharedPointer<Task>` típusú objektumot) ad vissza, azaz a tagfüggvényeket most sem a pont operátor használatával tudjuk hívni. A slot törzse az alábbi:

```
tasks = fileOperator->open();
if(!tasks.isEmpty()){
    ui->listWidget->clear();
    QListIterator<QSharedPointer<Task> > ji(tasks);
    while(ji.hasNext())
        ui->listWidget->addItem(ji.next()->job());
}
```

A főablakkal végeztünk, irány a `FileOperator` osztály. Az első dolgunk, hogy a fejléccálományában ahol eddig a tagfüggvények paramétere, illetőleg visszatérési értéke `QStringList` osztályú objektum volt, ott most állítsunk be `QList<QSharedPointer<Task> >` típust. Rádadásul talán van értelme felszámolni a még mindig egyszerű felsorolásra utaló `thingsToDo` nevet, jó lesz helyette a lényegesen laposabb, de ezúttal mégis elfogadhatóbbnak tűnő `list` név. Az egyes sorok átírásakor megjelenő kis villanyégő arra figyelmeztet, hogy el ne felejtjük a signature-ökon is változtatni, amit egyébiránt a Qt Creator az **Alt+Enter** billentyűkombináció lenyomásakor fel is ajánl. Ha mindent átírtunk, amit kellett, váltsunk át a **fileoperator.cpp** szerkesztésre (**F4**, emlékszünk még)?

Itt is át kell írunk a `thingsToDo`-kat `list`-ekre – a deklarációknál persze a típust is megfelelően módosítanunk kell. Tekintve az imént bemutatott **Alt+Enter**-nyomkodás eredményét, a `save()`, a `saveAs()` és az `open()` tagfüggvényen más változtatnivaló nincs is. Hja, kérem, a gondos tervezés!...

A `performSaveOperation()` tagfüggvény esetében a fentiek már nem mernénk kijelenteni – eltekintve a tervezés minőségét firtató megállapítástól, természetesen. Szerencsére a cserébe nem fogunk belerokkanni, amennyiben mindössze az a feladat vár ránk, hogy az „`out << list.at(i) << endl;`” sor helyett az

```
out << list.at(i)->job() << "|" << list.at(i)->category() << endl;
```

sort helyezzük el. Figyeljük meg, hogy a fájlunkban egy sor továbbra is egy feladatot tárol, de annak két részét – a teendőt és a kategóriát – egy cső („|”, magyar billentyűzetkiosztáson **AltGr+W**) karakter választja el egymástól. Azért pont ez a karakter, mert a hétköznapi életben elég ritkán használjuk mondanivalónk megfogalmazására.

A `performLoadOperation()` tagfüggvénybe ismét bele kell tûrni pár sornyt. A `while`-ciklus magját kell kissé megváltoztatnunk, sőt, felhizlalnunk:

```
QStringList sl = in.readLine().split("|");
QSharedPointer<Task> t(new Task(sl.at(0), sl.at(1)));
tasks.append(t);
```

Az első utasítás a beolvasott sort a cső karakternél kettészedi, az eredményt egy karakterlánc-listába töltve. A második sor ebből a karakterlánc-listából létrehozza a `Task` osztályú objektumot, s annak mutatóját, nagyon hasonlóan ahhoz, amikor a szerkesztősorból és a „default category” helyőrzőből alakítottuk ki. A harmadik sor pedig, ahogy már rámutattunk pár bekezdéssel korábban: biggyeszt.

Már csak az új listát kezdő slot maradt hátra. A változtatása teljes egy sor lesz, lévén annyi uge a változás, hogy az adatszerkezetet már nem a `listWidget` tárolja, így a lista ürítésén túl a `tasks` objektum listáját is ki kell pucolnunk. Íme:

```
tasks.clear();
```

Egyszer menne a billentyűzettakarítás ilyen gyorsan!

Az a nagy helyzet, hogy kész vagyunk. Futtassuk a programunkat és örvendezzünk. A külsín nem sokat javult – ha nagyon ragaszkodunk a tényekhez, visszafejlődésről kell beszélnünk, hiszen a többszörös kijelölés és törlés eltűnt. Akinek nagyon hiányzik, a `QListWidget::selectedItems()` tagfüggvény használata mellett lehetősége nyílik az újbóli megvalósítására. Szóval, a külsín nem javult, na de a belbecs! A programunk immáron készen áll a kategóriák bevezetésére.

6.4. Kategóriák költöznek a ToDoList-be

A kategóriákról eddig még keveset értekeztünk. A `Task` osztály megvalósításából talán már kiderül, hogy egy teendő legfeljebb egy kategóriával bírhat. A kategória megadása megint csak `QLineEdit` osztályú szerkesztősorban fog történni, azaz a teendő megadására szolgáló sor `lineEdit` elnevezése zavaróvá válik. Jobb lenne, ha ez az „edit” job lenne. Tehát `jobEdit`.

Az átnevezést két lépésben ejtjük meg. Az elsőben a grafikus felhasználófelület-szerkesztőn kijelöljük az elemet, és akár a jobb egérgomb lenyomására megjelenő helyi menü megfelelő pontját választva, akár a **Property Editor**-ban megadjuk az új nevet. A második lépés pedig az, hogy a kódszerkesztőben a kurzort a `lineEdit` valamely előfordulásához helyezzük, majd a menüből a **Tools - C + + - Rename Symbol Under Cursor** (a kurzor alatti szimbólum átnevezése) lehetőséget választjuk. Ekkor a Qt Creator megmondja, hogy hol és mennyi előfordulását fedezte fel a `lineEdit`-nek, mi meg megadjuk az új nevet (`jobEdit`), majd a **Replace** (csere) gombbal végre is hajtatjuk a műveletet. Futtassuk a programunkat – a sikeres fordítás és némi tesztelés bizonyossá teszi, hogy minden szándékainknak megfelelően zajlott le.

Folytassuk azzal a munkát, hogy elhelyezzük a kategória szerkesztősorát. Viszonylag könnyen

a helyére talál az elrendezések rengetegében, elvileg nem lesz vele gondunk. Sőt, alighanem a tab-sorrenden sem kell módosítanunk, valószínűleg az is azonnal jó lesz. A szerkesztősor-objektum neve legyen `categoryEdit`. Fölé helyezzünk el egy címkét (`QLabel` osztályú objektumot), rajta a szöveg nem meglepő módon kategória lesz, sőt: &kategória, hogy az **Alt+K** billentyűkombináció működhessen. S ha már itt tartunk, nem felejtkezünk el a kispajtás (`buddy`) megadásáról sem.

Már csak az fájdtja a szívünket, hogy a kategória megadását követően hiába nyomunk **Enter**-t, az új feladat nem vétetik fel. Hogy is oldottuk meg ezt a problémát a `jobEdit` szerkesztősor esetében (bár akkor a `jobEdit` még lánykori nevén, `lineEdit` néven szerepelt)? Egy jó kis `connect` utasítással a főablak konstruktorában, amellyel a szerkesztősor `returnPressed()` signal-ját az `addButton` objektum `clicked()` signal-jához kapcsoltuk. Akkor ezek szerint hasonló ármányt kell elkövetnünk a `categoryEdit` esetében is? Így van. Ne feledjük az `on_addButton_clicked()` slot-ot úgy átírni, hogy a kategória szerkesztősora is törlődjön.

Akkor kész is? Dehogyan, hiszen továbbra is minden kategória „default category” lesz: nem vesszük figyelembe, hogy drága egyetlen felhasználónk (aki egyébiránt mi magunk vagyunk) mit írt a kategóriaszerkesztőbe. Mondjuk nem ettől fognak elkopni a kézi-üzleteink: annyi mindössze a dolgunk, hogy az imént emlegetett `on_addButton_clicked()` slot `QSharedPointer<Task> t(new Task(txt, "default category"))`; sorát a következőre cseréljük:

```
QSharedPointer<Task> t(new Task(txt, ui->categoryEdit->text().simplified()));
```

Persze arról, hogy a módosításunknak haszna is van, csak úgy tudunk meggyőződni, ha a mentett fájlba belekukkantunk. Tegyük meg, és örvendezzünk!

Így, a nap végén, megmutatjuk a hörcsögünknek – aki épp karalábét majszol a tenyerünkben, s ekképp nem igazán ér rá odafigyelni –, hogy mekkora jó programot írtunk. Azonban menet közben ráébredünk, hogy a „megetetni a papagályt” az mégis a másik j, mert így nagyon bután néz ki a szó. És akkor most törölni kell a feladatot és létrehozni egy újat, mert a meglévő feladatok módosítására képtelen a program? Nos, ma még megtesszük, de legközelebb úgy átírjuk a programot, hogy csak na!

7. MÓDOSÍTHATÓ FELADATOK, FELAJÁNLOTT KATEGÓRIÁK ÉS ALAPÉRTELMEZETTEN BETÖLTÖTT FELADATLISTA-FÁJL

Ha már a múltkor ilyen csúnyán elragadtattuk magunkat a szárnyasunk miatt, fogjunk hozzá a feladatok módosíthatóságának megvalósításához. Minekutána ezzel megvagyunk, akkor olyat varázsolunk, hogy a kategóriaszerkesztő a már meglévő kategóriákat felajánlja a kategória gépelése közben. Amikor már ez is megvan, akkor az maradt, hogy tároljuk az utoljára mentett fájl nevét, és a fájlt automatikusan betöltetjük induláskor.

7.1. A már megadott feladatok módosítása

Valahogy úgy képzeljük a dolgot, hogy ha rákattintunk egy feladatra, akkor annak a teendője és a kategóriája megjelenik a megfelelő szerkesztősorban. Mi úri kedvünknek megfelelően átírjuk akár az egyiket, akár a másikat, majd szerzői működésünk végeztével a kétsoros remekművet akár új feladatként, akár az előző helyére mentve, annak módosításaként visszahelyezzük a listába.

Természetesen a signals and slots mechanizmusra szeretnénk támaszkodni, így első közelítésben a `QListWidget` osztály dokumentációját böngésszük át használható signal után kutatva. Kis gondolkodás után – vagy épp anélkül, a szabadon maradó erőforrások függvényében – úgy döntünk, hogy a `currentRowChanged(int currentRow)` signal-t fogjuk használni. Jobb egérgombot kattintunk a Qt Creator felhasználói felület-tervezőjében a `listWidget`-en, és a *Go to slot...* lehetőséget választjuk, a felbukkanó listából pedig megkeressük a kinézett signal-t.

Az a vágyunk, hogy a kijelölt feladat teendője kerüljön a `jobEdit`, a kategóriája pedig a `categoryEdit` szerkesztőbe. A létrejött slot függvény törzsét a kódszerkesztőben a következőképp adjuk meg:

```
ui->jobEdit->setText(tasks.at(currentRow)->job());
ui->categoryEdit->setText(tasks.at(currentRow)->category());
```

Szépen kihasználjuk a signal-tól kapott `currentRow` paramétert. Teszteléskor jónak is tűnik minden, egészen addig, amíg az utolsó feladatot is ki nem töröljük, ugyanis ekkor a programunk elszáll, kezdeti jókedvünkkel együtt.

Na, akkor még a végén kénytelenek leszünk gondolkodni, pedig valaki már említette, hogy ilyen grafikus fejlesztőkkel azok írnak programot, akiknek nem a gondolkodás az erősségük.

Remélhetőleg nagyjából a következő gondolatmenetet járjuk be:

- valójában a signal nem akkor jön, amikor kijelölünk egy sort, hanem amikor az aktuális sor megváltozik – ez a két dolog nem pontosan ugyanaz
- amit kitörlünk az nem lehet aktuális
- törléskor az aktuális érték változik
- amikor az utolsó elemet töröljük, nem lehet valódi az aktuális érték (ha `qDebug()` -gal kiíratjuk, látjuk is, hogy a -1. elem lesz az aktuális)
- a mutatólista -1. elemének tagfüggvényét hívni az álmoskönyv szerint csupa rosszat jelent

Akkor most mi lesz? Az egyik lehetőség az, hogy egy feltételvizsgálattal vesszük körbe a fenti két utasítást: csak akkor történjenek meg, ha a `currentRow` értéke legalább 0. Így jó is a programunk, de a további tesztelgetés során felfedezünk még egy zavaró mellékhatást. Ugye mi az aktuális elem megváltozásához kötöttük a két szerkesztősor kitöltését. Márpedig törlést követően is változik az aktuális elem, ami azért baj, mert ilyenkor olyan elemek adatai is a szerkesztősorokba kerülnek, amelyekkel épp nem óhajtottunk foglalkozni.

Úgyhogy vissza a dokumentációhoz, ahol ezúttal arra az `itemClicked(QListWidgetItem *item)` signal-ra fanyalodunk rá, amelyről az imént épp azért siklott tovább tekintetünk, mert nem sort ad vissza, hanem elemet. A sor meg még ki kell nyernünk.

Nos, ha már így jártunk, megint állíttassuk elő a megfelelő slot-ot (a régi törlésekor a fejlécfájlból lévő deklarációról se felejtkezzünk el), és alakítsuk ilyenné:

```
void MainWindow::on_listWidget_itemClicked(QListWidgetItem *item)
{
    int row = ui->listWidget->row(item);
    ui->jobEdit->setText(tasks.at(row)->job());
    ui->categoryEdit->setText(tasks.at(row)->category());
}
```

No, ezzel már nem lesz baj, főleg amennyiben a `mainwindow.h` állományban megadjuk a `<QListWidgetItem>` fejléceket is. Ha a szöveget a szerkesztősorok valamelyikében módosítván úgy döntünk, hogy új feladatként kívánjuk rögzíteni változtatásaink eredményét, akkor semmi további programoznivalónk nincs. Amennyiben a jelenlegi feladatot módosítanánk, akkor viszont még akad egy s más.

Helyezzünk el egy módosítógombot `modifyButton()` néven. Állítsunk be neki gyorsbillentyűt, és állítsuk elő a slot függvényét.

Most, a sokadik slot előállításakor talán elfilóztatunk rajta, hogy amikor saját signal-t és slot-ot írtunk, össze kellett őket kapcsolni egy `connect` utasítással. Ilyenkor erre miért nincs szükség? Nos, azért nincs, mert a slot előállításakor olyan függvény jön létre, amelynek *neve* bizonyos szabályokat követ, nevezetesen:

- „on”-nal kezdődik, amit alávonás („_”) követ
- ezt követi a signal-t emittáló objektum neve, majd ismét alávonás
- végül a signal neve és a paraméterlista következik

Természetesen kézzel is írhatunk ilyen függvényt. Ha így járunk el, fordításkor automatikusan kialakul a kapcsolat, további teendők nincsenek.

Akkor vissza a slot-hoz.

Két feladatunk van: a változtatásokat bevezetni a `tasks` listába – a megfelelő listaelem jellemzőinek használatával –, illetőleg a teendő kiírása a `listWidget`-be. Mindezt azonban csak akkor érdemes elvégeznünk, ha a teendő nem lett üres – ha igen, akkor szóljunk a felhasználónak, hogy használja a törlés gombot, ha már egyszer megírtuk neki. A slot törzse az alábbi:

```

QString txt = ui->jobEdit->text().simplified();
if(!txt.isEmpty()){
    int row = ui->listWidget->currentRow();
    tasks[row]->setJob(txt);
    tasks[row]->setCategory(ui->categoryEdit->text());
    delete ui->listWidget->takeItem(row);
    ui->listWidget->insertItem(row, txt);
}else{
    QMessageBox mb;
    mb.setIcon(QMessageBox::Information);
    mb.setText("Feladatot nem módosítok üresre.");
    mb.setInformativeText("Ha tényleg ezt akarod, akkor töröld.");
    mb.exec();
}

```

Minthogy a teendő szerkesztősorának tartalmát több helyen használjuk, változóban tároljuk az értékét. A `listWidget` sorait, elemeit nem tudjuk módosítani. Így aztán a felhasználó elől elrejtve, sutyiban kivesszük és töröljük a kijelölt elemet – az eljárás már ismerős a törlés gomb slot-jából. Annak a sornak a számát, ahonnan az elemet kivettük, a `row` változóban tesszük el. Az egyetlen új elem az `insertItem()` tagfüggvény, amellyel az előzőleg törölt elem régi helyére beszúrjuk az új feladatot.

Hurrá, tudtunk feladatot módosítani! Így már sokkal könnyebben lesz a papagályból papagáj, igaz?

7.2. Kategóriák felajánlása a *QCompleter* osztály használatával.

Első pillantás a modellekre

Azt szeretnénk, ha a felhasználó nem adna meg fölöslegesen sok új kategóriát, mert ugye ha csak egy adott kategóriába tartozó feladatokat óhajtunk megjeleníteni (igen, ilyesmit is akarunk), akkor ugye nem jó, ha a hörcsögünkkel kapcsolatos feladataink egyszer hõri, másszor hörcsög, megint másszor aranyhörcsög kategóriába kerülnek. A sok kategória használatát megtiltani persze nincs értelme, de ha okosan felajánljuk a hasonló kategóriákat, akkor a felhasználó hajlamosabb lesz egy már létező kategóriát kiszemelni, mint újat szülni. Ez pedig végső soron a program használhatóságát növeli, igaz?

Első feladatunk egy mutatót deklarálni a *QCompleter* osztályú dokumentumunknak. Agyunk hosszas erőltetésével a **completer* nevet sikerül kiötlennünk. Megadjuk a `<QCompleter>` fejléct is. A főablak konstruktorában példányosíthatjuk is. Ha megnézzük a *QCompleter* osztály dokumentációját, látható, hogy halmozottan hátrányos, szakszóval többszörösen túlterhelt függvényről van szó. Mi ezúttal azt a verziót használjuk, amely *QStringList* osztályú objektumból ajánlja fel a választási lehetőségeket, és egyelőre statikus listát adunk meg. Az objektum példányosítását követően azonnal be is állítjuk a `categoryEdit` objektumot úgy, hogy vegye használatba legújabb játékszerünket:

```

completer = new QCompleter(QStringList() << "hörcsög" <<
"papagáj" << "számítógép"), this);
ui->categoryEdit->setCompleter(completer);

```

Hát, ez remekül megy, igaz? Már csak azt kell megoldanunk, hogy a lista a valódi kategóriákat ismerje. A feladat két ponton kínál kihívásokat.

Az egyik az, hogy mikor is frissítjük a lehetséges kategóriákat? A kínálkozó mód az lenne, hogy akkor, amikor létrehozunk egy feladatot. Ha a feladat kategóriája még nem szerepel az eddigi kategóriák között, akkor írjuk hozzá a listához. Hát igen, és ha módosítunk egy feladatot, akkor lehet, hogy a kategóriát is átírjuk. Ha meg törölünk egyet, akkor meg kell nézni, hogy ez volt-e a kategória utolsó használata, és ha igen, akkor pusztuljon a listából is. Feladatlista betöltésekor meg kategórialistát is kell generálnunk. Ez már azért több kettőnél, igaz?

Úgyhogy nem így járunk el. Kicsit tán erőforrás-igényesebb lesz a mi változatunk, de talán egyértelműbb is. Meg hát az a helyzet, hogy lassan a ToDoList is attól szenved, amitől a tanuló-programok általában: sok mindent kipróbálnak rajtuk, és ettől néha kicsit félre áll a fejük.

Úgy fogjuk megoldani a feladatot, hogy figyeljük, mikor kap fókuszot a `categoryEdit` objektum. Több módszer is szolgál e célra, mi most csak az egyiket nézzük meg. Amikor az objektumunk megkapja a fókuszot, gyorsan frissítjük a kategóriák listáját. A `QLineEdit` osztály dokumentációjában hiába keresünk a fókusz megkapásakor emittált signal-t, nincsen ilyen. De pár lépcsővel magasabban, a `QApplication` osztálynál megtaláljuk a `focusChanged()` signal-t. Van nekünk ilyen osztályú objektumunk? Esetleg eddig nem nagyon tudatosult bennünk, de a `main.cpp` fájlban ott bújik egy, amely a hangzatos és fantáziadús "a" nevet viseli. Volt már róla szó, még a régi szép időkben, a könyv elején, amikor a grafikus programok főciklusáról beszéltünk.

A `MainWindow`-on belül a `main.cpp` állomány egyéb objektumaira nem hivatkozhatunk, így a kapcsolatot kiépítő `connect` utasítást kénytelenek leszünk magában a `main.cpp`-ben kiadni. Előbb azonban írjuk meg a slot deklarációját és törzsének kezdeményét a `MainWindow`-ban. A `focusChanged()` signal-től két, `QWidget` osztályú objektumra mutató mutatót kapunk: az első arra az objektumra mutat, amelyiké volt a fókusz, a második arra, amelyiké lett. Ennek fényében a slot piszkozata a következő:

```
void MainWindow::appFocusChanged(QWidget *old, QWidget *now)
{
    qDebug() << "focus changed";
}
```

A `main.cpp` fájlban kell kiadnunk a `connect` utasítást. A szokott formához képest két ponton térünk el. Az egyik, hogy megadjuk az osztálya nevét is (statikus tagfüggvényről van szó). A másik pedig az, hogy a két objektum mutatójára van szükségünk, ezeket elő kell állítanunk. Az utasítás így néz ki:

```
QObject::connect(&a, SIGNAL(focusChanged(QWidget*, QWidget*)), &w,
SLOT(appFocusChanged(QWidget*, QWidget*)));
```

Helyezzük a `return a.exec();` sor elé. Futtassuk és próbáljuk ki, megy-e minden rendben.

Szóval észre fogjuk venni, ha belép a felhasználó a `categoryEdit` szerkesztőbe. Ekkor majd elő kell állítanunk a lehetséges objektumok listáját, és bepakolni az egész hóbelevancot egy karakterlánc-listába. A következő tagfüggvény pont ezt a feladatot látja el:

```
QStringList MainWindow::categories()
{
    QStringList list;
    for(int i = 0; i < tasks.size(); i++)
    {
        QString category = tasks.at(i)->category();
        if(!list.contains(category))
            list << category;
    }
    return list;
}
```

Már csak az a kérdés, hogy miként vegyük rá a `completer` objektumot az esetleg megváltozott lista használatára? Hiszen a múltkor egy kész listával példányosítottuk, és a példányosítás után meg hiába változtatjuk meg a kiinduló listát. Az egyik megoldás lenne, hogy az `appFocusChanged()` slot-ban mindig új `QCompleter`-t példányosítunk a `categories()` tagfüggvény kimenetéből, és a `categoryEdit` objektumnál megadnánk az új példányt `completer`ként. A másik megoldás használatához újat kell tanulnunk. És ugyan mi egyébért olvas az ember strandkönyvet, mint hogy sok újat tanulhasson?

Aki már olvasott programtervezési mintákról (design patterns), aligha kerülte el az MVC (Model-View-Controller, modell-nézet-vezérlő) mintát, aki meg nem olvasott, az most kap egy egyetlen mondatos összefoglalót. A tapasztalat szerint szép és okos dolog szétválasztani a programnak azt a részét, amely az adatszerkezet változásaival foglalkozik (a modell) attól, amely megjeleníti az adatot (a nézet) és attól, amelyen keresztül a felhasználó változtatgat az adaton (a vezérlő).

A Qt a fenti koncepció View és Controller részeit View néven egyesítve¹³ a Modell-View mintát preferálja, azaz van egy adatmodellünk, meg egy valamink, ami azt megjeleníti és amin keresztül módosítani tudjuk. És most jön a lényeg: a Qt-ban a modellt használó osztályok *önműködően észreveszik és feldolgozzák a modell változásait*. Egyelőre ennyit az MV mintáról, egy-két fejezet, és kapunk belőle csöstit.

A `QCompleter` osztályú objektum persze nem „nézet” a szó alapértelmében, de abban az értelemben igen, hogy a példányosítását követően is észreveszi, ha a modellje változik.

Azaz minekutána a `mainwindow.cpp` állományban elhelyeztük első modellünk, a `QStringListModel` használatát lehetővé tevő fejléceket és a

```
QStringListModel *completerModel;
```

sorral deklaráltunk egy jó kis mutatót, a `MainWindow` konstruktorában a `completer` objektumot példányosító sort váltsuk fel az alábbi kettővel:

```
completerModel = new QStringListModel(this);
completer = new QCompleter(completerModel, this);
```

Az első sor előállítja azt a modellt, ahova friss, ropogós mutatónk mutat, a második pedig használatba is veszi. Már csak az a baj, hogy a modell még üres. Mikor is kell nekünk először a `completer`? Akkor, amikor először gyalogolunk bele a `categoryEdit` objektumba, a kategóriaszerkesztő-sorba. Azt ugye meg már észre is vesszük egy `appFocusChanged()` slot-tal. Ja, nem. Azt vesszük észre, hogy máshova került a fókusz, de ugye mi nem akarunk *minden* fókuszváltáskor kategórialistát képezni, és azt modellbe tölteni, igaz?

13 Vannak viták arról, hogy a Qt terminológiája helyes-e: <http://stackoverflow.com/questions/5543198/why-qt-is-misusing-model-view-terminology>

A slot két paraméteret kap, az egyik a fókuszváltás előtt fókuszban volt elem, a másik a fókuszváltást követően fókuszba került elem *mutatója*. (Ez utóbbinak *now* a neve.) Miként tudjuk meg, hogy melyik elem került fókuszba? Az elem osztályát épp ki tudjuk halászni a

```
now->metaObject()->className();
```

utasítással, de ez nekünk édeskevés: még ebben a kis programban is két azonos osztályú elem van a felhasználói felületen. A fókuszba objektum neve kellene, az meg nem triviális.

Elindulunk megtanácskozni a dolgot a hörcsöggel, de félúton rájövünk... hogy mutatót kaptunk, amin remekül lehet összehasonlító műveletet végezni!

Szóval a slot törzse ez lesz:

```
Q_UNUSED(old);
if(now == ui->categoryEdit){
    completerModel->setStringList(categories());
}
```

Az első sorban lévő makró arra jó, hogy a fordító ne figyelmeztetgessen bennünket arról, hogy nem használjuk az *old* változót. A második sor megvizsgálja, hogy a minket érdeklő objektum kapott-e fókuszt, és ha igen, akkor a harmadik sorban a *categories()* tagfüggvény *QStringList* osztályú kimenetét betöltjük a *QStringListModel* osztályú *completerModel*-be. A modell változását a megfelelően példányosított *completer* objektumunk automatikusan észreveszi, és mindig a legfrissebb listát tolja elénk.

Kipróbálhatjuk a programunkat, és ha megy, akkor az imént a közeledtünkre előbújó hörcsögnek vihetünk némi nasit – szegény úgy hoppon maradt az előbb, hogy nem nyilvánvaló az állatpszichológus mellőzhetősége.

Ha a hörcsög megvigasztalódott, akkor a főablak konstruktorában helyezzünk el még két remek sort, a *completer* objektum példányosítását követően:

```
completer->setCaseSensitivity(Qt::CaseInsensitive);
completer->setModelSorting(QCompleter::CaseInsensitivelySortedModel);
```

Az első megoldja, hogy a kis „p” begépeléskor a „Pali” és a „papagáj” is megjelenjen, a másik pedig arról tesz, hogy a modell tartalmát rendezzük megjelenítés előtt, méghozzá nem figyelve a kis- és a nagybetűkre.

7.3. Automatikus fájlbetöltés a program indulásakor – a *QSettings* osztály

A *QSettings* osztályú objektumok feladata a program beállításainak mentése, mégpedig úgy, hogy kihasználjuk az operációs rendszerben megszokott módszereket. Ez a Unix-szerű rendszereken a legegyszerűbb, itt egyszerű INI-fájlokról van szó. OS X-en XML beállításfájlokat használ, Windows-on pedig a rendszerleíró adatbázisba (registry) menti a beállításokat. A programozónak pusztán annyi a dolga, hogy megadja, milyen beállítás milyen értéket kapjon – a beállítások, beállításfájlok elhelyezése már nem az ő dolga.

Az objektum példányosítása során meg kell adnunk a cégünk és a termékünk nevét – a beállítások helyét ez alapján álmódja meg a *QSettings*. Ha a programból több helyen, több objektumban is babrálunk a beállításokkal, akkor érdemes a program indulásakor megadnunk

az alkalmazás nevét, az internet-tartományt¹⁴ és a cég nevét a `QCoreApplication` osztály statikus tagfüggvényeinek használatával. Ha így járunk el, a `QSettings` osztályú objektum példányosításakor nem kell újra meg újra megadni az adatokat: kiolvassa magának a tárolt értékeket.

Helyezzük el például a programunk `mainwindow.h` állományának betöltendő fejlécei között a `<QCoreApplication>` fejléct, majd a főablak konstruktorának elején helyezzük el a:

```
QCoreApplication::setOrganizationName("Sufni & Co.");
QCoreApplication::setOrganizationDomain("example.com");
QCoreApplication::setApplicationName("ToDoList");
```

sorokat. A `FileOperator` osztály privát tagjai között példányosítsunk magunknak objektumot (szükség lesz a `<QSettings>` fejlécre):

```
QSettings settings;
```

A `performSaveOperation()` tagfüggvény sikeres mentést követően a `fileName` változó értékéül megadta az `fn` helyi változóban tárolt fájlnévet. Ezzel az utasítással még el is menti ezt a beállítást:

```
settings.setValue("GeneralSettings/ListFileName", fn);
```

Futtassuk a programunkat és mentjük az adatainkat.

A mentés után elvileg azonnal megvannak az adataink, de lehet, hogy kicsit késnek. Ha ez zavaró, nézzük meg az osztály dokumentációjában a `sync()` és a `status()` tagfüggvényt.

Ubuntu Linuxon ellenőrizve:

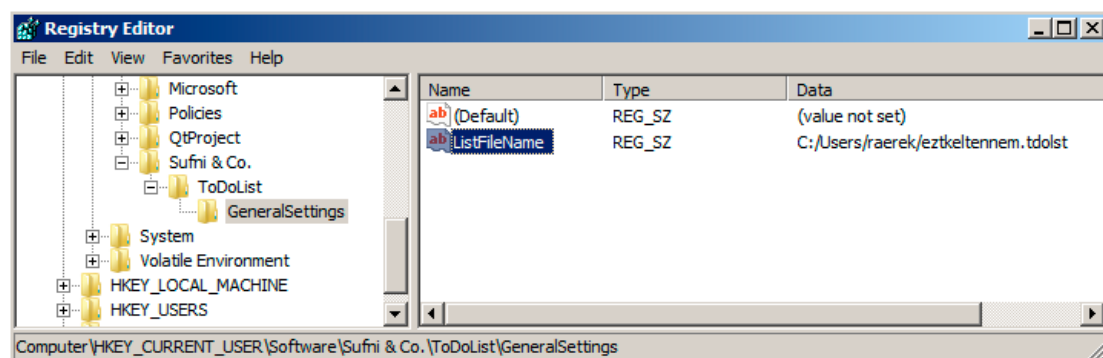
```
raerek@cucc:~$ cat .config/Sufni\ \&\ Co./ToDoList.conf
```

```
[GeneralSettings]
```

```
ListFileName=/home/raerek/eztkelltennem.tdolst
```

A Windows rendszerleíró adatbázisában pedig ez látszik:

Az, hogy az egyes platformokon hol van a beállítások pontos helye, az osztály dokumentációjából kiderül.



10. ábra: Beállításaink a Windows registry-ben

A beállítás visszatöltése sem számít igazán bonyolult műveletnek. Azt érdemes róla megjegyeznünk, hogy – lévén egy szöveges fájlban tárolt értékről nehéz első blikkre megmondani, hogy milyen típusú – a `QSettings::value()` tagfüggvény visszatérési típusa `QVariant`. Hogy az meg mi a csuda? Hát, szabad fordításban magyarul nagyjából így hangzik:

14 OS X-en van jelentősége – ott van szokásban ennek az adatnak a használata

„mittomén”. Azaz nekünk kell a `toInt()`, a `toString()`, a `toMiegymás()` tagfüggvénnyel a kívánt típusra alakítanunk.

Már csak az a kérdés, hogy miként valósítsuk meg a teendők automatikus betöltését. Betöltjük a beállításokat – pontosabban azt az egyet, ami eddig megvan, a listafájl nevét –, eddig tiszta. És aztán?

Gondoljuk át, miként történik jelenleg egy fájlmegnyitás:

1. a felhasználó a megnyitás `QAction` osztályú objektumán kiváltja a `triggered()` signal emittálásához vezető eseményt – kattint, vagy `Ctrl+O` billentyűkombinációt nyom
2. a főablak slot-ja futni kezd, és
3. hívja a `fileOperator.open()` tagfüggvényt, amely
 - I. fájlnevet szerez és ezt átadva hívja a `fileOperator.performLoadOperation(QString fn)` tagfüggvényt, amely
 - II. betölti a fájlt, előállítja a `QList` objektumot és átadja
 - III. a `fileOperator.open()` tagfüggvénynek, ami átadja
4. a slot-nak, ami folytatva a munkát, megjeleníti az eredményt a `listWidget`-ben

Arab számokkal a `MainWindow`-on belüli műveletek, római számokkal a `fileOperator` objektum munkája.

Ez azonban arra az esetre készült, amikor nincs fájlnevünk. Nekünk meg most van, a `settings` objektum árulja el. Így persze hívhatnánk közvetlenül a `performLoadOperation()` tagfüggvényt. Gyorsan alakítsuk is publikussá.

A következő dolgunk az, hogy a slot-ból (becsületes nevén: `on_actionMegnyitas_triggered()`) külön tagfüggvénybe tesszük át azt a részt, ami a kapott `QList` osztályú objektumból kipakolássza a teendőket a `listWidget`-be. Ezzel a slot törzséből az első sor kivételével mindent elveszünk. Az új tagfüggvény neve legyen `showJobs()`. A slot-ból persze hívunk kell, így a slot törzse végül mégiscsak két soros lesz.

Utolsóként még az a feladatunk van, hogy írunk egy tagfüggvényt, amely kiolvassa a megfelelő beállítást. Ha van ilyen, akkor a talált fájlnev használatával megpróbálja betölteni a feladatokat, majd a szép, új, ropogós `showJobs()` függvényt használva meg is jeleníti őket. Íme:

```
void MainWindow::openLastSavedTdoListFile()
{
    QSettings settings;
    if(settings.contains("GeneralSettings/ListFileName")){
        tasks = fileOperator->performLoadOperation(settings.
value("GeneralSettings/ListFileName").toString());
        showJobs();
    }
}
```

Ne feledjük a `MainWindow` konstruktorából hívni ezt a remek kis tagfüggvényt.

Kész vagyunk. Odaadhatjuk kicsit a mókuskereket a hörcsögnek, teperjen most már ő is.

8. KERESÉS, GRAFIKON ÉS TÖBBNYELVŰSÉG

Ebben a fejezetben az első dolgunk az lesz, hogy végre használjuk is a megadott kategóriákat: megvalósítjuk az egy adott kategóriához tartozó feladatok megjelenítését. Ha ezzel elkészültünk, megrajzoljuk a leggyakrabban használt kategóriák grafikonját. A grafikonrajzolás után pedig megismerkedünk a `tr` függvénnyel és azzal, hogy miként tehetjük programunkat többnyelvűvé.

8.1. A keresőfül

Ha ilyesmit szeretnénk kialakítani, a grafikus felhasználóifelület-tervezőben lévő elemek közül essen választásunk a **Tab Widget**-re. Húzzuk ki a főablakba, és kis ügyeskedéssel pakoljuk bele egész eddigi művünket. Ha sikerült, kattintsunk valahova az első fül területére, de az elemeink *mellé*, azaz „sehova” – hasonlóan ahhoz, amikor az első elrendezést alakítottuk ki. Ekkor jött a döntő lépés: itt kellett akár vízszintes, akár függőleges elrendezést alkalmazni – a lényeg az volt, hogy valamilyen elrendezés legyen. Nos, legyen most is valamilyen elrendezésünk, és szépen átméretezhető lesz a főablak, benne az összes elemmel.

Adjunk nevet a két fülnek. Az első annyira nem is fontos, a másodiké, csak azért, hogy a könyvvel szinkronban legyünk, legyen „keresés” - igen, az aláhúzás a gyorsbillentyűt jelöli. A keresés fülön helyezzünk el egy újabb `QLineEdit` osztályú objektumot, a neve legyen `searchEdit`. Ide fogjuk beírni, hogy melyik kategória érdekel bennünket, s ekképpen volna értelme ezt a szerkesztőt is rávenni, hogy használja a jó kis automatikus felajánlásokat. Úgyhogy helyezzük el a főablak konstruktorában az alábbi sort:

```
ui->searchEdit->setCompleter(completer);
```

Ha jó ötletnek tartjuk, hogy a lehetséges kategóriákat tartalmazó `completerModel` akkor is frissüljön, mikor ez a szerkesztősor kap fókuszot, akkor írjuk át az `appFocusChanged()` tagfüggvényben megfogalmazott feltételt ilyenre:

```
if((now == ui->categoryEdit)|| (now == ui->searchEdit))
```

Eddig minden szép és jó, talán azt az egyet leszámítva, hogy a találatok nem jelennek meg. Helyezzünk el számukra egy másik `QListWidget` osztályú objektumot, a neve maradhat `listWidget_2`.

A `showJobs()` tagfüggvényről erősen puskázva alakítsuk ki azt a tagfüggvényt, ami majd a `searchEdit` tartalmának felhasználásával kiírja a találatokat:

```
void MainWindow::showJobsHavingCategory()
{
    ui->listWidget_2->clear();
    QListIterator<QSharedPointer<Task> > ji(tasks);
    while(ji.hasNext())
        if(ji.next()->category() == ui->searchEdit->text())
            ui->listWidget_2->addItem(ji.next()->job());
}
```

Eltüntettük a `tasks` objektum ürességét vizsgáló sort, és írtunk egy újat is, amelyik a `while`-ciklus közepében dönt arról, hogy az adott elemet kiírjuk-e, avagy sem. A magunk bölcsességében úgy döntünk, hogy nem helyezzünk el a keresés megindítására szolgáló

külön nyomógombot, hanem a `searchEdit` egyik signal-ját, az `editingFinished()` nevűt használjuk. Az imént megírt jó kis függvényünket minősítsük át privát slot-tá, a főablak konstruktorába pedig írjuk még be a

```
connect(ui->searchEdit, SIGNAL(editingFinished()),
        this, SLOT(showJobsHavingCategory()));
```

utasítást. Elkészültünk a keresőfüllet.

8.2. Grafikká válva grafikont rajzolunk, és a tetejébe még iterálgatunk is

Elképzelhető, hogy amit ebben az alfejezetben fejlesztünk, nem teljesen életszagú. Ugyanis most azzal fogjuk tölteni az időt, hogy a drága kis `QTabWidget` osztályú elemünkön létrehozunk egy harmadik fület, amin grafikont készítünk a leggyakrabban használt kategóriákról. Ha valahogy nem jelenik meg lelki szemeink előtt az a felhasználó, aki épp azon filózgat egy ilyen grafikont nézegetve, hogy „Nincs elég aranyhőrcsög kategóriájú tennivalóm – akkor most nem törődöm eleget vele?“, akkor nem tuti, hogy bennünk van a hiba.

Ez azonban, mint azt már számos alkalommal megállapítottuk, egy strandkönyv. Nem a valóság. És ugyan melyik strandkönyvhős¹⁵ szalasztana el egy ilyen alkalmat Qt-ismereteinek gyarapítására?

Az előző fejezetben már megemlékeztünk arról, hogy a Qt előszeretettel használja a Model-View programozási mintát, és elmondtuk a dolog lényegét is. Eszerint van egy modellünk, amelynek fő funkciója az adatok tárolása. Van továbbá egy nézetünk, amely elkéri az adatokat a modellről, megjeleníti őket, és esetleg lehetővé teszi a felhasználónak az adatok változtatását. Ez utóbbi esetben a változásokat visszaírja a modellbe. Mindezt egy nagyon egyszerű példán mutattuk be: egy `QStringList` osztályú objektumon, amely egy `QCompleter` osztályú társának szolgált modellül.

Ezúttal a nézet egy `QGraphicsView` osztályú objektum lesz. A `tabWidget`-re vegyünk fel egy harmadik fület, majd bal oldalról, az elemek közül húzzunk ki rá egy nézet-objektumot. Az automatikus átméretezést biztosítandó a szokásos módon – mellé, de még a `tabWidget` kereten belül kattintva – adjunk meg vízszintes vagy függőleges elrendezést. A nézettel már csak egy dolgunk lesz: ki kell majd jelölni számára a modellt. Modellünk egyelőre túl kevés van, legfőbb ideje hát kialakítani egyet.

Lévéen a főablak `cpp`-fájlja már így is nagyon hosszú, meg amúgy is egy jól körülhatárolható feladatkört ellátó tagfüggvény-együttes kialakítása lesz az alfejezet további témája, a modellt és a segéd-tagfüggvényeit új objektumban helyezzük el. A `QObject` ősű `DiagramCreator` osztályú `diagramCreator` nevű objektumról van szó. Hozzuk létre az osztályt és a fejlécét helyezzük el a `mainwindow.h` állományban. Adjuk meg a mutató deklarációját, majd a főablak konstruktorában a `new` utasítással állítsuk elő az objektumpéldányt is, nem felejtkezve el a `QObject`-leszármazottak automatikus memóriakezelését lehetővé tévő `this` ős megadásáról:

```
diagramCreator = new DiagramCreator(this);
```

Slattyogjunk vissza a `diagramcreator.h` állományba, és publikus mutatóként adjuk meg a „színpadot”, ha tetszik: játékkeret – persze csak a hasonló nevű fejléc használatba vételét követően.

¹⁵ Persze, hogy ennek a strandkönyvnek az aranyhőrcsög az első számú hőse. De egy másodvonalbeli hősnek is lehetnek Qt-ambíciói.

```
QGraphicsScene *scene;
```

Az osztály konstruktorában létre is hozhatjuk a megfelelő objektumot:

```
scene = new QGraphicsScene(this);
```

Ezután visszasompolygunk a főablak konstruktorába, és az előbb beírt sor után jó kerítőként elrendezhetjük a nézet és a modell (ez esetben: szín) egymásra találását:

```
ui->graphicsView->setScene(diagramCreator->scene);
```

Mostantól a színen lévő valamennyi történés azonnal láthatóvá válik a harmadik fülön. Már csak az a szinte említésre sem méltó feladatunk van, hogy a színt benépesítsük.

A `diagramCreator` objektumnak két feladata lesz. Az első, hogy – a `tasks` objektum adatainak ismeretében – előállítsa azt a három soros, két oszlopos adatszerkezetet, amelyben a leggyakrabban előforduló három kategória neve és számossága lesz. A második, hogy ennek az adatszerkezetnek a felhasználásával megrajzolja a grafikon.

A főablakon belül nem szeretnénk sokat bajlódni a dologgal, így aztán az tűnik a megfelelő útnak, hogy átadjuk a `tasks` objektumot a `diagramCreator` objektum egy tagfüggvényének, és onnantól a főablakban – legalábbis fejlesztői szemszögből – elfelejtjük a problémát. Azért hangsúlyoztam a fejlesztői szemszöveget, mert a `graphicsView` nézet természetesen reagál majd a modellje – a `scene` objektum változásaira, de ez már fejlesztői beavatkozást nem igényel.

Hozzuk hát létre a `DiagramCreator` osztályban azt a publikus tagfüggvényt, amelyik majd elvégzi – illetve elvégezteti – a fent vázolt két feladatot. A deklaráció a következő alakot ölti:

```
void drawDiagram(QList<QSharedPointer<Task> > tasks);
```

Kialakíthatjuk az egyelőre üres definíciót is, majd, még mielőtt elfelejtkeznénk róla, kullogjunk vissza a főablakba és ott is a tervezői nézetbe. A `tabWidget` objektumon kattintsunk a jobb egérgombbal, és a felbukkanó menüből a `Go to slot...` lehetőséget választva kattintsunk a `currentChanged(int index)` signal-ra. Az előálló slot-ba pedig írjuk meg azt a törzset, amely a harmadik – tehát a második számú – fül aktiválódásakor az imént kialakított tagfüggvényt hívja:

```
if(index == 2)
    diagramCreator->drawDiagram(tasks);
```

Ha úgy gondoljuk, a `drawDiagram` tagfüggvény törzsében elhelyezett `qDebug() << „0h, yeah!”` utasítással tesztelhetjük is, hogy eddig minden klappol-e.

Mikor kipróbáltuk, kukázhatjuk az utasítást, úgyis írunk helyette sok jót hamarosan. De előbb beszéljük meg, hogy milyen lesz a grafikonunk.

A három leggyakoribb kategóriát jelenítjük meg, méghozzá olyan oszlopdigramként, melynek „oszlopai” valójában vízszintes sávok. A leggyakoribb kategória sávja legyen mondjuk 200 képpont hosszú, a két következőé meg arányosan rövidebb. A sávokat ne egyszerű színnel töltsük ki, hanem valamilyen színátmenettel, és a könnyebb azonosíthatóság végett a sávokra ráírjuk a kategória nevét.

Nos, ha ilyen grafikon szeretnénk, akkor a legelső dolgunk a három nyertes kategória megállapítása, illetve annak feljegyzése, hogy melyik hányszor fordult elő. Már most kijelentenénk, hogy nem óhajtjuk újrainventálni egyik bevált grafikonrajzoló sem. Ha például nálunk négy azonos előfordulás-számú kategória van, akkor megjelenítjük az elsőnek fellelt hármat, és kész.

A három – pontosabban a valahány – legtöbbet használt kategória nevének és előfordulás-számának begyűjtésére külön tagfüggvényt írunk, amelyet a `drawDiagram()` hív majd, egyéb bokros teendők közepette. A függvény visszatérési értéke `QMap<int, QString>` lesz. Persze előbb arra gondoltunk, hogy ez úgy logikus, hogy kategóriánév - előfordulásszám párokat tárolunk, de akkor eszünkbe jutott, hogy a `QMap` osztályú objektumok a bennük tároltakat kulcs szerint rendezik, és nekünk ez a rendezés igen jól jön – még akkor is, ha kénytelenek leszünk visszafelé bejárni majd az objektumot, hogy elsőnek a legtöbbször előforduló kategóriát kapjuk meg. Szóval ezért a `QMap<int, QString>` változat: elől a „hányszor”, utána a „mi”.

A függvénynek át fogjuk adni a `tasks` objektumot (pontosabban annak `diagramCreator`-béli másolatát, aminek szintén `tasks` a neve), és azt a számot, amelyből megtudja, hogy hány kategóriát kell kirajzolni. A `drawDiagram()` tagfüggvény majd a következőképp hívja a csilivili új tagfüggvényt:

```
QMap<int, QString> best3Categories = bestCategories(tasks, 3);
```

Figyeljünk a változó és a tagfüggvény nevei közötti egyetlen különbségre. A 3-as számra a tagfüggvény belsejében (meg persze már a deklarációjában is `firstX` néven utalunk).

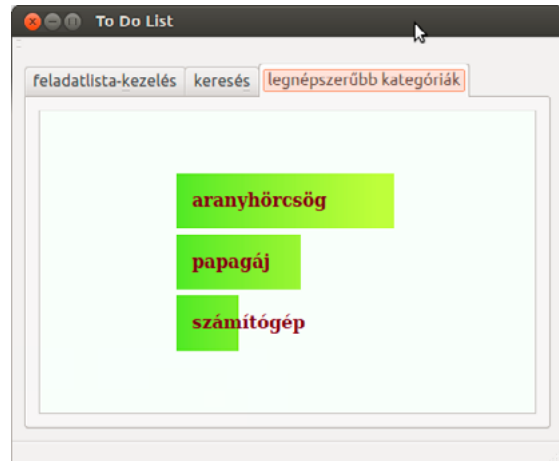
Lássuk, mihez kezd a `bestCategories()` tagfüggvény a neki juttatott adatokkal. Először is kialakítunk egy `QHash<QString, int>` adatszerkezetet, összeszámolva benne az összes kategória összes előfordulását:

```
QHash<QString, int> categoriesUsage;
for(int i = 0; i < tasks.size(); i++){
    QString cat = tasks[i]->category();
    if(categoriesUsage.contains(cat))
        categoriesUsage[cat] += 1;
    else
        categoriesUsage[cat] = 1;
}
```

Valami ilyesmit kapunk majd:

aranyhőrcsög	7
számítógép	2
bicikli	1
szekrény	2
papagáj	4

A `QHash<QString, int>` alakból arra következtetünk, hogy ez olyasmi, mint a `QMap`. Nem is tévedünk, hiszen első blikkre csak egyetlen, de annál nagyobb különbség van a két osztály között: míg a `QMap` osztályú objektumok bejárásakor mindig a kulcsuk szerint vannak rendezve (és ekképp a kulcsuk csak olyan dolog lehet, amire értelmezett a kisebb mint, nagyobb mint



11. ábra: Ilyen lesz a grafikonunk

összehasonlítás), a `QHash` osztályúak megjósolhatatlan sorrendben bukkannak fel. Azért használtunk `QHash` osztályú adatszerkezetet a `QMap` osztályú helyett, mert ebben a strandkönyvben szeretnénk tolsztoji magasságokba emelkedni – legalábbis ami a szereplők számát illeti. A `QHash` osztály ráadásul kicsit gyorsabb is a `QMap`-nál a legtöbb esetben – ez nagyjából húsz, vagy annál több feladat esetében válik igazzá¹⁶. De ez aligha jelentős szempont a mi esetünkben.

A kész leltárt áttöltjük egy, immáron a tagfüggvény visszatérési értékének megfelelő `QMap`-ba:

```
QMap<int, QString> categoriesOrdered;
QHash<QString, int>::const_iterator ci = categoriesUsage.
constBegin();
while(ci != categoriesUsage.constEnd()){
    categoriesOrdered.insertMulti(ci.value(), ci.key());
    ++ci;
}
```

Az áttöltéshez STL-típusú konstans iterátort használunk – futólag már megismerkedtünk vele régebben. Az adatok áttöltését a `QMap` osztályú `categoriesOrdered` objektumba nem a szokásos `categoriesOrdered[kulcs] = "érték"` formában végezzük, mert számítunk rá, hogy esetleg „aranyhőrcsög” kategóriájú feladatunk ugyanannyi van, mint „csekkbefizetés” kategóriájú. (Bár az nem lehet. Még elképzelni is rossz... Brr...) Ilyenkor a másodikként áttöltött kategória felülírná az elsőt. Szerencsére azonban itt a `QMap::insertMulti()` tagfüggvény, amellyel egy kulcshoz több értéket is megadhatunk.

Az előállított és csurig feltöltött `categoriesOrdered` objektum tartalma nagyjából a következő:

1	bicikli
2	szekrény
2	számítógép
4	papagáj
7	aranyhőrcsög

Merthogy a `QMap` ugyebár kulcs szerint rendezett. Már csak az vele a baj, hogy ami nekünk kell belőle, az az utolsó három sor, ráadásul a végéről. Nosza, pusztuljon, amin fölösleges! A törlés során megint STL-típusú iterátort használunk, de nem konstans iterátort, mert ugye ha valaminek kitöröljük egy részét, akkor az a valami már nem konstans. A `firstX` változó az, amelyben a tagfüggvény a hívásakor a szükséges, visszaadandó sorok számát kapta paraméterül.

```
QMap<int, QString>::iterator i = categoriesOrdered.begin();
while(categoriesOrdered.size() > firstX){
    i = categoriesOrdered.erase(i);
}
```

Ha figyelmesen böngésszük a fenti négy sort, feltűnhet, hogy a `while`-cikluson belül sosem léptetjük az iterátort. Vagy mégis? A `QMap::erase()` tagfüggvény visszatérési értéke az a hely, ahova az aktuális elem törlése után az iterátor mutat.

Minekutána a `categoriesOrdered` objektumot immáron `firstX` sorúra töpörítettük, laza mozdulattal visszaadjuk a hívónak:

```
return categoriesOrdered;
```

16 http://woboq.com/blog/qmap_qhash_benchmark.html

Mielőtt nekilátnánk, hogy most már tényleg megírjuk a `drawDiagram()` tagfüggvény törzsét, megvakargathatjuk a hörcsög picike pocikáját. Ki ne nyomjuk belőle az imént elrágcsált mogyorókat. Se.

Akkor hát `drawDiagram()`. Haladjunk sorban. Még mielőtt sort kerítenénk a `best3categories` deklarálására és feltöltésére, elvégezzünk pár feladatot:

```
scene->setSceneRect(0,0,200,160);
```

```
QFont font("Serif");  
font.setWeight(QFont::Bold);
```

```
QColor backgroundColor(250, 255, 250);  
QBrush backGroundBrush(backgroundColor);  
scene->setBackgroundBrush(backGroundBrush);
```

```
QColor startColor(145,232,66);  
QColor endColor(210,255,82);
```

```
QLinearGradient linearGradient(QPointF(0, 0), QPointF(200, 0));  
linearGradient.setColorAt(0, startColor);  
linearGradient.setColorAt(0.9, endColor);
```

```
QColor textColor(109,0,25);
```

A sort a szín „burkolónégyszögének” beállításával kezdjük. Ez egy olyan négyszög, amely valamennyi, a színen szereplő objektumot magába foglalja. A szélessége 200 képpont, mert ezt beszéltük meg a grafikon sávjainak maximális hosszáról. A magassága meg azért 160 képpont, mert a sávok 50 képpont szélesek lesznek, ami három sávnál 150 képpont, de köztük még ki is hagyunk 5-5 képpontot.

A következő két sor a használandó betűtípus megadása. Nem adunk nagyon konkrét betűtípus-nevet, mert nem sok esélyünk van elmondani, hogy a nagy multiplatformitás közepette ugyan milyen betűtípusok leledzenek majd a céleszközön, de annyit azért kijelentünk, hogy serif, azaz talpas betűket szeretnénk. Az operációs rendszer meg majd előadja az alapértelmezett talpas betűtípusát. Utóbb azt is megadjuk, hogy ha már `font`, legyen félkövér.

Az eztán begépelte három sorban RGB-kódokkal megadunk egy nagyonhalványszürke színt, a színnel pemzlit (jó-jó: ecsetet) inicializálunk, és ezzel a pamaccsal átszínezzük a színpadot. Megadhatnánk egy sorban is:

```
scene->setBackgroundBrush(QBrush(QColor(250,255,250))));
```

A `startColor` és az `endColor` nevű színt a lineáris színátmenet kiinduló- illetve végpontjaként kívánjuk felhasználni. Színátmenetből többfélét is ismer a Qt, mi most leragadunk a legegyszerűbbnél. Meg kell adnunk a két végpontot (a `QPointF` osztályban az `F` a Floating point-ot jelöli), illetve azt, hogy a távolság mekkora részénél kell elérnie a színátmenetnek a nevezett színt. Esetünkben a színátmenet a `startColor` színnel kezdődik, és már majdnem a végére érünk, mire `endColor`-ra kell alakulnia. Egy színátmeneten belül annyi színt adhatunk meg a 0-1 távon belül, amennyit nem szégyellünk.

Végezetül megadunk egy olyan színt, amellyel majd szövegeinket kívánjuk kiemelni az alapértelmezett feketeségből.

Most jött el az ideje annak, hogy átsomfordáljunk a `diagramcreator.h` állományba és megadjunk néhány privát mutatót:

```
QGraphicsRectItem *rect1;
QGraphicsRectItem *rect2;
QGraphicsRectItem *rect3;
QGraphicsTextItem *category1;
QGraphicsTextItem *category2;
QGraphicsTextItem *category3;
```

Az első három a sávokra mutat majd, a második három a kategóriák neveinek kiírására. Visszasuntyoghatunk a tagfüggvényünkbe.

Visszaérve a már ismertetett módon előállítjuk a legjobb három kategóriát bújató, `QMap` osztályú adatszerkezetet. De nem elégszünk meg ezzel, hanem rögtön iterátort is definiálunk neki, ráadásul ezúttal Java-félét. Ha még emlékszünk, egy jól szituált Java-féle iterátor nem az elemekre, hanem azok közé, elé, illetve mögé mutat (mondjuk akkor is oda mutat, ha véletlen nem emlékeznénk). Minthogy nekünk elsőként az utolsó elem kell a `best3categories` objektumból, gyorsan be is állítjuk az iterátort a kategóriák utánra:

```
QMap<int, QString> best3Categories = bestCategories(tasks, 3);
QMapIterator<int, QString> ji(best3Categories);
ji.toBack();
```

A következő sorban megálmodjuk, hogy mennyi legyen a grafikon-sávok alapegysége. Azt mondtuk, hogy amiből a legtöbb van, az 200 képpont hosszú lesz. Azaz:

```
float diagramUnit = 200/ji.peekPrevious().key();
```

Hmm, mi is az a peek? Kukucskálás? Na, ezt gondoljuk végig. Az a helyzet, hogy az iterátorunk az utolsó elem *után* áll. Ha egy `ji.previous().key()` hívással kérnénk el az előtte álló, azaz a legutolsó elem kulcsát, akkor az iterátor is visszalépne egyet. Márpedig nekünk pillanatokon belül szükségünk lesz az utolsó elem értékére is. Azaz most csak visszakukucskálunk az előző elemre, lekukucskáljuk a kulcsát, és majd csak akkor lépünk visszább, ha az értéket nézzük le.

A következő négy sorral kirajzoljuk az első sávot, és ráírjuk a kategória nevét:

```
rect1 = scene->addRect(0, 0, diagramUnit*ji.peekPrevious().key(),
50, Qt::NoPen, linearGradient);
category1 = scene->addText(ji.previous().value(), font);
category1->setPos(10,10);
category1->setDefaultTextColor(textColor);
```

Az `addRect()` tagfüggvénnyel kérünk egy téglalapot a színre. A `rect1` mutatóban eltesszük a létrejött téglalap memóriabéli helyét, amire sok szükségünk elvileg nem lesz, de azért eltesszük. Az első négy paraméter a grafikon-sáv átlójának végeit jelöli. A `diagramUnit*ji.peekPrevious().key()` szorzatnak a gépi számábrázolás határai okozta hibán belül kutya kötelessége 200-nak lenni, és pusztán a következő grafikon-sáv megadásához való hasonlatosság okán írtuk így. Az ötödik paraméter `QPen` osztályú, ez a konstans érték épp azt jelenti, hogy a sávunk körül ne legyen semmiféle vonal. A hatodik paraméter `QBrush` osztályú, és a sávot kitöltő pemzli színét adja meg. A szín lehet színátmenet is, esetünkben is ez a helyzet.

A második sor a kategórianévet írja ki – a szöveg mutatóját megint csak eltesszük. Vegyük észre, hogy a `ji.previous().value()` tagfüggvény-hívás a kategória nevének visszaadásán

túl az iterátort is lépteti – most már nem csak kukucskálunk. A szöveg betűtípusa a korábban megadott **font** változóban beállított értéknek megfelelő lesz.

A harmadik sor a szöveg burkolónégyzetének bal felső sarkát állítja máshova, a negyedik meg a pár sorral feljebb beállított színűre színezi a kategória nevét.

A következő sávot kirajzoló kódot még megnézhetjük itt is, a harmadikat meg már bizonyosan egyedül is összehozzuk, s ha mégis másként történne, hát letöltjük a forráskódot a könyv webhelyéről. Kövessük nyomon, ahogy először csak kukucskálunk, majd ismét léptetjük az iterátort:

```
rect2 = scene->addRect(0, 55, diagramUnit*ji.peekPrevious().key(),
50, Qt::NoPen, linearGradient);
category2 = scene->addText(ji.previous().value(), font);
category2->setPos(10,65);
category2->setDefaultTextColor(textColor);
```

Így történt, hogy elkészült a grafikon.

8.3. Többnyelvű alkalmazás és a QT Linguist

Az elején azt ígértük, hogy a forráskódban minden angolul lesz, de ezt az ígéretünket csak részben tartottuk be. Eddig.

Mielőtt bármit újítanánk, gyorsan írjuk át angolra az összes előforduló karakterláncot a programban – nem gáz, ha nem lesz nagyon angolos, csak úgy nagyjából elmenjen. Két helyen vannak magyar szövegek: a **fileoperator.cpp** és a **mainwindow.cpp** fájlban. A karakterláncokon kívül át kell írni a felhasználói felület összes elemének feliratát, és a menüt is (érdemes használnunk az **Object Inspector** ablakát, a **menuBar** objektumra ugyanis problémás lehet rákattintani). A menü meg ugyebár **QAction**-öket definiál. Ilyen kis programnál van vagy négy objektumunk ebből az osztályból, így úgy döntünk, hogy ezeknek az objektumoknak nem csak a feliratát, de a nevét is módosítjuk. Ha azonban így teszünk, akkor kénytelenek leszünk a kapcsolódó slot-ok nevét is átírni. Nosza, tegyük meg ezt is, nem felejtkezve el a fejlécfájlból lévő deklarációkról sem.

Futtassuk a programunkat, és nézzük meg, hogy nem hagytunk-e ki valamit.

A következő dolgunk, hogy a megjelenítendő karakterláncokat körbe vesszük a **tr()** függvénnyel, aminek a neve egyébiránt a *translate*, magyarul *lefordít* igéből származik. Azaz, ami az előző alfejezet végén még

```
mb.setInformativeText(„Csinálj valami okosat.”);
```

volt, az mostanra

```
mb.setInformativeText(tr(„Try and do something wise.”));
```

lett.

Nem kell **tr()** -rel körülvenni a felhasználói felület elemeinek, illetve a **QAction**-öknek a nevét. Ha az objektum **text** tulajdonságát vizsgáljuk a **Property Editor**-ban, és a **text** szó előtti nyílra kattintunk, látszani fog, hogy az objektum alaphelyzetben is *translatable*, azaz *lefordítható*.

Akkor mostanra van egy remek (khmmm) angolsággal beszélő alkalmazásunk, ami egy pár perce még ízes magyar nyelven beszélt, tyuhaj. Most pedig ismét megoldjuk, hogy beszéljen

magyarul, de sokkal bonyolultabban. Igazi kocka-feladat.

Elsőként olyan helyre nyúlunk, ahova eddig még nem: a projektleíró fájlba, a **ToDoList.pro**-ba. Helyezzünk el benne egy

```
TRANSLATIONS = ToDoList_hu.ts
```

sort. A fájlnev hagyományosan, de nem kötelezően az alkalmazás nevéből, alávonásból és a nyelv nevének rövidítéséből áll. A kiterjesztés **ts**. Ha több nyelvi fájl is készítünk, akkor azokat szóközzel elválasztva helyezhetjük el, vagy ha szépen akarjuk csinálni, akkor a sortörést visszaperrel jelezzük:

```
TRANSLATIONS = ToDoList_hu_HU.ts \
                ToDoList_de_DE.ts
```

Ha Qt 4-gyel dolgozunk, és a fordításfájlok UTF-8 kódolásúak, akkor még a

```
CODECFORTR = UTF-8
```

sorra is szükség lesz. Qt 5 esetén nem kell.

Ezt követően a Qt Creator **Tools** menüjéből válasszuk az **External-Linguist-lupdate** lehetőséget. Figyeljünk a kimenetre a lenti ablakban. Fontos, hogy a **lupdate**, és a párja, az **lrelease** futtatását megelőzően mentsük a projektfájlt – a mentetetről mostanra elvileg csúnyán leszoktunk a fordításkor felajánlott automatikus mentés miatt. A **lupdate** egyébiránt nem egyéb, mint egy parancssori eszköz, azaz futtathatjuk onnan is, ha épp úri kedvünk úgy diktálja. És hogy mire való? Megnyitja a **.pro** fájlt, megkeresi, hogy mi lesz a fordításokat tartalmazó fájl neve, és létrehozza a fájlokat. Szép XML-eket kapunk, benne az összes lefordítanivalóval. Ha a forráson módosítunk, a fordítást megelőzően újra futtatnunk kell a **lupdate** alkalmazást, amely a változásoknak megfelelően frissíti a **ts**-fájlt.

Még mindig nem kezdünk a fordításhoz, előbb még pár sorral bővítjük a **main.cpp**-t. Kell bele egy olyan, amelyik betölti a `<QTranslator>` fejléct, meg még három sor a **main** függvény törzsébe:

```
QTranslator translator;
translator.load("ToDoList_hu_HU");
a.installTranslator(&translator);
```

Ha azt szeretnénk, hogy a programunk a meglévő fordítások közül automatikusan válassza ki a felhasználó nyelvi beállításainak megfelelőt, akkor a fenti blokk második sorát cseréljük ki az alábbi kettőre:

```
QString locale = QLocale::system().name();
translator.load(QString("ToDoList_") + locale);
```

Ha pedig azt szeretnénk, hogy más kezdőkönyvtár-beállítással – azaz akárhonnan – indítva is megtalálja a programunk a **qm**-fájlját, akkor a fenti két sor közül a másodikat cseréljük erre:

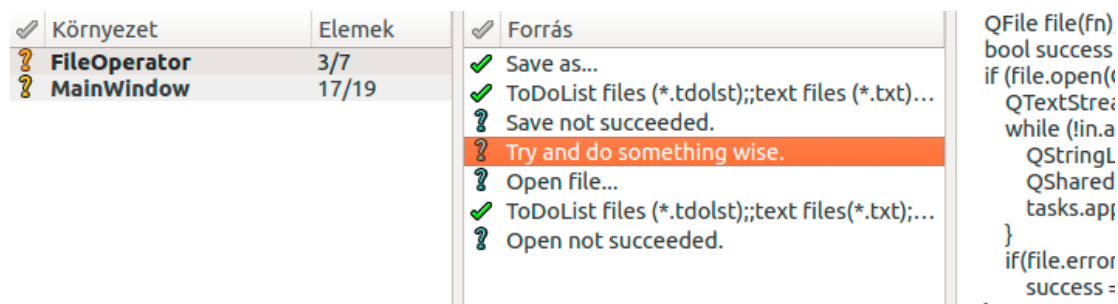
```
translator.load(QString(QCoreApplication::applicationDirPath() +
"/ToDoList_") + locale);
```

Persze a `<QCoreApplication>` fejléc betöltéséről gondoskodni illik. Windows-on sem lesz zűr abból, hogy nem visszaper használunk a nyitó idézőjel után, de ha mégis félünk, akkor használjuk a `QDir::toNativeSeparators()` tagfüggvényt az előre-vissza probléma kezelésére.

Fordítsuk le a programunkat. Ne zavarjon bennünket, hogy egyelőre sehol nincs a fordítás, a Qt-t sem zavarja. Úgy áll a dolog, hogy a Qt-alkalmazás futtatáskor megnézi, hogy van-e fordításfájlja, és ha nincs, hát az sem baj: beszél a forráskódba drótozott karakterláncokkal. Úgyhogy futtassuk is a művünket, és gyönyörködjünk abban, hogy az előző futtatás óta semmit sem változott, pedig mennyit melóztunk.

Ezek után nyissuk meg a Qt Linguist alkalmazást, amelyben nem találunk „Új” menüpontot, hiszen a fordításfájl a `lupdate` programmal kell létrehozni. Nyissuk meg a `ToDoList.hu_hu.ts`-t, és fordítsuk le, amit úgy tartja kedvünk. A karakterláncok fájlanként vannak felsorolva, mellettük ott a forrás is, hátha segít. Ne felejtsük meg a gyorsbillentyűkről, amiket ugyanúgy jelölünk, mint eddig: egy `&`-jellel a megfelelő karakter előtt. Amelyik fordítás kész és tutira jó így, a mellett a kérdőjelet állítsuk pipára.

Ha elkészültünk, mentsünk, és a Qt Creator-ban válasszuk az `External-Linguist-lrelease` lehetőséget. Ténykedésünk hatására a `ts`-fájlokból előállnak azok az optimalizált, `qm`-kiterjesztésű párjaik, amelyeket a program megpróbál majd a futása kezdetén megnyitni – ha erre a `main.cpp`-ben utasítást adtunk.



12. ábra: Haladgatunk a fordítással: pipa jelzi, amivel elkészültünk

Ha most futtatjuk a programunkat alighanem az a kínos meglepetés vár ránk, hogy még mindig angolul beszél, akkor is, ha az operációs rendszer nyelvi beállítása magyar. Nos, ez azért van így, mert a fordítást a Qt Creator alapértelmezés szerint nem a források könyvtárában készíti, hanem valahol máshol. Többnyire a projekt könyvtárának szülőkönyvtárában kell keresnünk valamilyen szokatlanul hosszú, de a projektre utaló könyvtárnevet. Nos, futtatás előtt ide kell átmásolnunk a `qm`-fájlt.

És futtatás, és lőn bábeli zűrzavar. De hát épp ezt akartuk, igaz?

Sikereink öröme a hörcsög kaphat talán egy salátalevelet. Jó hallgatni, ahogy ropogtatja.

9. IKONOK, ABLAKOK, AUTOMATIKUS MENTÉS

A kilencedik fejezet első részében végre kiderül, hogy miként passzol egymáshoz a tennivalók listája és a hörcsög. Miután a titokról le hull a lepel, ablakokat nyitogatunk – pedig nem is szeretnénk szellőztetni. A megnyitott ablakban pedig beállítjuk, hogy hány percenként kérünk automatikus mentést. És, ha már beállítottuk, meg is valósítjuk.

9.1. Ikonok és QAction-ök a QToolBar-on

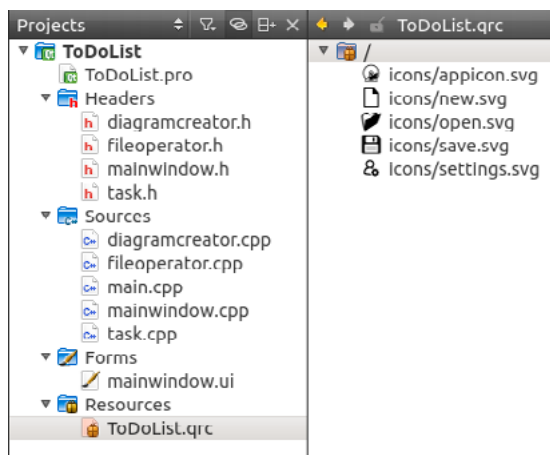
Egyre tökéletesebb alkalmazásunk mindez idáig fájdalmasan nélkülözi az ikonokat. Ezen azonban hamarosan változtatunk. Szerezzünk be ikonokat. Összesen ötöt szeretnénk használni: egy magának az alkalmazásnak, három a fájlműveleteknek – új, megnyitás, mentés –, és egy a majdan elkészülő beállítások ablaknak. Grafikusabb hajlamúak rajzolhatnak maguknak, elvégre nem kell mindig csak programozni a számítógépen. A grafikus hajlamukban vagy képességükben korlátozottabbak pedig keressenek maguknak a neten. A könyvhöz tartozó letölthető forrásokban lévő ikonok többnyire – e könyvhöz hasonlóan – Creative Commonsos licencűek, azaz legalább lesz dolgunk a licenc és a készítő feltüntetésével. Sebj, úgyis mindjárt megtanulunk ablakot készíteni, s így lesz mire használnunk új ismereteinket. A fájlformátumra ne legyen nagy gondunk – ha nem valami nagyon egzotikussal állunk elő, akkor a Qt-nak sem lesz.

Az ikonokat célszerűen a projekt egy alkönyvtárában helyezzük el. Ha nem így tennénk, a Qt Creator úgy is pampogni fog miatta, úgyhogy menjünk elébe a feladatnak.

Az ikonok tárolásához nem volna éppen kötelező erőforrásfájl¹⁷ létrehozni, de érdemes, úgyhogy járjunk el így.

Erőforrás-fájl az ember úgy készíti, hogy jobb egérgombbal kattintott ott, ahol új osztályfájlt is készítené – azért fogalmazunk rébuszokban, mert ez a hely ugye Qt Creator verzióként más és más lehet –, és az **Add new...** lehetőséget választja. A megjelenő párbeszédablakban felsorolt sablonok közül a **Qt** csoportban lévő **Qt Resource file** lesz a barátunk. Az erőforrásfájl nevét választhatjuk akár a programunk nevével megegyezőre, a kiterjesztése meg úgyis **qrc** lesz. Ha megnyílt, akkor először is meg kell adnunk egy prefixet (előtétet), ami alá kerülnek majd a betöltött fájlok. Ha a fájlok a projekt alatti alkönyvtárban vannak, akkor a könyvtár neve is bekerül az elérési útjukba, azaz kis projektünk esetében megkockáztatható, hogy a prefix egy egyszerű per jel („/”, és nem a kolostorfőnök) legyen. A prefixum megadását követően lehetőségünk nyílik a fájljaink betöltésére, s e lehetőséggel élünk is.

Ne felejtjük menteni a fájlt, mert a grafikus felhasználófelület-szerkesztőben csak akkor választható ki a szívünknek kedves ábra, ha a **qrc** fájl már az adathordozón terpeszkedik.



13. Ábra: Erőforrásfájl készítettünk

¹⁷ Nem tökéletes a fordítás de ez van. Angolul: resource file, nem keverendő össze azzal a forrás-fájlal, amiben a forráskódok vannak, és ami angolul csak source file.

Kezdjük akkor az alkalmazás ikonjával. Mindössze annyi a dolgunk, hogy a főablak kijelölése után a **Property Editor**-ban megkeressük és beállítjuk a **windowIcon** tulajdonságot. A lefelé mutató nyílra kattintva felbukkanó menücskből a **Choose Resource** menüpontot kell választanunk. A program fordítását és futtatását követően a futó programokat felsoroló helyeken – tálca, **Alt+Tab**, miegymás – már a szép új ikonunk látszik:

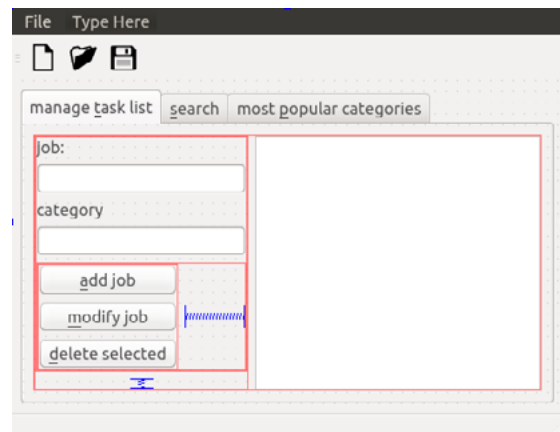
A futtatható állomány az operációs rendszer fájlkezelőjében látható ikonjának beállítása szerencsére platformspecifikus, így itt most nem foglalkozunk vele¹⁸.

A Fájl menü menüpontjainak ikonjaival folytatjuk a munkát. Ha még emlékszünk (és amennyiben történetesen nem emlékeznénk, akkor is), a létrehozott menüpontok **QAction** osztályú objektumok. Így aztán a felhasználóifelület-szerkesztőben előhívható **Action Editor** részen láthatók – ha nem volna ilyen rész a monitorunkon, a **Window** menü **Views** almenüjén kell elhelyeznünk egy pipát.



14. ábra: Hörcsög a mókuserékben - kell-e kifejezőbb ikon egy tennivalólista-alkalmazásnak?

Az **Action Editor**-on belül kattintsunk jobb egérgombbal valamelyik soron, és a forrásfájlból válasszuk ki a hozzá való ikont. Ezzel a menüben már látszik is a művünk, de előbb még tegyünk meg annyit, hogy a menüsor (**menuBar**) alatt, de minden más fölött megjelenő, helyesebben bujkáló eszköztárra (**toolBar**) húzzuk ki egérrel az ikonnal is bíró **QAction**-öket. Ilyen lesz a kép:



15. ábra: A QAction-ök beköltöztek az eszköztárra (is).

Mit is műveltünk? A **QAction** osztályú objektumaink most egyszerre két helyen jelennek meg: a menüben és az eszköztáron. Ez a két megjelenés egy és ugyanaz az objektum, a hozzá kapcsolódó slot is egy maradt. Nem kell két helyen figyelni rá, ha valamit változtatunk. Baró.

Programunk többnyelvűsítése óta most először fogunk új karakterláncokat felvenni, úgyhogy bevezetünk egy új eljárást is: itt, a könyvben angolul írjuk a karakterláncot, majd megadjuk a magyar fordítását is, ha fordítandó. A fordítási eljárás persze elolvasható az előző fejezetben, de azért még egyszer, röviden: **lupdate**, hogy az új karakterláncok átkerüljenek a **ts**-fájlba (ha többféle fordítás is készül, akkor fájlalba), fordítás a Qt Linguist-ben, majd **lrelease**, hogy előálljon a **qm**-fájl, amit szükség szerint a **qm**-fájl másolása követ a futtatható állomány könyvtárába.

Hozzunk létre új menüt Others (Egyebek) néven, majd helyezzünk el benne két menüpontot Settings (Beállítások) és About (Névjegy) szöveggel. Az **actionSettings** objektumnak állítsuk be az ikonját, majd húzzuk az eszköztárra. Készítsük el a slot-ját mindkettőnek, a **triggered()** signal-ra.

Jöhet a következő alfejezet.

18 De aki nem nyugszik nélküle, annak tessék: <http://qt-project.org/doc/qt-4.8/appicon.html>

9.2. Módos és egyéb párbeszédablakok, a QTextBrowser és újabb adalékok az erőforrásfájlok természetéhez

A névjegy ablakának megvalósításával folytatjuk kiseded programunk fejlesztését. Új osztályt készítünk az ablaknak, de nem a sima C++-osztálysablonok közül válogatunk, hanem a Qt csoporton belüli **Qt Designer Form Class** lehetőséget választjuk. A megjelenő párbeszédablakból pedig a **Dialog without Buttons** válik szimpatikussá. Az osztály neve legyen **AboutWindow**. A **mainwindow.h** fájl fejlécei között helyezzük el az **aboutwindow.h** állományt, majd készítsünk az ablakunknak privát mutatót:

```
AboutWindow *aboutWindow;
```

Az About menüpont slot-jának törzsében példányosítsuk az objektumot, ahova a mutató mutatni fog:

```
aboutWindow = new AboutWindow(this);
```

Ha most futtatjuk a programunkat, akkor a megfelelő kattintgatásokat követően létrejön az ablak, épp csak nem látszik. Már az ókori építészek sem szerették a láthatatlan ablakokat, és ezzel lényegében mi sem vagyunk másképp. Mutassuk hát meg!

Előbb azonban van némi megfontolnivalónk. Az angolul beszélő okosok szerint egy ablak vagy *modal* (móddal bíró, modális, „módos”), vagy *modeless*, más szóval *non-modal* (mód nélküli)¹⁹.

Az előbbi csoportba tartozó ablakok felbukkanásuk után lehetetlenné teszik a felhasználó számára a program más ablakaival való kommunikációt: adatbevitelt, gombok nyomkodását, miegymást. Sőt, megkülönböztethetünk ablakmodális és alkalmazás-modális ablakokat: az első esetben csak a szülőablak felé gátolt a felhasználói interakció, a második esetben az alkalmazás valamennyi egyéb ablaka felé. Ilyen ablakokat akkor használunk, ha valami elképesztően fontosat akarunk megtudni a felhasználótól, vagy tutibiztosak akarunk lenni abban, hogy a felhasználó megértette az elképesztően fontos üzenetünket.

A *modeless* ablakok pedig, legyen bármilyen meglepő, de épp az ellentétei az előző csoportba tartozóknak, azaz felőlük nyugodtan foglalkozhatunk a többi ablakkal, ahogy épp úri kedvünk diktálja. Tipikusan ilyen a szövegszerkesztőnk helyesírás-ellenőrző ablaka, ami felhívja a figyelmünket a helyesírási hibára, és lehetővé teszi, hogy az eredeti szövegben, a másik ablakban javítgassuk a szöveget.

A Qt-ban több módszerrel is elérhető egyik vagy másik viselkedés, és ebben a strandkönyvben természetesen a legegyszerűbb esetet ismertetjük²⁰. Aki a ToDoList névjegyét *modal* ablakként nyitná meg, a slot törzsének második soraként használja a

```
aboutWindow->exec();
```

sort. A többiek pedig úgy jutnak *modeless* ablakhoz, hogy inkább a

```
aboutWindow->show();
```

¹⁹ Hogy miért pont így hívják őket, az a következő magyarázatból nem fog kiderülni, de némi Wikipedia-olvasgatással képet kaphatunk a dologról. Akit érdekel a dolog háttere, annak ajánlott a http://en.wikipedia.org/wiki/Modal_window és a http://en.wikipedia.org/wiki/Mode_error#Mode_errors hivatkozásokon található szövegek áttanulmányozása.

²⁰ A témában elmélyedni vágyók kedvéért persze itt egy jó kis hivatkozás: <https://qt-project.org/doc/qt-5.1/qtwidgets/qdialog.html#details>

utasítást adják ki. Esetünkben lényegében mindegy, hogy melyik utat választjuk, de mielőtt döntenénk, próbáljuk ki mindkét lehetőséget.

Kisebb elmélkedés következik, amit végigkövetve talán jobban megértjük a két viselkedésmód mélységeit, ráadásul kiszűrünk egy ronda programhibát is. Most ugyebár mutatót adtunk meg az ablakunknak, és a heap-en hoztuk létre az ablakot, azaz a létrehozott ablak életben marad a `MainWindow::on_actionAbout_triggered()` slot lefutását követően is.

Megtehetjük volna azonban azt is, hogy az ablakot a slot-ban helyi hatáskörű (angolul: local scope) változóba példányosítsuk az

```
AboutWindow aboutWindow;
```

sorral. Ha ekkor az

```
aboutWindow.exec();
```

sorral nyitjuk meg az ablakot, akkor a program szépen megvárja, amíg be nem záródik az ablak – a slot futása addig *felfüggesztődik* –, és minekutána az ablak bezárul, a program eléri a slot végét, és megsemmisülnek a helyi objektumok, így az ablakunk is. A felhasználó a változából semmit sem vesz észre.

Amennyiben az ablak megjelenítésére az

```
aboutWindow.show();
```

módszert választjuk, a felhasználó majd panaszkodik, hogy nincs névjegy-ablak. Pedig van, csak épp az utasításunkat követve a slot vigyorgva tovább fut, és jó eséllyel még azelőtt véget ér, hogy az ablaknak ideje lenne megnyílni.

Elmélkedésünk első tanulsága tehát, hogy *modeless* ablakot okos fejlesztő a heap-en hoz létre.

A fenti mondatból az következik, hogy lesz még tanulság. Valóban: nekilátunk a programozási hiba elhárításának.

Mi is történik, amikor a slot lefut? Létrehozunk egy `AboutWindow` osztályú ablakot, a memóriacímét értékül adjuk az `aboutWindow` mutatónak, majd megmutatjuk az ablakot. Slot vége. Persze tudjuk, hogy nem kell szöszölnünk az ablak objektumának megsemmisítésével, hiszen a `this` szülő miatt a Qt majd takarít helyettünk. Igen ám, de ha a felhasználó *újra* megnézné a névjegyet, akkor lesz még egy ilyen ablakunk, ráadásul az előzőnek elfelejtjük a címét. Az előző névjegy-ablak így aztán ott fog téblábolni a memóriában, majdnem mindenkitől elfeledve, amíg véget nem ér az alkalmazásunk futása. Erre meg semmi szükség, nem igaz? (A jelenség személtetésére a példányosítást végző sor után helyezzük el a `qDebug()` << `aboutWindow;` sort. Valahányszor megnyitjuk a névjegyet, mindig más memóriacímet látunk majd.)

Valahogy jó volna ellenőrizni, hogy a mutatónk null-pointer, vagy sem, mert ha az, akkor még nincs ilyen ablakunk, ha meg nem az, akkor már van. A legokosabb, ha először például a főablak konstruktorában az

```
aboutWindow = 0;
```

sorral NULL értékre inicializáljuk a mutatót – közben megállapíthatjuk, hogy a konstruktor mostanra csúnyán elhízott. Ezek után a slot-ot az alábbi formában véglegesítjük:

```
void MainWindow::on_actionAbout_triggered()
{
    if(!aboutWindow)
        aboutWindow = new AboutWindow(this);
    aboutWindow->show();
}
```

Így a slot nem készít mindig új ablakot, csak újra meg újra megmutatja nekünk a régit. Ha az ablakon már lesz mit változtatni, akkor még a változás is megmarad majd két megmutatás között.

Elmélkedésünk második tanulsága tehát az, hogy vigyáznunk kell, nehogy ellepjenek bennünket az ablakok, kivéve, ha épp ez a cél.

Még egy utolsó megjegyzés (aztán befejezzük az elmélkedést, és végre nekilátunk a névjegyablak benépesítésének): ha a felhasználó esetleg csak háttérbe tette a *modeless* ablakot, akkor könnyen elfelejtkezik róla, és nem érti, hogy a menüből megnyitva miért nem válik láthatóvá. Ilyen esetek kezelésre érdemes megismerkednünk a `QWidget::raise()` slot-tal és a `QWidget::activateWindow()` tagfüggvénnyel.

Nos, most, hogy megnyitható a névjegyablak, kutyaoljunk át az `aboutwindow.ui` fájlra. Az ablak címe (`windowTitle` tulajdonság a *Property Editor*-ban) legyen About ToDoList (A ToDoList névjegye). Megnyílik a grafikus szerkesztő. Az ablak aljára helyezzünk el egy `QPushButton`-t, és a feliratát állítsuk „I just got smarter” (Okosabb lettem) értékre.

A nyomógomb fölé pedig húzzunk ki egy `QTextBrowser`-t, azaz szövegnézegetőt. A `QTextBrowser` osztály a `QTextEditor` (szövegszerkesztő) leszármazottja. A szöveget csak megjeleníteni tudja – ilyen értelemben le van butítva –, viszont extrát is tud: lehet a benne lévő hiperszövegben navigálni.

A `textBrowser` objektum létrejöttét követően állítsunk be elrendezéseket, hogy az átméreteződések szépen menjenek, majd gondolkodjunk el, miféle okosságokat írunk majd ide. A könyv webhelyéről letölthető verzió ékes angol nyelven ír pár sort arról, hogy mi is ez az alkalmazás, majd a licenceket kezdi taglalni, aminek több jó oldala is van: részint a felhasznált ikonok licence teszi ezt kötelezővé, részint lesz okunk képet és hivatkozást is elhelyezni. A Qt Creator-ban a `textBrowser`-re kattintva meg is szerkeszthetnénk a tartalmát – így kapnánk egy statikus, bedrótzott szöveget –, de az túl egyszerű volna.

Kicsit álmodozunk: tételezzük fel, hogy a projektünk már nagy és nemzetközi, és a nyílt forráskódnak köszönhetően önkéntesek hada dolgozik rajta. A hozzájárulók száma verzióról verzióra nő, és már a kutya nem győzi a névsoruk frissítését. Szerencsére az intelligens verziókezelő, meg a scriptjeink mindig előállítják a névsort, de semmi kedvünk a másolás-beillesztgetéshez minden fordítás előtt és bokros teendőink megakadályoznak abban, hogy erre külön mechanizmust implementáljunk. Nos, a Qt itt is segít – tiszta jó fej, nem?

A `QTextBrowser` osztályú objektumok forrásául ugyanis megadható egy némileg csupasz, HTML4 nyelvű forrásfájl. Csupasz alatt azt értjük, hogy az elejéről le hagyjuk a `!DOCTYPE` részt (a Qt majd odateszi), lényegében csak a `<html>` és a `</html>` címkét, illetve a közöttük lévő részt adjuk meg benne. Ráadásul nem is használhatjuk a teljes nyelvi készletet, de azért ellesszünk ennyivel²¹. Ha ezt a `html`-fájlt elhelyezzük az erőforrások között (pontosan úgy, mint az ikonokat, lásd az előző alfejezetben), akkor a `textBrowser` objektumot kiválasztva,

21 Pontos lista: <http://qt-project.org/doc/qt-4.8/richtext-html-subset.html>

a **Property Editor**-ban a **source** tulajdonságnál megadható ez a fájl. Az ikonokhoz hasonlóan a **html**-fájlt is azért érdemes az erőforrások közé helyezni, mert így nincs többé gondunk arra, hogy a futtatható fájljal együtt másolgassuk.

Nos, hogy ez az eljárás miként segíti a népes fejlesztőgárda névsorának napra készen tartását az eljövendő nagy projektünkben? Hát, ha a **html**-fájl előállítás megoldott, akkor a fordítások során mindig az épp aktuális változat kerül a programunkba. Remeg dolog, próbáljuk is ki! Készítsünk egy **html**-fájlt, helyezzük el az erőforrások között, majd adjuk meg a **textBrowser** objektum forrásaként. Meg is jelenik a tartalma, már a felhasználóifelület-szerkesztőben is látjuk. Ha módosítjuk a fájlt, és szeretnénk megnézni a változást, nos az nem látszik (legalábbis a szerző Qt Creatorában nem), még akkor sem, ha újra beállítjuk a **textBrowser** forrásaként. Mondjuk ez nem olyan kardinális probléma mert a program fordítását és futtatását követően az *igazi* névjegy már tükrözi a változásokat, és ez az, ami fontos.

Ha az a vágyunk, hogy a **html**-fájlban lévő hivatkozásokat egérgattintásra megnyissa az alapértelmezett böngésző, akkor a **Property Editor**-ban tegyünk pipát az **openExternalLinks** tulajdonság mellé.

Megmutattuk már a hörcsögnek, hogy rajta van az ikonon? És legalább megismerte magát?

Ja, még valamit. Ha kevés a **QTextBrowser**, illetve a **QTextEditor** osztály HTML-ismerete, akkor használhatjuk épp a **QWebView** osztályt is, amely lényegében böngészőt varázsol a Qt-alkalmazásunkba. Az előálló böngésző a használt Qt-verziótól függően vagy a WebKit-re, vagy a Chromium-féle motorra épül. A Qt ugyanis nagyjából e fejezet elkészültével egy időben váltott az előbbiről az utóbbira²². Persze egy kis névjegy megjelenítéséhez erős túlzás lett volna egy komplett böngészőt indítani a háttérben – ha napjaink személyi számítógépeinél ez nem is annyira szempont, vannak mobileszközök, ahol erősen az.

9.3. Automatikus mentés a QTimer osztály használatával

Elsőként a Beállítások ablakot készítjük el. Lényegében úgy dolgozunk, ahogy az iménti ablaknál: új osztályt hozunk létre a párbeszédablaknak **SettingsWindow** néven, és az **on_actionSettings_triggered()** slot-ban megjelenítjük a **settingsWindow** nevű példányt, mégpedig az **exec()** tagfüggvény futtatásával, azaz *modal* ablakként. A különbség annyi, hogy ezt az ablakot a *stack-en*, azaz a *slot-on* belül példányosítjuk.

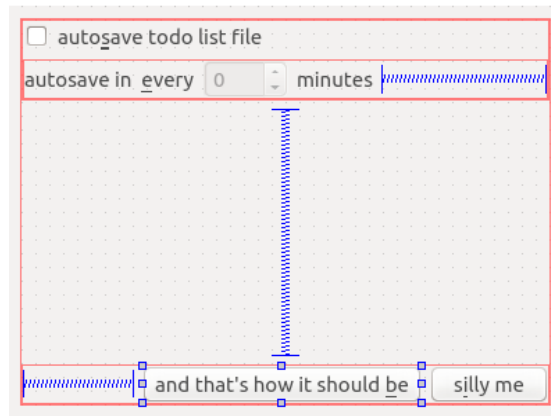
Mire is lesz jó ez a beállításablak? Hát arra, hogy megadhassuk: akarunk-e automatikus mentést, és ha igen, hány percenként.

A felhasználóifelület-tervezőben állítsuk be az ablak címét. Helyezzünk el az ablakon egy **QCheckBox**-ot. Ebbe tesszük a pipát, ha kérünk automatikus mentést, úgyhogy a neve legyen „autosave todo list file” (feladatlistafájl automatikus mentése). Kell még egy **QSpinBox** a percek számának megadására. Állítsuk be a **PropertyEditor**-ban a **maximum** tulajdonságát 60-ra, a **singleStep** tulajdonságát 5-re (ennyivel fogja léptetni a nyilacska az értéket), továbbá alapértelmezetten legyen kikapcsolva, azaz az **enabled** tulajdonság mellől vegyük ki a pipát. Tegyünk a **QSpinBox** elé és mögé egy-egy **QLabel**-t, „autosave in every” (automatikus mentés), illetve „minutes” (percenként) felirattal. Az elsőnél adjuk meg *buddy*-ként a **QSpinBox**-ot. Helyezzünk még el két nyomógombot. Az egyik neve legyen **buttonOk**, a másiké **buttonCancel**, a feliratuk pedig „and that’s how it should be” (és legyen így), illetve „silly me” (butuska voltam). Állítsunk be gyorsbillentyűket és készítsünk elrendezést. Ha van kedvünk, a

²² <http://blog.qt.digia.com/blog/2013/09/12/introducing-the-qt-webengine/>

fordítást is elkészíthetjük – legalább látjuk, hogy a Qt nem akad fenn azon, hogy ha magyarul más hosszúságú egy szöveg, mint angolul.

A következő dolgunk némi kattintgatással slot-ot készíteni a `checkBox` objektum `stateChanged()` signal-jához. A slot törzsét úgy akarjuk elkészíteni, hogy ha a `checkBox`-ba pipát tesz a felhasználó, akkor bekapcsoljuk a `spinBox` objektumot, ha meg kiveszi a pipát, akkor kikapcsoljuk. Íme a slot a maga teljes szépségében:



16. ábra: A beállításlablak elrendezése

```
void SettingsWindow::on_checkBox_stateChanged(int arg1)
{
    if(arg1 == Qt::Unchecked)
        ui->spinBox->setEnabled(false);
    else if(arg1 == Qt::Checked)
        ui->spinBox->setEnabled(true);
}
```

Az egész típusú `arg1` változó megmondja, hogy milyenre változott a `checkBox` állapota. Valójában elvileg három eset lehetséges: pipálatlan, részlegesen pipált és pipált. A három állapotnak rendere a 0, 1 és 2 szám, illetve a `Qt::Unchecked`, a `Qt::PartiallyChecked` és a `Qt::Checked` nevesített konstans (angolul: enum) felel meg. Ha az egyszeri programozó csodálkozik, hogy valami hogy lehet részlegesen pipálva, akkor megjegyezzük, hogy vannak esetek, amikor a beállítás alatti hierarchiaszint további beállításai között egyes értékek pipálva vannak, mások nem. Az ilyesmit halovány pipa jelzi grafikusán. Szerencsére piciny programunkban ilyen borzadályos eset nem fordul elő.

Következő dolgunk a két nyomógomb slot-jainak megírása. A `buttonCancel` gomb `clicked()` signal-ját a `settingsWindow` ablak `close()` slot-jához kötjük. Régen műveltünk már ilyet, viszont akkor teljesen grafikusán tettük – ha már elfelejtettük volna a dolog mikéntjét, lapozzuk fel a második fejezetet, azon belül is „A signals and slots mechanizmus” című részt, és informálódjunk.

A `buttonOk` slot-ja már húzósabb téma: itt az ideje a beállítások tárolásának. Úgy döntünk, hogy ismét használatba vesszük a `QSettings` osztályt. Ha még emlékszünk, ez úgy megy, hogy ha beállítottuk a cég nevét, internetes címét és az alkalmazás nevét – márpedig mi beállítottuk –, akkor a helyben példányosított `QSettings` osztályú objektum azonnal tudja, hol kell tárolni a beállításainkat. Így aztán tároljuk is őket. A tárolás végeztével meg ugye csak be kell zárni az ablakunkat. Mindez együtt így néz ki:

```
void SettingsWindow::on_buttonOk_clicked()
{
    QSettings settings;
    settings.setValue("GeneralSettings/AutoSaveEnabled", ui->checkBox
->isChecked());
    settings.setValue("GeneralSettings/AutoSaveMinutes", ui->spinBox
->value());
    this->close();
}
```

Nem vagyunk még kész a slot-tal, mert nem szoltunk még a főablaknak, hogy mostantól mentegessen – vagy épp ne tegye –, de ezt egyelőre elhalasztjuk. Inkább azzal törődünk, hogy ha újranyitjuk az beállításablakot, akkor olvassa be, és jelenítse meg a beállításokat. Sejtjük már, hogy akkor megint kell egy `QSettings` osztályú objektum. Van értelme a konstruktorban elhelyezni, mégpedig a beolvasó-megjelenítő utasításokkal együtt. Úgyhogy a `SettingsWindow` osztály konstruktora az alábbi három sorral bővül:

```
QSettings settings;  
ui->checkBox->setChecked(settings.value("GeneralSettings/  
AutoSaveEnabled", "false").toBool());  
ui->spinBox->setValue(settings.value("GeneralSettings/  
AutoSaveMinutes", 0).toInt());
```

A `toBool()` és a `toInt()` tagfüggvény már ismerősnek tűnik. A `QSettings` osztály `value()` tagfüggvényének visszatérési értékei ugyebár `QVariant` típusúak, amiről korábban már megemlítettük, hogy lényegében a „mittoménmiez” típust azonosítjuk vele. A `QVariant` osztály persze kínál olyan függvényeket, amelyekkel a benne tárolt értékeket emberi típusokra tudjuk alakítani: a `toBool()` és a `toInt()` függvény épp ilyen.

Már csak az a kérdés, mik a `QSettings::value()` tagfüggvény vessző utáni paraméterei, nevezetesen a „false”, illetve a 0? Nos, ezek alapértelmezett értékek. Ha a beállítások között nincs ilyen beállítás, akkor a `settings` objektumunk ezeket adja vissza. Így megspórolható egy kör, amiben a beállítási kulcs jelenlétét ellenőriznénk a `QSettings::contains()` tagfüggvénnyel.

Eddigre talán szöveget ütött a fejünkbe, hogy ha már úgyis kétszer példányosítunk magunknak `QSettings` osztályú objektumot, nem volna-e értelme a stack helyett a heap-en elhelyezni, `this` szülővel, hogy az ablak megsemmisülésével ez az objektum is megsemmisüljön? De, alighanem volna, úgyhogy tegyük is meg. Ne feledjük a `value()` és a `setValue()` tagfüggvény-hívások előtti pontot nyílra cserélni, hiszen a `settings` mostantól mutatóként él az osztályban. Nem mernénk arra nagyobb összegben fogadni, hogy a programunk ezzel – minden platformon – gyorsabb is lett, de átláthatóbb igen, és ugye az a mai világban a legtöbb esetben fontosabb, mint a sebesség.

Akkor hát visszatérhetünk ahhoz a problémához, hogy a főablakot is értesíteni kellene a helyzet változásáról, ha a felhasználó úgy dönt, hogy az új beállítások jók.

Ugye megint az a helyzet, hogy a gyerekobjektum szól a szülőhöz, azt meg úgy érdemes neki, hogy signal-t emittál, amit a szülő slot-ja elkap majd.

Vagy mégsem? Nem annyira, ugyanis a `QObject::connect()` tagfüggvény a küldő objektum mutatóját szeretné megkapni, nekünk meg helyi objektumunk van, mutatónk nincs hozzá. Ha nagyon akarunk, persze tudunk hozzá mutatót szerezni, de van egyszerűbb módszer: visszaadjuk, hogy érvényesíteni akarja-e a felhasználó a változtatásait, vagy sem: a `buttonOk`, vagy a `buttonCancel` gombot nyomta-e le.

Mindezt úgy valósítjuk meg, hogy a `SettingsWindow::on_buttonOk_clicked()` slot utolsó sorát másikra cseréljük:

```
this->done(QDialog::Accepted);
```

Ez a sor is bezárja az ablakot, a vezérlés visszakerül a `MainWindow::on_actionSettings_triggered()` tagfüggvényhez. A `close()` tagfüggvény használatához képest az a különbség, hogy itt megmondható, hogy végül is milyen eredménnyel zárult be az ablak.

A `QDialog::Accepted` megint csak egy *enum*, az értéke 1. Lényegében csak azért nem azt írjuk oda, hogy 1, mert így jobban olvasható marad a kód.

Így a Beállítások ablakot megnyitó tagfüggvényben már van értelme vizsgálni a `settingsWindow.exec()` utasítás kimenetét, hiszen az végre nem minden esetben 0. Az `on_actionSettings_triggered()` slot belsejébe tehát írhatnánk, hogy

```
if(settingsWindow.exec())
    tegyünk0kosat();
```

mi azonban, megint csak a jobb olvashatóság kedvéért a

```
if(settingsWindow.exec() == QDialog::Accepted)
    tegyünk0kosat();
```

formát választjuk. Már csak az a kérdés, hogy mi legyen az az okosság? Sok választási lehetőségünk van, de figyelembe kell vennünk, hogy akkor is be kell olvasnunk a beállításokat, azaz úgyis kellene egy privát tagfüggvény, ami beolvassa ezt a két beállítást, és ami a program főablakának konstruktorából hívható.

Mit is kell majd tennie pontosan ennek a tagfüggvénynek? Beolvasni a két beállítást, eddig fix. Kelleni fog hozzá egy `QSettings` osztályú objektum. Minthogy a főablakban máshol úgy sem használjuk, meg hát annyira sokszor elvileg nem fog kelleni (a legtöbb esetben csak a program indításakor), létrehozhatjuk a tagfüggvény stack-jén is. És mi lesz a beolvasott adatokkal?

Most vesszük elő az alfejezet címében beígért `QTimer` osztályt. Az ebből az osztályból példányosított objektumok azt tudják, hogy a beállított idő leteltével, illetve beállított időközönként (azaz ismétlődően) emittálnak egy szép kövér `timeout()` signal-t, magyarul elordítják magukat, hogy „Időőőőő!”. Nos, a tagfüggvényünk az időzítés beállítását végzi majd el.

A `mainwindow.h` fájl fejlécei közé vegyük fel a `<QTimer>` nevűt is, majd hozzunk létre egy `QTimer` osztályú objektumra mutató mutatót `timer` néven, a főablak konstruktorában (Említettük már, hogy *nagyon* hízik?) példányosítsunk hozzá objektumot `timer` néven – ne felejtsük el a szülő (`this`) beállításáról. Ha szeretnénk, a `qDebug()` `<< timer->isSingleShot();` utasítással megbizonyosodhatunk róla, hogy a létrejött objektum ismétlődően ordítózik majd, ha letelik az idő.

Deklaráljuk végre a privát függvényünket:

```
void loadAutoSaveSettings();
```

formában. A törzset pedig alakítsuk ilyenre:

```
QSettings settings;
if(settings.value("GeneralSettings/AutoSaveEnabled", "false").
toBool())
    timer->start(settings.value("GeneralSettings/AutoSaveMinutes").
toInt()*60000);
else
    timer->stop();
```

Létrehozunk benne egy `settings` objektumot, ami beolvassa, hogy kell-e automatikusan menteni (ha nincs ilyen beállítás, akkor is `"false"` értékkel tér vissza. Ha kell, akkor beolvassuk azt is, hogy hány percenként kell. Ezúttal nem mondjuk meg, mi a teendő, ha nincs ilyen beállítás, ugyanis, ha csak kézzel nem piszkáljuk a beállításokat, akkor vagy mindkettő

beállítás van, vagy egyik sem. A percek számát megszorozzuk összesen 60000-rel, merthogy a `timer` objektum ezredmásodpercekben gondolkodik. A kapott számmal elindítjuk a visszaszámlálás-sorozatot. Ha nem kell automatikus mentés, akkor számítunk rá, hogy eddig esetleg be volt állítva, csak épp most kapcsolta ki a felhasználó, így leállítjuk az időzítést. Ha eddig sem ment, akkor sem lesz baj a dologból.

A kész tagfüggvényt hívunk kell a főablak konstruktorán kívül az `on_actionSettings_triggered()` slot-ból is, onnan, ahol az imént még csak annyit tudtunk, hogy valami okosat kell tennünk.

Odáig jutottunk hát, hogy van egy szép és pontos `timer` objektumunk, amely pontosan jelzi, ha itt az „Időőőőő!”, de a franc sem figyel rá. A megoldás persze megint a főablak konstruktorát hszlalja:

```
connect(timer, SIGNAL(timeout()),  
        this, SLOT(on_actionSave_triggered()));
```

Kisebb probléma, hogy a slot sajátosságai miatt, ha még nem volt mentve a fájl, akkor az idő lejártával kapunk az arcunkba egy kérdést a leendő fájlnevről. Ezzel azonban nagyvonalúan nem törődünk.

Itt az „Időőőőő!”, hogy kitakarítsuk a hörcsög terráriumát, a következő fejezetben pedig a magunk háza táján takarítunk.

10. KOMOLYAN HASZNÁLNI KEZDJÜK A MODEL-VIEW MINTA LEHETŐSÉGEIT

Ha végeztünk a hörcsög terráriumának kitakarításával, újra átgondoljuk, hogy mit is tudunk a Model-View (magyarul: modell-nézet) mintáról. Van ugye egy nézetünk, ami nem más, mint egy Qt-objektum. Meg van egy modellünk, ami egy másik Qt-objektum. A két objektumot összekapcsoljuk, mégpedig úgy, hogy a nézet tudja, hogy ki az ő modellje, a modellnek viszont halvány lila gőze nincs arról, hogy ki az ő nézete. Sőt – ilyesmit még nem műveltünk, de akár művelhettünk is volna – egyetlen modellhez több nézetet is társíthatunk.

És itt a lényeg: a modell változásairól a nézet értesül, és a változásoknak megfelelően *frissíti magát*, mindezt anélkül, hogy akár mi, akár a hörcsög egyetlen mozdulatot tennénk az ügy érdekében.

Eddig két esetben használtunk modellt. Az egyik esetben egy `QStringListModel` objektumot használtunk az automatikus kiegészítés működtetésére, mégpedig úgy, hogy a `QCompleter` osztályú objektumunknak megmondtuk, hogy melyik `QStringListModel` az ő modellje, azaz honnan vegye a tippjeit az automatikus kiegészítéshez. Amikor ezek után kedvünk szottyant, módosítottuk a modellt, és a `QCompleter` osztályú objektumunk erről értesült, pedig nem szóltunk neki. Azzal ugyanis, hogy a `QCompleter::setModel()` tagfüggvény hívásával megmutattuk neki, hogy melyik objektum a modellje, ráruháztuk azt a feladatot és felelősséget is, hogy a modell változásait figyelje.

A másik esetben egy `QGraphicsView` osztályú objektumot használtunk nézetként, a modellül pedig egy `QGraphicsScene` osztályú objektum szolgált. Ez esetben a nézet nevében is benne van, hogy ő egy View, azaz nézet. A grafikonunk oszlopait, sávjait a színen, a `QGraphicsScene` osztályú objektumban helyeztük el, és sosem szóltunk a nézetnek, hogy mit ügködünk a színen, mert mindig a nézet dolga erre figyelni.

A két fenti esetben azonban épp csak érintettük a Modell-Nézet minta használatában rejlő lehetőségeket, ugyanis a programunk lényegi része, nevezetesen a teendő-kategória párosokból álló feladatlista mindezidáig nem került modellbe, minden változtatás esetén kézzel ügködtünk annak érdekében, hogy a változás látszódjék is. Na, majd most.

10.1. Elem-alapú és modell-alapú widget-ek

Ebben a rövidke alfejezetben megismerkedünk az átalakítás során használt egyik legfontosabb widget-ünkkel, és bemutatjuk a testvéreit: a hűgát és a nővérét.

Kicsit zavaró ebben a magyar nyelvű könyvben, hogy legalább két szakkifejezést fordítunk az elem szóval. Az egyik kifejezés a widget, azaz a grafikus felhasználói felület *eleme*. Fordíthatnánk épp bigyónak is, de valahogy úgy alakult, hogy a magyar szaknyelvben nem honosodott meg a szó eredetét (window + gadget) tekintve helyes, de komolytalanul hangzó ablakműtűr, ablakbigyó szó. Más programozási környezetekben a widget szóval nagyjából egyenértékű control (magyarul: vezérlőelem, vagy röviden vezérlő) szó terjedt el, de ebben a strandkönyvben pont a Qt-ről van szó, ott meg widget-ek vannak. A másik szó pedig az item, azaz valamilyen sornak, adatszerkezetnek az *eleme*.

E hosszabb kitérő után lássuk a widgeteinket. Ha a Qt Creator grafikus felhasználói felület-szerkesztőjének bal oldalán név szerint szűrünk, mondjuk a *list* szóval, akkor két widget-et találunk:

Az egyik az elem-alapú `QListWidget`, a másik a modell-alapú `QListView`. Figyeljünk a widget és a view névre. Mindkettő widget (mert persze mind a kettő widget, azaz grafikus elem) arra való, hogy listát jelenítsünk meg vele, de az egyik esetben elemenként, azaz itemenként pakolgatjuk a megjelenítendő elemeket, a másikban az elemek egy modellbe kerülnek, és a `QListView` nevű widget tulajdonságai között megadjuk a modellt.

Szűrjünk most a *table* szóra!

Megint ketten vannak. És, ahogy már sejtjük, az egyik megint elem-alapú, a másik meg modell-alapú. Ezt az elemet, a `QTableView` nevű widget-et fogjuk a közeljövőben használatba venni. A hűgával, a `QListView` nézettel ellentétben ebben az elemben nem egy, hanem kétdimenziós adatszerkezeteket jelenítünk majd meg. Már csak a nővér – és az ő ellentétpárja – van hátra a nagy bemutatkozásból. Szűrjünk a *tree* (fa) szóra!

A táblákkal ebben a könyvben nem foglalkozunk. Ők az igazi nagygagyúk: akárhány dimenziós adatszerkezetek megjelenítésére jók, amennyiben minden adat elhelyezhető egy hierarchiában, minden adatnak megadható a szülőeleme. A szülő-gyerek kapcsolatok mentén fává alakítható az adatszerkezet, így oldják meg a sokdimenziós struktúra kétdimenziós kijelzőn való megjelenítését.

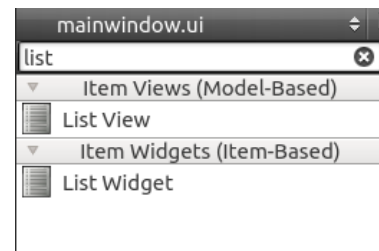
10.2. Újratervezés

A következő néhány fejezetben a programunkat jószerével az elejéről kezdve újraírjuk, aminek – lássuk be – talán legfőbb ideje volt már. Persze ilyesmit profi Qt-programozó²³ nem tesz – na de profi Qt-programozó nem is olvasgat Qt-strandkönyveket. Mondhatnánk, hogy gondolkodhattunk volna már korábban is, meg hogy így kellett volna már a legelejétől megírni – igen ám, de a könyv elején lényegesen kevesebbet tudtunk a Qt-ről, mint most. Talán máris sejtjük, hogy a mostani változat sem lesz végleges, de megnyugtattunk mindenkit: ha ezzel a változattal is készen vagyunk, már csak egy markáns átalakítás marad hátra, és arra már csak a könyv vége felé kerítünk sor. És még az a változat is messze áll majd a tökéletestől. Az a helyzet, hogy egy program ugye ritkán van kész – a fejlesztési folyamat során legfeljebb a *kiadásra érett* állapotig juthatunk el, a „Helló világ” programok bonyolultságát meghaladó projekteken mindig van mit javítani. Úgyhogy e felett ne is búslakodjunk, fogadjuk el, hogy így van és kész.

10.3. Mit tartunk meg az előző változatból?

Nem sok mindent. Kezdjünk új projektet, majd másoljuk át a projekt mappájába az *about* és az *icons* mappáinkat – igen, tartalommal együtt. Jöhet még az `aboutwindow.cpp`, az

23 Meg ugye semmilyen más nyelven programozó profi programozó sem tesz ilyet. Kivéve, ha a programozó ugyan profi, de nem azon a nyelven, amelyiken épp programot ír.



17. ábra: Listák megjelenítésére alkalmas elemek



18. ábra: Táblázatok megjelenítésére alkalmas elemek



19. ábra: Bonyolultabb adatszerkezetek megjelenítésére alkalmas elemek

`aboutwindow.h`, `aboutwindow.ui` és a `ToDoList.qrc`. Vegyük is fel őket a Qt Creator-ban is a projekt fájljai közé (nem elég, ha benne vannak a mappában), valamint helyezzük el a fordításfájl (-fájlok) sorát a projektfájlban – ha elfelejtettük már a módját, lapozzunk vissza. Maguk a fordításfájlok nem kellenek, majd létrehozzuk őket újra.

10.4. A főablak kialakítása

A főablak képének felhasználásával alakítsuk a magunkét is ilyenre. Alább következnek az egyes objektumok nevei is, amelyeket azért érdemes így beállítanunk, mert a programunk is ezt használja majd.

A gombok (`QPushButton` osztály) objektumnevei (zárójelben a feliratok egy-egy lehetséges magyar fordítása):

- `buttonDelete` (feladat törlése)
- `buttonUndo` (utolsó törlés visszavonása)
- `buttonNewAfter` (új a kijelölt után)
- `buttonNewBefore` (új a kijelölt előtt)
- `buttonUpSelected` (felfelé a kijelöltöt)
- `buttonDownSelected` (lefelé a kijelöltöt)

A középen lévő nagy fehér semmi egy `QTableView`, aminek a fantáziadús `tableView` nevet adtuk. A File menüben lévő `QAction`-osztályú objektumok neve és felirata (zárójelben a magyar fordítás):

- `actionNew`, New (Új)
- `actionOpen`, Open (Megnyitás)
- `actionSave`, Save (Mentés)
- `actionSave_as`, Save as... (Mentés másként)

Az `Others` (Egyebek) menüben lévő két `QAction`:

- `actionSettings`, Settings (Beállítások)
- `actionAbout`, About (Névjegy)

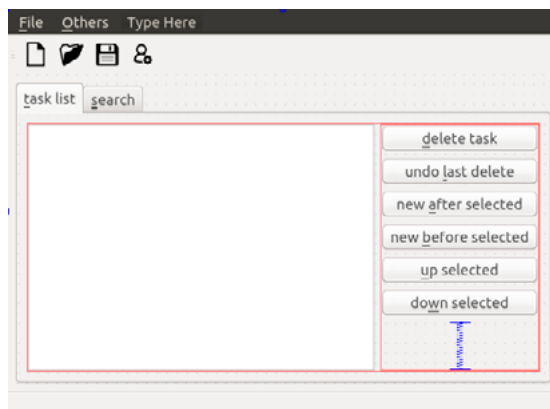
Hozzuk létre az `Others` menü `QAction`-jei számára a `triggered()` signal-hoz kötődő slot-ot. Ahol van ikon, rendeljük a `QAction`-hoz, majd húzzuk a `QAction`-t az eszköztárra.

A második fülön helyezzünk el egy másik `QListView`-t, a neve legyen `searchView`. Szükség lesz még egy `QLineEdit`-re (`searchEdit` néven), meg egy `QLabel`-re. Ez utóbbi szövege legyen `list tasks in category:` (a megadott kategória feladatainak listája:), és adjuk meg buddy-jául a `searchEdit`-et.

Úri kedvünknek megfelelően definiáljunk gyorsbillentyűket az objektumokhoz. A `QTabWidget` első, `task list` feliratú füle legyen előtérben, és a tab-sorrendben legyen a `tableView` az első számú objektum.

A főablaknak állítsuk be a címét, és adjuk meg az alkalmazás ikonját.

A `mainwindow.cpp` fejlécei között helyezzük el az `#include "aboutwindow.h"` sort, majd az `on_actionAbout_triggered()` slot törzsét alakítsuk a következőképp:



20. ábra: A főablak gombja

```
AboutWindow aboutWindow;
aboutWindow.exec();
```

Ezzel a mozdulattal a legeslegfontosabb része már működik is a programunknak. Ha nagyon elvesztünk a tennivalóban, a könyv webhelyéről le tudjuk tölteni a mostani állapotnak megfelelő archívumot. Aki elég szemfüles, annak talán feltűnt, hogy nem alakítottunk ki fület a grafikonunknak – nos, nem is fogunk.

10.5. A modell, meg az osztály, ami kezeli

Készítsük el a `ModelManager` osztályt, mint a `QObject` osztály leszármazottját. Ezt az osztályt (illetve a belőle példányosított `modelManager` objektumot) fogjuk megbízni majd mindennel, ami a modellünk manipulációjával kapcsolatos. Egyelőre azonban még nincs modellünk, készítsünk hát.

A Qt-ban minden modellek őse a `QAbstractItemModel` osztály. Ez, mint a neve is mutatja, eléggé absztrakt, magyarul elvont ősosztály, telis-tele virtuális tagfüggvényekkel, azaz csak úgy tudnánk használni, ha először is elkészítenénk egy alosztályát, amiben aztán végre megvalósítjuk a tonnányi tagfüggvényt. No, ezt egyelőre nem tesszük, mert akkor semmi időnk nem maradna a hörcsögre, és még a végén az elhanyagoltság oda vezetne, hogy sérülne szegényke lelki fejlődése.

Úgy döntünk tehát a magunk bölcsességében, hogy a nagy ős egyik leszármazottját, a kissé bonyolult sikeredett, ám azonnal használható `QStandardItem` osztályt fogjuk használni nemes céljaink – nevezetesen, hogy ne nekünk kelljen buherálni az adatstruktúrának megfelelő nézetfrissítésekkel – elérésére.

A `QStandardItemModel` osztály nevének első két szavából az derül ki, hogy ez egy olyan modell, amelynek elemei `QStandardItem` osztályba tartozó objektumok. A dokumentáció szerint az ilyen elemek rendszerint szöveget, ikont, vagy jelölőnégyzetet tartalmaznak, és tényleg: mi is épp szöveget készülünk beléjük tölteni: egy-egy teendőt, illetve kategóriát. Ráadásul, így a dokumentáció, az elemek ki-be kapcsolhatók, szerkeszthetők, kijelölhetők, és egyáltalán, mindenféle jóságra alkalmasak. Remek.

Szétnézve az osztály konstruktorai között (hangyányit túl vannak terhelve, de ezt itt, a Qt-ban lassan megszokja az ember) találunk olyat, amelyikkel sor és oszlopszám megadásával példányosíthatjuk az objektumunkat. Ha most arra gondolnánk, hogy jesszompipi, hát honnan tudjam én előre, hogy hány sor kell, akkor az `appendRow()` és a `removeRow()` tagfüggvényre vetve szemünk fényét, szívünk dobogása kicsi csitulhat.

Már most eláruljuk, hogy nem csak a `modelManager` objektum fog túrni a modellben, hanem majd megint írunk `fileOperator` osztályt is. Így a modellt érdemes lesz a heap-en elhelyezni, és a mutatóját eltenni magunknak.

Ennyi elméleti megfontolás után végre írhatnánk pár sor kódot, nem?

A `modelmanager.h` fájl fejlécei között helyezzük el a `QStandardItemModel` és a `QStandardItem` osztály használatát lehetővé tevő fejléceket, majd készítsünk egy `todoListModel` nevű, `QStandardItemModel` osztályú objektumra mutató publikus mutatót. A `ModelManager` osztály konstruktorában pedig példányosítsunk magunknak objektumot:

```
todoListModel = new QStandardItemModel(0, 2, this);
```

Ezek szerint a ropogós új modellünk 0 sor, 2 oszlop, és nem felejtkezünk el az szülő beállításáról sem. Hogy miért nulla sora van? Nos, azért, mert

- menet közben lesz, hogy a modellünket majd kiürítenénk (olyankor, amikor új listát kezdünk, meg a kész listák betöltését megelőzően),
- a modell ürítésére a `clear()` tagfüggvény használatos,
- a `clear()` tagfüggvény a sorok számát nullára csökkenti (sőt, még a fejléceket is törli, erről mindjárt lesz szó),
- azaz minden törlés után új sort kell majd hozzáadnunk a modellhez, amibe a drága felhasználó az első feladatot beírhatja,
- ráadásul a fejléceket is vissza kell állítani,
- azaz jó sok ismétlődő feladat lesz, amit önálló tagfüggvényre bízunk,
- és a tagfüggvénynek ne kelljen már gondolkodni, hogy most akkor van-e már sor a modellben, vagy nincs.

Hát, ezért.

Készítsük is el gyorsan a tagfüggvényt, ami mindezt elvégzi:

```
void ModelManager::emptyModel()
{
    todoListModel->clear();
    QList<QStandardItem* > newRow;
    todoListModel->appendRow(newRow);
    todoListModel->setHorizontalHeaderLabels(QStringList() << tr("job")
    << tr("category"));
}
```

A `clear()` tagfüggvényt megbeszéltük. Az `appendRow()` tagfüggvény mutatólistát vár, így először el kell készítenünk ezt a listát, `QList<QStandardItem* >` osztályú objektumként (ne feledjük a `QList` osztály használatát lehetővé tevő fejléc felvételét). Ígértük, hogy a fejlécről is lesz szó, hát tessék: a tagfüggvény utolsó sora végzi a vízszintes fejléc beállítását. Függőleges fejléceket nem adunk meg, így a `tableView` objektum majd szül egyet.

A kész tagfüggvény hívását helyezzük el az osztály konstruktorának törzsében, a modellt példányosító sor alá.

Egyelőre elvégeztünk mindent, amit ebben az osztályban szerettünk volna, úgyhogy visszaslattyoghatunk a főablakba, ahol példányosíthatjuk az osztályt, természetesen a heap-en. Az előző fejezet vége felé már láttuk, hogy a főablak konstruktora erősen elhízott, esetleg érdemes volna talán a feladatait külön tagfüggvényben megfogalmazni – aki számára ez javítja a program áttekinthetőségét, annak mindenképp.

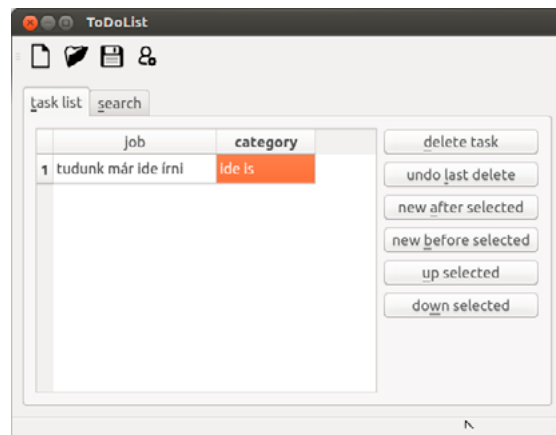
```
void MainWindow::preparationAtStartup()
{
    modelManager = new ModelManager(this);
    ui->tableView->setModel(modelManager->todoListModel);
}
```

A tagfüggvény utolsó sorában megadjuk a `tableView` objektum modelljét. A főablak konstruktorából hívjuk ezt a függvényt.

Az alkalmazás futtatását követően ki is próbálhatjuk a nézetet: máris tudunk bele írni.

Ha úgy érezzük, hogy a függőleges fejléc – a számozás – nem olyan fontos, akkor a felületszerkesztőben kapcsoljuk ki: keressük meg a **Property Editor**-ban a **verticalHeaderVisible** tulajdonságot, és vegyük ki mellőle a pipát.

Aki elveszett volna, természetesen megint megtalálja a források jelenlegi állapotát a könyv webhelyén.



21. ábra: Modellt kapott a nézetünk

10.6. Kényelem mindenekfelett

A szép új ToDoList most épp nem is annyira szép, mert a modell nem tölti ki a nézetet, sem hosszában, sem széltében. A hosszábanról igazán nem tehetünk: kevés benne a feladat, azaz a sor. A széltével viszont kéne kezdeni valamit, mert oszlopból több sosem lesz már, ezt a kettőt kell beosztani. Szerencsére ez nem probléma, egy egész utasítással megoldható – persze a Qt-től nem is vártunk mást, ugye? Akkor ez most a modell, vagy a nézet beállítása lesz? Ne feledjük, a modell azt sem tudja, hogy neki van nézete, arról meg aztán épp fogalma nincs, hogy a nézet miként jeleníti meg az adatokat. E logika alapján a **QTableView** osztály tagfüggvényei között kell szétnézni. Meg is találjuk, amit kerestünk, és a nagy keresgetés végén a főablak **preparationAtStartup()** tagfüggvényének törzsében utolsó sorként elhelyezzük a

```
ui->tableView->horizontalHeader()->setSectionResizeMode
(QHeaderView::Stretch);
```

sort. A két kettőspont meg a körülöttük lévő két szó ismét egy **enum**, azaz nevesített konstans, aminek értéke valójában 1. Az ilyen **enum**-ok lehetséges értékeinek listáját legokosabb, ha a kettőspont előtti osztály dokumentációjában keressük meg.

Ha már ilyen szépen kinyújtottuk a fejlécet, tegyük ki a hörcsögöt kicsit a billentyűzetre, merítsünk inspirációt abból, ahogy a lábacskaival tapodva szebbnél szebb karakterláncokat jelenít meg a képernyőn.

Mikor kellően inspirálódtunk, megvilágosodunk: nincs „új feladat” gomb. Persze van két másik, amivel végül is – majd, ha már leprogramoztuk – lehet új feladatot felvenni, de a gombok nyomkodása tönkretenné az inspirációt, és ugye nem mindenkinek van hörcsöge. Mekkora jó volna, ha a feladat felvitelével automatikusan kapnánk egy új sort.

Új sort ugye a modellben kell elhelyezni, de hát honnan jövünk rá, hogy *mikor* kell odatenni a sort? Ez valami signal lesz, úgyhogy lapozzuk fel a **QStandardItemModel** signal-jait. Hogyaszongya: **itemChanged(QStandardItem*)**, és ennyi. Ha megnézzük az ősosztálytól örökölt signal-okat, akkor sem javul a helyzet. Szóval itt vagyunk, egy árva signal-lal, amiből ugye a megváltozott elem mutatója derül ki. Nekünk meg mondjuk nem volna épp hátrányunkra, ha tudnánk, hogy hányadik sorban változott meg az elem, mert ugye, ha nem az utolsóban, akkor arról van szó, hogy a felhasználó valamelyik régebbi feladaton – urambocsá’ kategórián – változtatott, és akkor nem kell üres sort biggyesztenünk a modell végére.

Kicsit felderül az arcunk, ha megnézzük, hogy mit lehet kezdeni a visszakapott mutatóval. A `QStandardItem` osztály dokumentációjában van `row()` és `column()` tagfüggvény, azaz a modell elemei tudják magukról, hogy melyik sorban és oszlopban vannak. Hát akkor szüret! Oldalogjunk át a `ModelManager` osztályunkba és valósítsuk meg az `itemChanged(QStandardItem*)` signal-t elkapó privát slot-ot:

```
void ModelManager::modelItemChanged(QStandardItem *changedItem)
{
    if((changedItem->row() == toDoListModel->rowCount()-1) &&
        (changedItem->column() == 0) && (changedItem->text() != "")){
        QList<QStandardItem* > newRow;
        toDoListModel->appendRow(newRow);
    }
}
```

Azaz amennyiben az utolsó sorban változott az elem, és az első oszlopban változott az elem (mert ha a drága felhasználó a kategória megadásával kezdené, akkor nem adunk neki új sort), és a változás végeredménye nem üres teendő (mert esetleg meggondolta magát, és mégsem akar új sort), akkor kap új sort a modell. A `ModelManager` osztály konstruktorában adjuk ki a megfelelő connect utasítást:

```
connect(toDoListModel, SIGNAL(itemChanged(QStandardItem*)),
        this, SLOT(modelItemChanged(QStandardItem*)));
```

és már használhatjuk is alkalmazásunk legújabb képességét. Gondoljuk csak át, mi is történik: a modell szól a `ModelManager` osztályú objektumnak, hogy megváltozott az egyik elem, mire az megkéri a modellt, hogy ugyan vegye már föl utolsó sorként a frissen előállított, egyébiránt üres elemlistát. A grafikus rész, azaz a nézet, az egészsből annyit vesz észre, hogy valamilyen furcsa okból hirtelen eggyel több sor lett, amit szolgálai meg is jelenít.

A források jelenlegi állapota megint csak letölthető a könyv webhelyéről, de mostantól ezt nem ismételtetjük többé, nem baj?

10.7. Bevetjük a `QItemSelectionModell` osztályt

Itt az ideje megvalósítani a jobb oldali gombsorhoz, pontosabban a gombok `clicked()` signal-jaihoz rendelt slot-okat. Átgondolva a feladatot, elfilóztatunk azon, hogy a *kijelölt* sor törléséről van szó, ugyebár. De honnan a vizesvödörből derül ki, hogy melyik sor van kijelölve? Tudja a modell? Dehogy, az még azt sem tudja hogy látszanak az adatai. Tudja a nézet? Hát, akár tudhatná is, de nem ilyen egyszerű a helyzet.

A Qt szerint mindezt úgy érdemes megoldani, hogy külön `QItemSelectionModell` (azaz kijelölésmodell) osztályú objektumban tartjuk nyilván a kijelöléseket. Ez az objektum annyira szorosan kötődik a modellhez, hogy már a konstruktor paramétereként meg kell adnunk a forrásmodellt. Ebben az objektumban beállítjuk, hogy mi minden legyen kijelölve. A megjelenítés onnantól zajlik, hogy a nézetnek is szóltunk a kijelölésmodell létezéséről. Azaz elvileg megint van rá módunk, hogy egy modellnek egyszerre többféle kijelölése is éljen, és a nézetben ezek közül azt használjuk, amelyiket épp alkalmasnak tartjuk. Mi most meglegszünk egy kijelölésmodellel, amit a `ModelManager` osztály belsejében keltünk életre. Magához a modellhez hasonlóan a kijelölésmodell is a heap-re költözik, és nyilvánossá tesszük a mutatóját, mert a főablakon belül létező nézetnek kell tudni belőle olvasnia – sőt, írnia is bele, mert ugye

ott fogjuk majd kijelölgetni a szerkeszteni, törölni, mozgatni vágyott sorunkat.

A `modelmanager.h` fejlécei között helyezzük el a `<QItemSelectionModel>`-t, hozzuk létre a mutatót `*todoSelectionModel` néven, majd a `ModelManager` osztály konstruktorában a modellt példányosító sort követően adjuk ki a

```
todoSelectionModel = new QItemSelectionModel(todoListModel, this);
```

utasítást.

A főablak `preparationAtStartup()` tagfüggvényének törzsében negyedik sorként mutassuk be a kijelölésmodellt a nézetnek:

```
ui->tableView->setSelectionModel(modelManager->todoSelectionModel);
```

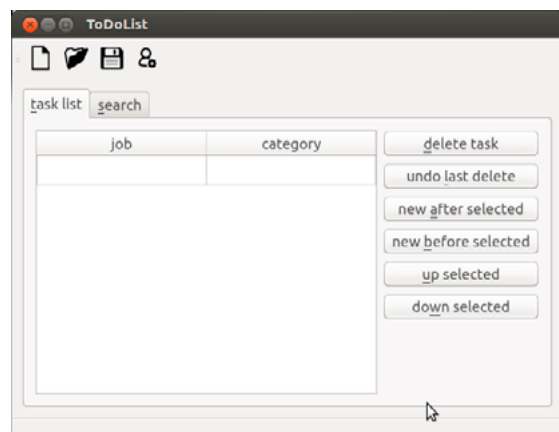
Ha most lefordítjuk és futtatjuk a programunkat, akkor az ég egy világon semmilyen változást nem látunk benne, de mi már tudjuk, hogy a `modelManager` objektum belsejében is minden pillanatban ki tudjuk deríteni, hogy a nézetnek épp melyik cellája van kijelölve. S ha már itt tartunk, a `Property Editor`-ban a `tableView` tulajdonságai között állítsuk be a kijelölés módját cellánkéntire, azaz a `selectionMode` tulajdonság értékéül adjunk meg `singleSelection`-t. Kacérkodhatunk a gondolattal, hogy a `selectionBehavior` (kijelölés viselkedése) tulajdonság értékét `selectRows`-ra (soronkénti kijelölés) állítjuk, de az a helyzet, hogy a kijelölést mi arra is használni akarjuk, hogy a felhasználónak megmutassuk, melyik cellába fog írni, ha gépelni kezd. Úgyhogy inkább hagyjuk `selectItems`-en (elemenkénti kijelölés). Utolsóként még annyit tegyünk meg, hogy a `tabKeyNavigation` mellől is kivesszük a pipát, s így lehetővé tesszük, hogy a tab-sorrend érvényesüljön, és a tabulátor nyomkodásával is el lehessen jutni a nyomógombokig.

Apropó, nyomógombok! Hát épp azért kezdtünk bele a kijelölésmodell használatába, hogy a nyomógombok végre működni tudjanak. Még mielőtt tényleg hozzáfognánk, a `ModelManager` osztály konstruktorában a modell inicializálását végző `emptyModel();` sor alatt helyezzük el, hogy

```
todoSelectionModel->setCurrentIndex(todoListModel->index(0,0),QItemSelectionModel::SelectCurrent);
```

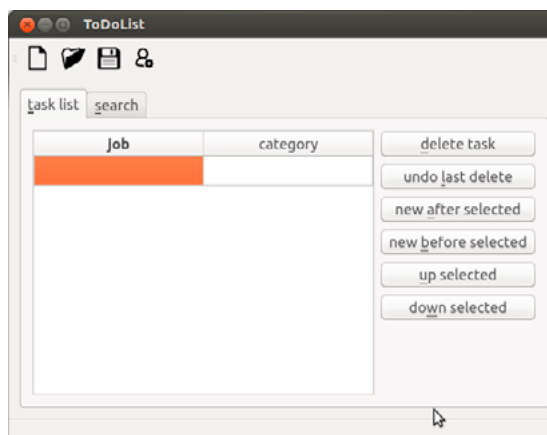
Azaz jelenlegi (nem kijelölt, csak kiválasztott) indexként állítsuk be a 0. sor 0. oszlopának elemét, de úgy, hogy a kiválasztással párhuzamosan azért kapjon kijelölést is. A két következő ábra összehasonlításával képet kapunk a dolog értelméről.

Így azért jobban hívogat, hogy „Írj már ide valamit, lécci-lécci-lécci!”



22. ábra: A program indulása az első cella kijelölése nélkül

Akkor végre a gombok. Az általuk emittált signal-okat közvetlenül a **modelManager** objektum slot-jaihoz kötjük majd, így ezeket a slot-okat publikusnak kell deklarálnunk. Lássuk a feladat törlését végző slot-ot:



23. ábra: A program indulása az első cella kijelölésével

```
void ModelManager::deleteSelectedTask()
{
    int row = toDoSelectionModel->currentIndex().row();
    if(row >= 0){
        toDoListModel->removeRow(row);
        if(row >= toDoListModel->rowCount()-1)
            toDoSelectionModel->setCurrentIndex(toDoListModel->index
(row-1,0),QItemSelectionModel::SelectCurrent);
        else
            toDoSelectionModel->setCurrentIndex(toDoListModel->index
(row,0),QItemSelectionModel::SelectCurrent);
    }
}
```

A kijelölésmodellből megtudjuk, hogy melyik sor van kijelölve. Azért kell vizsgálnunk, hogy a sor száma legalább nulla-e, mert kis ügyeskedéssel pillanatok alatt -1 értékűvé tudjuk tenni majd, pusztán a gomb nyomogatásával. Az esetszétválasztás első lehetősége akkor fut majd le, ha az utolsó sort töröltük, minden más esetben a második, az **else** utáni lehetőség valósul meg. Mindkét esetben kijelöljük az aktuális sort, hogy a felhasználónak nagyon nyilvánvaló legyen, hova gépel majd.

Eszünkbe jut a **connect** utasítás is, amit a főablak **preparationAtStartup()** tagfüggvényének törzsében helyezünk el, célszerűen olyan részen, ahol már létezik mindkét objektum: a nyomógomb is, meg a **modelManager** is.

```
connect(ui->buttonDelete, SIGNAL(clicked()),
        modelManager, SLOT(deleteSelectedTask()));
```

A törlés visszavonását lehetővé tevő gombot szemérmesen átugorjuk, és az új sorokat beszűrő gombok slot-jainak megvalósításával folytatjuk.

```
void ModelManager::newTaskAfterSelected()
{
    int row = todoSelectionModel->currentIndex().row();
    QList<QStandardItem* > newRow;
    todoListModel->insertRow(row+1, newRow);
    todoSelectionModel->clearSelection();
    todoSelectionModel->setCurrentIndex(todoListModel->index(row+1,0),
    QTableWidgetItem::SelectCurrent);
}
```

```
void ModelManager::newTaskBeforeSelected()
{
    int row = todoSelectionModel->currentIndex().row();
    QList<QStandardItem* > newRow;
    todoListModel->insertRow(row, newRow);
    todoSelectionModel->clearSelection();
    todoSelectionModel->setCurrentIndex(todoListModel->index(row,0),
    QTableWidgetItem::SelectCurrent);
}
```

A két slot – nem is gondolná az ember – igen hasonlóra sikerült. Mindössze két helyen különböznek, mégpedig a tekintetben, hogy az aktuális sor helyére szúrnak-e be sort, a többit egyfelé előre léptetve (ezt a felhasználó majd úgy látja, hogy az aktuális sor elé került az új sor), vagy az aktuális sort követően. Az `insertRow()` tagfüggvény nagyon hasonlóan működik a már ismert `appendRow()`-hoz, attól a nüansznyi különbségtől eltekintve, hogy a beszúrásakor meg kell mondanunk azt is, hogy hova kérjük a sort, míg a hozzáfűzés esetében erre nyilvánvaló okból nincs szükség. A kijelölés elvégzése előtt töröljük az érvényben lévő, hiszen nem akarjuk hogy a régi és az új kijelölés is látszódjék. A törlés gomb esetében erre azért nem volt szükség, mert az aktuális sor, és vele a kijelölés is törlődött.

Elhelyezzük a megfelelő `connect` utasításokat is:

```
connect(ui->buttonNewAfter, SIGNAL(clicked()),
        modelManager, SLOT(newTaskAfterSelected()));
connect(ui->buttonNewBefore, SIGNAL(clicked()),
        modelManager, SLOT(newTaskBeforeSelected()));
```

Akkor a következő két gomb, megint egyben tárgyalva:

```
void ModelManager::moveUpSelectedTask()
{
    int row = todoSelectionModel->currentIndex().row();
    if(row > 0){
        todoListModel->insertRow(row-1, todoListModel->takeRow(row));
        todoSelectionModel->setCurrentIndex(todoListModel->index(row-1,0),
        QTableWidgetItem::SelectCurrent);
    }
}
```

```
void ModelManager::moveDownSelectedTask()
{
    int row = todoSelectionModel->currentIndex().row();
    if(row < todoListModel->rowCount()-1){
        todoListModel->insertRow(row+1,todoListModel->takeRow(row));
        todoSelectionModel->setCurrentIndex(todoListModel->index(row+1,0),
        QTableWidgetItem::SelectCurrent);
    }
}
```

A kezdet most is azonos: megtudjuk, hogy hanyadik sorban kóricál a felhasználó. A feltételvizsgálat már eltér. Amikor felfelé vinnénk valamit, akkor az a kérdés, hogy van-e még korábbi sor, és korábbi sor akkor van, ha a mostani sor sorszáma nullánál nagyobb. Amikor lefelé vinnénk a sort, akkor a kérdés úgy szól, hogy nem vagyunk-e máris mindennek a végén. Ha a vizsgált feltétel teljesül, akkor belekezdünk a sor mozgatásába. A `takeRow()` a `takeItem()` tagfüggvény pár fejezettel korábbi előfordulása miatt elvileg ismerős – akkor zavarónak találtuk a nevet, de mostanra megbarátkoztunk vele. Ő lesz az a kartárs, aki kiveszi a sort a helyéről, visszaadja a mutatóját, de egyébként a sort magát nem kukázza le. A múltkor, a `takeItem()` esetében ezen bosszankodtunk, mert épp nagy volt a a kukázhatnékunk. Most azonban okosan használjuk a dolgot, és a kivett sort még azon melegében vissza is tesszük az új helyére. Utolsó mozdulatként a kijelölésmodellt is megfelelően módosítjuk.

Kötelező fordulatként lássuk a megfelelő `connect` utasításokat (ugye nem feledtük, hogy a főablak fájljába kerülnek?):

```
connect(ui->buttonUpSelected, SIGNAL(clicked()),
        modelManager, SLOT(moveUpSelectedTask()));
connect(ui->buttonDownSelected, SIGNAL(clicked()),
        modelManager, SLOT(moveDownSelectedTask()));
```

Annak öröme, hogy ilyen szépen elkészültünk a gombok öthatodával, visszatérhetünk a kihagyott „undo last delete”, azaz az utolsó törlés visszavonása feliratúhoz.

Előrebocsátanánk, illetőleg fölhívjuk a figyelmet arra, hogy nem utolsó műveletet írtunk, hanem utolsó törlést. A visszavonás teljes értékű megvalósításához nagyon sokféle dolgot kellene feljegyezni, például azt, hogy hova került be egy új sor, hányszor mozgattunk felfelé-lefelé egy kész sort, ráadásul még azt is, hogy *miről* írtuk át a cellát olyanra, amilyen most. Erre a feladatra meg egy strandkönyvben nem szoktak vállalkozni.

Mi „csak” annyit teszünk, hogy készítünk egy vermet (angolul stack), ha tetszik, LIFO-t. A LIFO annyit tesz, hogy Last In, First Out, azaz ami utoljára ment be, az jön ki elsőnek. Vermet az ember nem ásóval-lapáttal készít a Qt-ban (kis híján olyan területre bukkantunk, ahol még a hörcsög is lepipált volna bennünket), hanem példányosít magának a `QStack<T>` osztályból. A T betűből persze megint tudjuk, hogy ez egy sablonosztály, azaz meg kell mondanunk, hogy miket, milyen típusú objektumokat tárolunk a belőle készített példányban. Mi a magunk bölcsességében úgy döntünk, hogy ha már úgyis adatpárokról, két `QString`-ről van szó, akkor miért ne használnánk a `QPair` osztályt? (Tárolhatnánk a kivett sorok mutatóit, de akkor megint `QSharedPointer`-ekkel kéne szöszölni, ahhoz meg már késő van.)

Úgyhogy a `modelmanager.h` állomány elejére vegyük fel a `<QStack>` és a `<QPair>` fejléct, majd privát objektumként deklaráljuk a

```
QStack<QPair> undo;
```

nevűt.

Az undo objektum töltögetését a törlésért felelős slot-ban végezzük majd, mégpedig úgy, hogy a sor törlését megelőzően (tehát a `todoListModel->removeRow(row);` sor előtt) kiadjuk az

```
undo.push(QPair<QString, QString>(todoListModel->index(row,0).data().toString(), todoListModel->index(row,1).data().toString()));
```

utasítást.

Szép hosszú, igaz?

A visszavonás megvalósítása nem jelent mást, mint a verem legfelső elemének kivételét, és a benne lévő két karakterlánc alapján képzett objektum elhelyezését a modellben. Minthogy nem tároljuk, honnan vettük ki – nem is beszélve arról, hogy azóta össze-vissza mozgathattuk, szerkeszthettük az elemeinket, és újakat is szúrhattunk be –, a modell legvégére biggyesztjük a régi-új sort.

```
void ModelManager::undoLastDelete()
{
    if(!undo.isEmpty()){
        QList<QStandardItem* > newRow;
        todoListModel->appendRow(newRow);
        todoListModel->setData(todoListModel->index(todoListModel->rowCount()-1, 0), undo.top().first);
        todoListModel->setData(todoListModel->index(todoListModel->rowCount()-1, 1), undo.pop().second);
    }
}
```

A `setData()` tagfüggvény első paramétere azt mutatja meg, hogy *hol* kell beállítania a második paraméterben megadott adatot. A második paraméterben látható `QStack::top()` tagfüggvény a verem legfelső elemére mutató hivatkozást ad vissza, és *nem távolítja el* az elemet a veremből. A legfelső elem történetesen egy `QPair<QString, QString>` osztályú objektum (igen, a többi is), aminek hívható a `first()` tagfüggvénye. A második `setData()`-hívás során már más tagfüggvényét használjuk a `QStack` osztálynak: egyszer `top()`, másszor `pop()`. A `pop()` igen hasonló a `top()`-hoz, de ki is veszi az elemet a veremből. Ha a slot működését megértettük, helyezzük el a megfelelő connect utasítást is:

```
connect(ui->buttonUndo, SIGNAL(clicked()),
        modelManager, SLOT(undoLastDelete()));
```

Futtassuk a programunkat, és ... hogy az a sistersgős-mennydörgős! A modell `itemChanged` signal-jához kötött slot most is teszi a dolgát, és képes beszúrogatni nekünk új sorokat. Nem lehetne megmondani neki, hogy kicsit hagyja abba? De. Létezik a `connect` párja, a `disconnect`, épp csak még sosem használtuk, de hát ugye mindig van egy első alkalom. Esetünkben az első alkalom nem az utolsó, mert mondjuk a fájl betöltésénél hasonló problémával kerülünk majd szembe. Így aztán írunk is egy jó kis tagfüggvényt, ami a paramétere értékétől függően ki-be kapcsolgatja a problémát okozó signal és slot kapcsolatot:

```
void ModelManager::connectItemChangedSignalToSlot(bool needed)
{
    if(needed)
        connect(toDoListModel, SIGNAL(itemChanged(QStandardItem*)),
                this, SLOT(modelItemChanged(QStandardItem*)));
    else
        disconnect(toDoListModel, SIGNAL(itemChanged(QStandardItem*)),
                this, SLOT(modelItemChanged(QStandardItem*)));
}
```

A fenti setData()-sorok előtt kikapcsoljuk:

```
connectItemChangedSignalToSlot(false);
```

utánuk meg bekapcsoljuk:

```
connectItemChangedSignalToSlot(true);
```

A törlés visszavonása remekül működik. Volna még értelme annak is, hogy a szép új ki-be kapcsolgató tagfüggvénynek már a ModelManager osztály konstruktorában is hasznát vegyünk, nevezetesen az ottani connect utasítást cserélhetjük le e tagfüggvény hívására. A hörcsög három centi répa fejében elárulja, hogy melyik paramétert kell híváskor megadnunk.

11. ADATMENTÉS ÉS –VISSZATÖLTÉS MODELL HASZNÁLATAKOR

Az adatmentés és az adatok visszatöltése első ránézésre nem különbözik alapvetően attól a változattól, amit a kilencedik fejezet végéig használtunk. Miként jártunk el addig? Volt egy listánk, ami `task` osztályú objektumokat – pontosabban `task` osztályú objektumokra mutató mutatókat mutatóit – tárolt. Ezt a listát nyálazta végig a `fileOperator` objektum mentéskor, illetve ezt töltötte fel új objektumokkal a mentett fájl megnyitásakor. Lényegében akkor is volt egy adatszerkezetünk, ami tárolta az adatainkat és most is van egy: a modell. A két eset között az a nagy különbség, hogy ezúttal nem a mi dolgunk az adatszerkezet előállítása, csak annyi a teendőnk, hogy a már kész modellben engednünk kell túrni a `FileOperator` osztályt. Szerencsére a modell, illetve a rá mutató mutató, bár a `modelManager` objektumunkban „él”, publikus, azaz a `fileOperator` objektum tud majd turkálni benne.

Így aztán kevesebb bajunk van az adataink integritásának megőrzésével, cserébe meg tudunk valósítani pár kényelmi funkciót.

11.1. A Beállítások ablak

Az utolsó mentett fájlt persze ezúttal is szeretnénk majd visszatölteni, meg jó volna megvalósítani az automatikus mentést ismét, azaz mindenképp szükségünk lesz a beállítások tárolására. Ha még emlékszünk, annak érdekében, hogy a beállításokat kezelő, `QSettings` osztályú objektumokat a programunk különféle területein könnyen tudjuk példányosítani, érdemes volt pár adatot megadni az alkalmazásunkról. Kezdjük hát azzal a munkát, hogy ezt megtesszük. Első dolgunk a `<QCoreApplication>` fejléc felvétele a főablakban. Ezt követheti a következő három sor elhelyezése a `preparationAtStartup()` tagfüggvény törzsének elején:

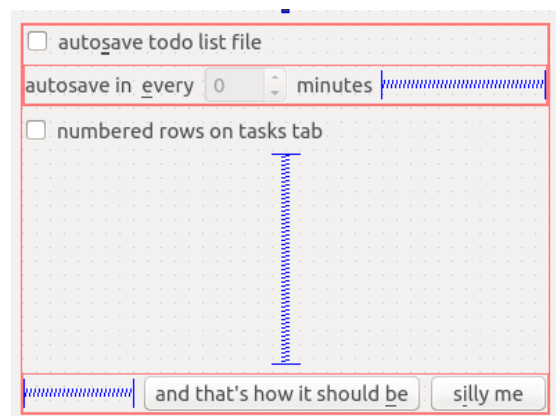
```
QCoreApplication::setOrganizationName("Sufni & Co.");
QCoreApplication::setOrganizationDomain("example.com");
QCoreApplication::setApplicationName("ToDoList");
```

Ezt követően alakítsuk ki a `SettingsWindow` osztályt. Az osztály őssztálya a `QDialog`. A felhasználói felületet alakítsuk ilyenre:

Az automatikus mentésen felül még egy beállítást tárolunk: kívánság esetén mégiscsak megjeleníthetővé tesszük a főablak `tableView` elemének sorszámozását.

Az elemek nevei (alighanem kitaláljuk, hogy melyik melyik):

- `autoSaveCheckBox`
- `autoSaveSpinBox`
- `numberedRowsCheckBox`
- `buttonOk`
- `buttonCancel`



24. ábra: A Settings ablak

Az ablakot modális ablakként jelenítjük majd meg, és ha a felhasználó elfogadta a módosításokat – magyarul az OK-gombot nyomta meg –, akkor tárolunk is mindent a `QSettings` osztály használatával. Az ablak modális mivolta miatt a `fileOperator` objektum ráér a már mentett változtatásokat használni, csak szólnunk kell neki, hogy valami változott. A főablaknak meg a sorszámozás miatt kell majd szólnunk.

A `settingswindow.h` fájlban deklarálunk egy `settings` nevű objektumot – a használatát lehetővé tevő `<QSettings>` fejléc felvételét se felejtjük el megejteni. Az osztály konstruktorában a következő három sorral az eltárolt beállításoknak megfelelően állítjuk be a beállítások ablak elemeit:

```
ui->autoSaveCheckBox->setChecked(settings.value("Files/
AutoSaveEnabled", "false").toBool());
ui->autoSaveSpinBox->setValue(settings.value("Files/
AutoSaveMinutes", 0).toInt());
ui->numberedRowsCheckBox->setChecked(settings.value("UI/
NumberedRows", "false").toBool());
```

Ugye emlékszünk még, hogy a `QSettings::value()` tagfüggvény a második, elhagyható paraméterével tette lehetővé, hogy olyankor is megadjunk vele értéket, ha még nincs tárolva a kiolvasni vágyott beállítás?

Szükség lesz még egy privát slot-ra, amelyik az `autoSaveSpinBox`-ot kapcsolgatja ki-be:

```
void SettingsWindow::on_autoSaveCheckBox_stateChanged(int arg1)
{
    if(arg1 == Qt::Unchecked)
        ui->autoSaveSpinBox->setEnabled(false);
    else if(arg1 == Qt::Checked)
        ui->autoSaveSpinBox->setEnabled(true);
}
```

Végül egy utolsóra, amelyik az OK gomb lenyomása esetén lendül akcióba:

```
void SettingsWindow::on_buttonOk_clicked()
{
    settings->setValue("Files/AutoSaveEnabled", ui->autoSaveCheckBox->
isChecked());
    settings->setValue("Files/AutoSaveMinutes", ui->autoSaveSpinBox->
value());
    settings->setValue("UI/NumberedRows", ui->numberedRowsCheckBox->
isChecked());
    this->done(QDialog::Accepted);
}
```

Figyeljük meg, hogy a `done()` tagfüggvény paramétereként adjuk tudtára az ablakot megnyitó objektumnak – esetünkben a főablaknak –, hogy végül is a felhasználó elfogadta a beállításokat.

Az utolsó két slot a neve alapján automatikusan kötve lesz a megfelelő signal-hoz, vagyis ezt a gondot elfelejtethetjük. A `SettingsWindow` osztály elkészült.

Itt van hát az ideje, hogy az előző fejezetben kialakított `MainWindow::on_actionSettings_triggered()` slot törzsét is megírjuk, de előtte még a főablak valamelyik fájljában vegyük használatba a `settingswindow.h` fejlécet. A slot törzse nem bonyolult, cserébe viszont rém egyszerű:

```
SettingsWindow settingsWindow;
if(settingsWindow.exec() == QDialog::Accepted)
    restoreApplicationState();
```

Azaz példányosítunk magunknak egy beállítások-ablakot, méghozzá a stack-en, azaz mihelyt kilépünk belőle és végigfut a slot, meg is szűnik létezni az ablak. Ha a felhasználó a beállítások elfogadásával zárta be az ablakot, akkor érvényre juttatjuk őket a kissé (szóval: igencsak) nagyképűen elnevezett `restoreApplicationState()`, azaz alkalmazásÁllapotánakVisszaállítása privát tagfüggvénnyel, amit egyébként a `preparationAtStartup()` tagfüggvény törzsének végén is hívni kell. A név nyilván nem volna ennyire nagyképű, ha beállítások százait tárolnánk, de ugye eddig pont három beállításunk van, és ezek közül ebben az osztályban – azaz a főablakban – csak egy érdekes. No, sebj. Maga a tagfüggvény a következő formát ölti:

```
void MainWindow::restoreApplicationState()
{
    QSettings settings;
    if(!settings.value("UI/NumberedRows", "false").toBool())
        ui->tableView->verticalHeader()->hide();
    else
        ui->tableView->verticalHeader()->show();
    fileOperator->adjustAutoSave();
}
```

A `fileOperator` objektum még sehol nincs persze, sőt, az osztálya sem, azaz, ha ki szeretnénk próbálni, hogy eddig minden rendben-e, akkor a tagfüggvény törzsének utolsó sorát alakítsuk megjegyzéssé.

11.2. A *FileOperator* osztály reinkarnációja

Végre hozzákezdünk ahhoz, ami a fejezet valódi témája. Adjunk a projekthez új osztály, `FileOperator` néven, és ezúttal ne a `QObject`, hanem a `QWidget` legyen az őszosztály, mert, ahogy arra bizonyára emlékszünk, a fájl-választó ablakok ősének mindenképp `QWidget` osztályúnak kell lennie. Az osztályt a főablakból fogjuk példányosítani, és a konstruktorának egyik paramétereként adjuk át a modellünk mutatóját. Azaz az osztály konstruktora ilyesmi lesz:

```
FileOperator::FileOperator(QStandardItemModel *todoListModel, QWidget
*parent) :
    QWidget(parent)
{
    model = todoListModel;
    timer = new QTimer(this);
}
```

A `timer` objektum természetesen az automatikus mentés miatt kell majd. Apropos, automatikus mentés: írjuk már meg gyorsan azt a fránya `FileOperator::adjustAutoSave()` függvényt, aminek a hívását az imént szégyenszemre megjegyzéssé kellett alakítanunk. A tagfüggvénynek nem lesz más dolga, mint a beállítások beolvasása, és a `timer` objektum megfelelő felparaméterezése. Az automatikus mentés kivitelezése egy másik tagfüggvény feladata lesz majd, ami ráadásul nem csak tagfüggvény, de slot is lesz, így hozzá tudjuk kötni a `timer` objektum ki(él?)süléséhez. Íme a tagfüggvényünk:

```
void FileOperator::adjustAutoSave()
{
    if(settings.value("Files/AutoSaveEnabled", "false").toBool())
        timer->start(settings.value("Files/AutoSaveMinutes").toInt()
*60000);
    else
        timer->stop();
}
```

Végezzük el a FileOperator osztály példányosítását a főablakban. Olyan helyen kell megejtenünk, ahol már van modell, de még a restoreApplicationState() tagfüggvény hívása előtt, mert különben nem indul be az automatikus mentés.

Ha eddig minden klappol, nekikezdünk a mentést végző tagfüggvény elkészítésének. Ezúttal a fájlnevet paraméterként kapja meg. A fájl megnyitása és használatba vétele a már megismert módon történik, és a modell mentése közben figyelünk arra, hogy üres teendőjű feladatot – azaz az olyan sorokat, amelyek első oszlopa üres – nem mentünk. Sikeres mentés esetén a beállításokban tároljuk a fájlnevet. A hibaüzeneteket ezúttal külön tagfüggvényben helyezzük el, hiszen mentéskor és megnyitáskor nagyon hasonlóan magyaráztuk el a felhasználónak, hogy baj történt. A hibajelzést végző tagfüggvény paraméterként kapja meg, hogy mikor is (mentéskor vagy megnyitáskor) történt baleset. A mentést végző privát tagfüggvény végül ilyenre sikeredett:

```
bool FileOperator::performFileSaveOperation(QString fileName)
{
    QFile file(fileName);
    bool success = false;
    if (file.open(QFile::WriteOnly | QFile::Truncate | QFile::Text)) {
        QTextStream out(&file);
        for(int i = 0; i < model->rowCount(); i++){
            QString job = model->index(i,0).data().toString().
simplified();
            if(!job.isEmpty())
                out << job << "|" << model->index(i,1).data().
toString() << endl;
        }
        if(!file.error())
            success = true;
    }
    if(success){
        settings.setValue("Files/LastFileName", fileName);
    }
    else{
        fileOperationErrorMessage(tr("Save"));
    }
    return success; //can be "false" too
}
```

Gyorsan megírjuk a fileOperationErrorMessage() privát tagfüggvényt is:

```

oid FileOperator::fileOperationErrorMessage(QString operation)
{
    QMessageBox mb;
    mb.setIcon(QMessageBox::Critical);
    mb.setText(operation + " " + tr("not succeeded."));
    mb.setInformativeText(tr("Try and do something wise."));
    mb.exec();
}

```

Az előző alkalommal úgy gondolkodtunk, hogy a felhasználó alighanem az előző fájl mellé mentené az újat is, ha meg nem volt előző fájl, akkor valahova a saját cókómkja közé. Ráadásul azt is feltételeztük, hogy hasonló gondolatmenet fut át az agyán a mentett fájl megnyitásakor is: hátha az előző mellől nyitna meg másikat, mert mondjuk az imént mellékattintott. Így aztán mindkét alkalommal ilyen megfontolásból kiindulva állítottuk elő azt az elérési útvonalat, amelyet felhasználva megnyitottuk a fájlválasztó párbeszédablakot. Ha viszont kétszer is kellett, ugye van értelme külön tagfüggvénybe pakolni a fejtegetést? A nagyrabecsült Olvasó gesztusát bólintásként értelmezzük, és már itt is az alapértelmezett elérési útvonalat javasló privát tagfüggvény:

```

QString FileOperator::suggestDefaultPath()
{
    QString path = QDir::homePath();
    QString lastFileName = settings.value("Files/LastFileName", "").
toString();
    if(lastFileName != ""){
        QFileInfo info(lastFileName);
        path = info.absolutePath();
    }
    return path;
}

```

Minden készen áll ahhoz, hogy megírjuk azt a tagfüggvényt is, amelyik előbb beszerzi a szükséges fájlnevet, és aztán ennek ismeretében végezteti el a mentést. A múltkor jól bevált módszer szerint, ha a felhasználó nem kegyeskedett megadni a fájl kiterjesztését, akkor majd mi megadjuk. Íme:

```

bool FileOperator::fileSave()
{
    QString newFileName = QFileDialog::getSaveFileName(
        this,
        tr("Save as..."),
        suggestDefaultPath(),
        tr("ToDoList files (*.tdolst);;text files (*.txt);;all
files (*)")
    );
    if(!newFileName.isEmpty()){
        QFileInfo info(newFileName);
        QString ext = info.suffix();
        if(ext != "tdolst" && ext != "txt")
            newFileName += ".tdolst";
        return performFileSaveOperation(newFileName);
    }
    return false;
}

```

Törjük kicsit a fejünket. Ha a felhasználó Ctrl+S billentyűkombinációt nyom, és van fájlnev (ki tudjuk olvasni a beállításokból), akkor csuklás nélkül hívjuk a `performFileSaveOperation()` tagfüggvényt. Ha nincs fájlne, akkor a `fileSave()` tagfüggvényt hívjuk, megtudjuk a fájlnevet és ennek ismeretében megint csak a `performFileSaveOperation()`-ra lőcsöljük a melót. Ha a felhasználó a Save as... menüpontot választaná, akkor tekintet nélkül arra, hogy van-e fájlnevünk vagy nincs, mindenképp újat kell kérnünk, hiszen a Save as... menüpontnak pont ez a lényege. Végiggondoltunk minden lehetőséget? Talán. Akkor megírhatjuk a két `QAction` osztályú objektum (mármint a Save és a Save as...) slot-ját. Hol volna érdemes? A főablakban egyszerűbb, a `FileOperator` osztályban elegánsabb. Mi elegánsak leszünk, és publikus slot-ként a `FileOperator`-ban helyezzük el a két következő szösszenetet. A függvények elnevezése nagyon hasonló ahhoz, amit automatikus létrehozásukkal kapnánk, de szándékosan nem teljesen ugyanolyan, ugyanis a moc (a Meta Object Compiler) azonnal hőbörögne, hogy látja ő, hogy itt a slot, de hol a signal? (Mi persze tudjuk, hogy egy másik osztályban, de a moc-nak nincs ám annyi esze, hogy megkérdezze tőlünk.) És akkor most mihez kössön, meg micsodát? Mi meg nem bírjuk, ha valaki hőbörög, így inkább picit változtatunk a neveken.

```
void FileOperator::onActionSaveTriggered()
{
    QString fileName = settings.value("Files/LastFileName", "").
toString();
    if(!fileName.isEmpty())
        performFileSaveOperation(fileName);
    else
        fileSave();
}
```

```
void FileOperator::onActionSaveAsTriggered()
{
    fileSave();
}
```

Már csak az a dolgunk velük, hogy a főablakban kiadjuk a megfelelő connect utasításokat:

```
connect(ui->actionSave, SIGNAL(triggered()),
        fileOperator, SLOT(onActionSaveTriggered()));
connect(ui->actionSave_as, SIGNAL(triggered()),
        fileOperator, SLOT(onActionSaveAsTriggered()));
```

Természetesen nincs sok teteje a dolognak, ha még a `fileOperator` példányosítása elé írjuk a sorokat, de a példányosítást követően azonnal jöhetnek. Elgondolkodhatunk azon is, hogy közvetlenül a `fileSave()` tagfüggvényt tesszük slot-tá, és azt kötjük a Save as... menühöz. Nem sok ellenérv van ellene, talán az, hogy egyesek számára a jelenlegi forma áttekinthetőbb. Ez viszont egy igen-igen fontos érv, mindenkit meggyőzhet, kivéve esetleg azokat, akik szerint a másik forma áttekinthetőbb. A hörcsög még nem nyilvánította ki véleményét az ügyben, úgyhogy a meccs még nem lefutott.

Megnyitás következik, pedig ez a könyv nem is a sakkról szól. A megnyitást végző tagfüggvények természetesen nagyon hasonlítanak majd a mentést végzőkhöz, és ugyanúgy párban járnak. Itt következik mind a kettő:

```
bool FileOperator::fileOpen()
{
    QString newFileName = QFileDialog::getOpenFileName(
        this,
        tr("Open file..."),
        suggestDefaultPath(),
        tr("ToDoList files (*.tdolst);;text files (*.txt);;all
files (*)")
    );
    if(!newFileName.isEmpty())
        return performFileOpenOperation(newFileName);
    return false;
}
```

```
bool FileOperator::performFileOpenOperation(QString fileName)
{
    model->clear();
    model->setHorizontalHeaderLabels(QStringList() << tr("job") <<
tr("category"));
    QFile file(fileName);
    bool success = false;
    if (file.open(QIODevice::ReadOnly | QIODevice::Text)){
        QTextStream in(&file);
        while (!in.atEnd()){
            QStringList sl = in.readLine().split("|");
            QStandardItem *item = new QStandardItem(sl.at(0));
            model->setItem(model->rowCount(), 0, item);
            item = new QStandardItem(sl.at(1));
            model->setItem(model->rowCount()-1, 1, item);
        }
        if(!file.error())
            success = true;
    }
    if(success){
        settings.setValue("Files/LastFileName", fileName);
        QList<QStandardItem* > newRow;
        model->appendRow(newRow);
    }
    else{
        fileOperationErrorMessage(tr("Open"));
    }
    return success; //can be "false" too
}
```

A `performFileOpenOperation()` tagfüggvényt a `fileOpen()` tagfüggvény hívja, és még ki? Mert, ha senki, akkor minek külön tagfüggvény? Nos, a másik hely, ahonnan hívni fogják, a program indításakor automatikus fájlbetöltés lesz. Úgyhogy van létjogosultsága a dolognak. Viszont a `fileOpen()` tagfüggvény hívása körül van még egy icuri kis probléma.

Alighanem eszünkbe ötlük még, hogy a kényelem mián `ModelManager` osztály berkeiben megírtuk azt a slot-ot, amelyik az utolsó sor első oszlopában lévő elem megváltozásakor azonnal létrehoz egy üres sort. Hamarosan bajba is keveredtünk vele, olyankor is elő-előkerült egy-egy üres sor, amikor az legkevésbé sem szolgáltá óhajainkat. Meg is írtuk a nagy ki-be kapcsolható `connectItemChangedSignalToSlot()` tagfüggvényt, amelyről már akkor előrevetítettük,

hogy a fájlbetöltéskor még jó hasznát vesszük majd. Nos, ez az idő – mármint a fájlbetöltés, meg a probléma előjövetele – itt van. Szeretnénk hívni a tagfüggvényt, de ugye a `fileOperator` objektum nem látja. Kénytelenek leszünk szégyenszemre ezt az egy slot-ot a főablakban megvalósítani? Azt már nem!

Úgy módosítjuk a `FileOperator` osztály konstruktorát, hogy ne csak a modell, hanem az egész `modelManager` objektum mutatóját vegye át. Íme:

```
FileOperator::FileOperator(ModelManager *modelManager, QWidget
*parent) :
    QWidget(parent)
{
    this->modelManager = modelManager;
    model = modelManager->todoListModel;
    timer = new QTimer(this);
}
```

A törzs első sora azt teszi lehetővé, hogy a `FileOperator` osztályon belül is `modelManager` néven hivatkozassunk az objektumunkra. Persze át kell alakítanunk a `fileOperator` objektumot példányosító sort is a főablak forrásában, sőt, a `ModelManager::connectItemChangedSignalToSlot()` tagfüggvényt is publikussá kell tennünk, merthogy eddig privát volt, a szentem. Végre megírható a megnyitást végző slot:

```
void FileOperator::onActionOpenTriggered()
{
    modelManager->connectItemChangedSignalToSlot(false);
    fileOpen();
    modelManager->connectItemChangedSignalToSlot(true);
}
```

És, igen, most sem hagyjuk el a `connect` utasítást a főablakból:

```
connect(ui->actionOpen, SIGNAL(triggered()),
        fileOperator, SLOT(onActionOpenTriggered()));
```

Akkor ez is megvan. Bueno.

11.3. Utolsó lehetőség a mentésre – a `QMessageBox` osztály további lehetőségei és a `Q_PROPERTY` makró

Most, amikor az új fájl – ha tetszik, új lista – kezdéséhez szükséges tagfüggvény megvalósításán kezdünk gógyizni, eszünkbe jut, hogy milyen mérges arcot vág majd a felhasználó, ha csak véletlenül kattint az „Új” ikonra, és törlődik az egész addigi munkája. Persze mi tudjuk, hogy ő lesz a hibás, miért nem állított be automatikus mentést?! Ami még nem is működik. Mármint a beállítás működik, a mentés nem.

Szóval, csak szólni kéne neki, hogy mentse már, ami menthető. Persze vaklármázni sem akarunk, mert mi a helyzet, ha épp az előző kattintásával mentette? Jó volna nyilvántartani valahogy, hogy volt-e változás az utolsó mentés óta. Hol is kellene ezt nyilvántartani? Legyen mondjuk a `modelManager` objektumnak egy tulajdonsága, legalább megtanuljuk, hogy miként készíti az ember amolyan igazi Qt-os tulajdonságot²⁴. A tulajdonság neve az lesz, hogy `todoListModelChanged`, de csak azért ilyen röviden, mert érezzük, hogy a

24 Olvasnivaló: <http://qt-project.org/doc/qt-5.1/qtcore/properties.html>

todoListModelChangedSinceLastSave túlzás lenne.

A `modelmanager.h` fájlban a `Q_OBJECT` makró alá helyezzük el az alábbi sort:

```
Q_PROPERTY(bool todoListModelChanged READ todoListModelChanged
WRITE setToDoListModelChanged NOTIFY todoListModelChangedChanged)
```

Ebből aztán a Qt világában tudni lehet, hogy ennek az osztálynak van egy fent említett nevű, `bool` típusú tulajdonsága, aminek a kiolvasófüggvénye (angolul: getter) is azonos névvel bír, a beállítófüggvényének (angolul: setter) neve a `WRITE` szó után van megadva, és amennyiben a tulajdonság megváltozna, az osztályból példányosított objektumok a `NOTIFY` szó után megadott signal-t emittálják (minden kedves Olvasótól elnézést kérünk a némileg zavaróra sikeredett signal név miatt).

Persze a makró nem a teljes igazságot hirdeti, amennyiben a privát tagváltozó neve valójában `m_todoListModelChanged` – már említettük, hogy ezt így szokás, és hogy az `m` valójában a „member”, azaz „tag” szó rövidítése.

A tulajdonság persze nincs azzal teljesen kész, hogy mindezt elmondtuk róla. Deklarálnunk kell még a

```
void todoListModelChangedChanged(bool);
```

signal-t, illetve deklarálásra és megvalósításra szorul még a kiolvasó- és beállítófüggvény is. Íme:

```
bool ModelManager::todoListModelChanged() const
{
    return m_todoListModelChanged;
}
```

```
void ModelManager::setToDoListModelChanged(bool todoListModelChanged)
{
    m_todoListModelChanged = todoListModelChanged;
    emit todoListModelChangedChanged(todoListModelChanged);
}
```

Most viszont tényleg elkészültünk vele, már csak használatba kell vennünk. Először is, a `FileOperator` osztályban jelezzük, ha az ügyködésünknek köszönhetően a modell azonos állapotúvá vált a fájlal. Ez mind a sikeres mentést, mint a sikeres megnyitást követően elmondható, azaz ahol a `performFileSaveOperation()`, illetve `performFileOpenOperation()` tagfüggvényben a függvény vége felé a beállítások közé mentjük a fájlnevet, beállíthatjuk ennek a tulajdonságnak az értékét is, mégpedig a

```
modelManager->setToDoListModelChanged(false);
```

utasítással. Magában a `modelManager` objektumban viszont jóformán minden ténykedésünkkel elrontjuk a fájl és a modell szép konzisztens állapotát, azaz

```
setToDoListModelChanged(true);
```

utasítást kell elhelyeznünk a `modelItemChanged()`, a `deleteSelectedTask()`, az `undoLastDelete()`, a `newTaskAfterSelected()`, a `newTaskBeforeSelected()`, a `moveUpSelectedTask()` és a `moveDownSelectedTask()` tagfüggvényben. Azonban az `emptyModel()` tagfüggvényben épp ellenkező a helyzet: itt `false` értéket kell megadnunk.

A tulajdonság megváltozásakor mindig emittálunk signal-t is, amihez a főablakban írunk egy jó kis publikus slot-ot:

```
void MainWindow::showIfModelChangedSinceLastSave(bool changed)
{
    QSettings settings;
    QString fileName = settings.value("Files/LastFileName",
tr("unsaved")).toString();
    if(changed)
        this->setWindowTitle(QCoreApplication::applicationName() + " *
"
+ fileName);
    else
        this->setWindowTitle(QCoreApplication::applicationName() + " "
+ fileName);
}
```

A slot igyekszik megtudni a jelenlegi fájl nevét, ha nem megy neki, akkor jelzi, hogy a fájl még nincs mentve. Ha a legutolsó fájlművelet óta van változás, akkor csillagot is ír a címsorba, máskor viszont nem. Ne feledjük összekötni a signal-lal:

```
connect(modelManager, SIGNAL(todoListModelChangedChanged(bool)),
        this, SLOT(showIfModelChangedSinceLastSave(bool)));
```

Most, hogy végre minden pillanatban tisztában vagyunk vele, hogy kell-e menteni, megírhatjuk az új listát kezdő tagfüggvényt, vagy legalább komolyabban átgondolhatjuk, mit is kellene tudnia. Először is, megnézi, hogy kellene-e menteni, és ha igen, akkor megkérdezi a felhasználót, hogy ő is úgy gondolja-e. Ha a felhasználó azt mondja, hogy mentene, akkor kap rá lehetőséget. Ha azt mondja, hogy nem mentene, akkor kap új, üres feladatlistát. Ha pedig azt állítja, hogy meggondolta magát, akkor nem erőltetjük mi sem a dolgot, és meghagyjuk neki a régi listáját.

Így adódott, hogy mégsem írjuk meg még a slot-unkat, előbb megírjuk azt a tagfüggvényt, amelyik majd figyelmezteti a felhasználót:

```
bool FileOperator::lastPossibilityToSaveFile() {
    if(modelManager->toDoListModelChanged()){
        QMessageBox mb;
        mb.setIcon(QMessageBox::Warning);
        mb.setText(tr("Your ToDoList has changed since the last
save."));
        mb.setInformativeText(tr("Grab this last possibility to save
it!"));
        mb.setStandardButtons(QMessageBox::Save | QMessageBox::Discard |
QMessageBox::Cancel);
        mb.setDefaultButton(QMessageBox::Save);
        int ret = mb.exec();
        switch (ret) {
            case QMessageBox::Save:
                onActionSaveTriggered();
                return true;
                break;
            case QMessageBox::Discard:
                return true;
                break;
            case QMessageBox::Cancel:
                return false;
                break;
        }
    }
    return true;
}
```

Az ablakunk egy üzenetablak, és ezúttal a szabvány gombok közül válogatunk. Megadjuk azt is, hogy alapértelmezés szerint a felhasználó igenis menteni akar, így legalább nem lesz gond, ha vaktában nyomkodja az **Enter**-t. A tagfüggvény visszatérési értéke dönti el, hogy kap-e üres listát a felhasználó. Ha nem is kell menteni, akkor **true** értékkel térünk vissza, azaz kap. Ha kell menteni, akkor hívjuk ugyanazt a slot-ot, amit a **Ctrl+S** billentyűkombináció lenyomása esetén is hívnánk, és ahol vagy azonnal megtörténik a mentés, vagy előbb fájlnevet kérünk, és azután igyekszünk menteni. Mindenesetre a végén csak megkapja a felhasználó az új listáját. Ha a felhasználót nem érdekli az adatainak elvesztése, vagy épp ezt akarta, akkor is kap új listát. Egyedül akkor nem kap újat, ha a „Mégse” gombra kattint.

Végre a slot-unk következik:

```
void FileOperator::onActionNewTriggered()
{
    if(lastPossibilityToSaveFile()){
        modelManager->emptyModel();
    }
}
```

Eltávolítjuk a bejegyzések közül a fájlnevre vonatkozót, és kérünk egy új modellt. Persze csak akkor, ha a felhasználó meg nem gondolta magát. Az **emptyModel()** tagfüggvény még privát, tegyük gyorsan publikussá, és ha már úgy is arra járunk, helyezzünk el benne egy

```
settings.remove("Files/LastFileName");
```

sort, különben néha-néha felülírunk olyasmit is, amit nem akarunk.

Most pedig futtassuk a programunkat, és örvendezzünk, igazán remek lett. Eltekintve attól, hogy megnyitáskor még simán rányitja a mentetlen listára a mentettet. Érdekes volna ezt is rendbekapnunk, főleg, hogy semmi többet nem kell tenni, mint az `onActionOpenTriggered()` slot törzsét körbevenni egy `if(lastPossibilityToSaveFile())` feltétellel.

11.4. Automatikus mentés és megnyitás

Az automatikus mentés már elég régen félkész állapotban leledzik. A `FileOperator` osztály létrehozását követően azonnal megírtuk az `adjustAutoSave()` tagfüggvényt, amit a főablak `restoreApplicationState()` tagfüggvénye hív, részint a program indulásakor, részint a beállításlablak beállításainak elfogadásakor. Az `adjustAutoSave()` függvény elindítja a visszaszámlálást, csak akkor még nem kötöttünk semmit a visszaszámlálás végét jelző `QTimer::timeout()` signal-hoz, mert nem volt mit. Most azonban hip-hopp kész vagyunk a megfelelő slot-tal:

```
void FileOperator::autoFileSave()
{
    performFileSaveOperation(settings.value("Files/LastFileName", "").
        toString());
}
```

Mi a helyzet akkor, ha be van állítva automatikus mentés, de nincs fájlnev? Nos, ez egy új problémát vet fel, de ezt most gyorsan elvarrjuk: a felhasználó majd kap hibaüzenetet, hogy nem ment a mentés, és majd gondolkodik, hogy ugyan milyen mentésről van szó. Az osztály konstruktorában még elhelyezzük a

```
connect(timer, SIGNAL(timeout()),
        this, SLOT(autoFileSave()));
```

utasítást, és az automatikus mentés életre kel. Lássuk a program indulásakor automatikus betöltést. Ezzel résen kell lennünk, mert a `ModelManager::emptyModel()` függvény, amit már az osztály konstruktora is hív, törli az utolsó mentett fájl nevét. Úgyhogy sutyiban eltesszük, még az osztály példányosítását megelőzően, egy

```
QSettings settings;
QString lastFileName = settings.value("Files/LastFileName", "").
    toString();
```

utasításpárral eltesszük magunknak, a `fileOperator` objektum létrejötte után meg a

```
if(!lastFileName.isEmpty())
    fileOperator->autoFileOpen(lastFileName);
```

utasítással kezdeményezzük a fájl megnyitását. A hívott tagfüggvény törzse teljes három sor:

```
void FileOperator::autoFileOpen(QString fileName)
{
    modelManager->connectItemChangedSignalToSlot(false);
    performFileOpenOperation(fileName);
    modelManager->connectItemChangedSignalToSlot(true);
}
```

És ezzel kész is vagyunk. Alighanem megérdemlünk egy lakomát. A hörcsög ülhet a főhelyre – de csak azért, mert ott jól látszik, és nem kell félnünk, hogy beszalad valamilyen sarokba, ahonnan soha többé nem tudjuk előkotorni.

12. A MODELL, A DELEGÁLT, MEG A PROXYMODELL

Ebben a fejezetben először azzal fogunk foglalatostkodni, hogy saját delegáltat írunk a modellünkhöz. Persze még nem feltétlen tudjuk, mi az a delegált, de majd idővel fény derül rá. Ha megvagyunk vele, ismét beizzítjuk az automatikus kiegészítést, ami ugyan nem lesz tökéletes, viszont elképesztően kevés munkával jár, és a kísérletező kedvű Olvasó előtt nyitva marad a lehetőség a tökéletesítésére. Végül elkészítjük a második fülön a keresést is.

12.1. Delegált megvalósítása a *QStyledItemDelegate* osztály használatával

Azt ígértük, hogy idővel kiderül, mi az a delegált. Most jött el az idő. A delegált szó eredeti értelmében ugye olyasvalakit jelöl, akire valamilyen feladatot rá lehet szólni, és az illető a nevünkben eljárva intézkedhet. A *QStyledItemDelegate* osztályú objektumokra a modell elemeinek megjelenítését, szerkesztését szózza rá az ember, már olyankor is, amikor nem is sejtji. A mi modellünk delegáltjai jelen pillanatban *QLineEdit* osztályú objektumok. Amikor szerkeszteni kezdünk egy elemet, az elem értékét átadjuk egy *QLineEdit* osztályú objektumnak, amely a szerkesztés befejeztével visszahelyezi a megváltozott elemet a modellbe – ezt az eseményt teszi közhírré a *QStandardItemModel::itemChanged()* signal.

Ha az alapbeállításokkal nem vagyunk maximálisan megelégedve, akkor alosztályt képzünk a *QStyledItemDelegate* osztályból, és megadjuk, hogy a nézetünk ezt használja inkább. Ebben az alfejezetben mi is pontosan ezt tesszük majd. Fogjunk is hozzá!

Első feladatunk az új osztály elkészítése. Adjunk tehát új osztályt a projektünkhöz, mégpedig *Delegate* néven. Az osztály ősosztályául adjuk meg a *QStyledItemDelegate* értéket, a *Type Information* sornál pedig állítsuk be, hogy *inherits QObject*. A friss osztályunkban az ősosztály négy virtuális publikus tagfüggvényét kell megvalósítanunk, úgymint:

```
• QWidget* createEditor(QWidget *parent, const QStyleOptionViewItem &option, const QModelIndex &index) const;
• void setEditorData(QWidget *editor, const QModelIndex &index) const;
• void setModelData(QWidget *editor, QAbstractItemModel *model, const QModelIndex &index) const;
• void updateEditorGeometry(QWidget *editor, const QStyleOptionViewItem &option, const QModelIndex &index) const;
```

Amikor elkezdjük a deklarációkat a *delegate.h* fájlban, a Qt Creator szerkesztője előbb-utóbb felismeri a nevet, és automatikusan fel fogja ajánlani nekünk a paraméterlistát. Okos, nem?

Amikor szerkeszteni kezdjük a modell egy elemét, a nézet a *createEditor()* tagfüggvény hívásával példányosít magának egy bármilyen *QWidget*-leszármazott objektumot. Az *updateEditorGeometry()* tagfüggvénynek az a dolga, hogy a jó helyen jelenítse meg a kész szerkesztőt. Ha megvan a szerkesztő, a nézet a *setEditorData()* hívásával adja át neki a szerkeszteni valót. A szerkesztés befejeztével a *setModelData()* függvény helyezi vissza a megváltozott elemet az eredeti helyére.

Valósítsuk meg ezeket a tagfüggvényeket. A *createEditor()* tagfüggvényben annyi a dolgunk, hogy létrehozunk egy *QLineEdit* osztályú objektumot, és visszaadjuk a mutatóját.

Íme a függvény törzse:

```
Q_UNUSED(option);
Q_UNUSED(index);
QLineEdit *editor = new QLineEdit(parent);
return editor;
```

A `Q_UNUSED` makrórt használtuk már – arra jó, hogy a fordító ne méltatlankodjon a paraméterlistában megtalálható, ámde általunk nem használt változók miatt.

A `setEditorData()` törzse az alábbi formát ölti:

```
QString value = index.model()->data(index, Qt::EditRole).toString();
QLineEdit *lineEdit = static_cast<QLineEdit*>(editor);
lineEdit->setText(value);
```

Először is kiharapjuk a megfelelő cella tartalmát, és elhelyezzük a `value` változóban. És, hogy mi az a `Qt::EditRole`? Igen, egy *enum*, de miért kell? Úgy áll a dolog, hogy a modell egy-egy eleme nem csak azt az adatot tartalmazza, amit első ránézésre gondol az ember. Hanem rengeteget még, például az előtér és a háttér színét, a betűtípust, meg hogy a szöveg merre van rendezve²⁵. Ezek a Qt nevezéktana szerint role-ok, azaz szerepek. Mi a fenti utasítással az adatok közül azt a role-t kérjük el, amelyik a szerkesztőben való munkára alkalmas.

Sajnos itt nem használhatjuk a jó kis `editor` nevet, mert összegabalyodik a paraméterlistában lévővel, így a `lineEdit`-re fanyalodunk. A paraméterlistában átadott szerkesztő mindig `QWidget` típusú, mi ennek egy alosztályát használjuk, de hogy melyiket, azt a törzs második sora előtt csak mi tudjuk, a Qt nem. A `static_cast` arra való, hogy elmondjuk a Qt-nak is, hogy ez a `QWidget` valójában és egész konkrétan egy `QLineEdit`. És miért olyan fontos számunkra, hogy a Qt is pontosan tisztában legyen a `lineEdit` objektum osztályával? Azért, mert a következő sorban olyan tagfüggvényt hívunk, amely nincs ám minden mezei `QWidget`-ben, de a `QLineEdit` osztályú objektumokban van.

Ott tartunk, hogy vadul szerkesztgethetjük az elemünket. Ha pedig befejeztük a szerkesztést, visszaadhatjuk a modellnek, hadd örüljön. Ezt a feladatot végzi a `setModelData` tagfüggvény. Íme a függvény törzse:

```
QLineEdit *lineEdit = static_cast<QLineEdit*>(editor);
model->setData(index, lineEdit->text());
```

A negyedik tagfüggvényt kicsit még implementálatlanul hagyjuk, pontosabban, minthogy már deklaráltuk, megírjuk, de üres törzsszel. Enélkül is elfut majd a delegáltunk, de kicsit furán: a szerkesztőablak rossz helyen jelenik majd meg. Legalább látjuk, hogy miért érdemes dolgoznunk. A sok nem használt változó miatt majd kapunk persze fél kiló warningot, de kibírjuk, ha meg nem, ott a `Q_UNUSED`.

Mielőtt az osztály példányosításába fognánk, még el kell helyoznünk az osztály valamelyik fájljában a `<QLineEdit>` fejlécet. Ha megvan, oldalogjunk át a főablakba. Deklaráljuk a `Delegate` osztályú objektumra mutató `delegate` mutatót, majd a `preparationAtStartup()` tagfüggvényben példányosítsunk neki objektumot, és mutassuk is be az új delegáltat a nézetnek:

```
delegate = new Delegate(this);
ui->tableView->setItemDelegate(delegate);
```

A program fordul, a delegált működik. Igaz, kicsit idétlen, hogy mindig a nézet bal felső

25 Itt a többi is: <http://qt-project.org/doc/qt-5.1/qtcore/qt.html#ItemDataRole-enum>

sarában van, de hát ugye, aki nem valósítja meg az `updateEditorGeometry()` tagfüggvényt, annak nem jár jobb. Akkor hát valósítsuk meg. Két teljes sor a törzse:

```
Q_UNUSED(index)
editor->setGeometry(option.rect);
```

Ha most futtatjuk a programunkat, klappol minden. Jó sokat dolgoztunk, hogy végre legyen egy saját delegáltunk, ami ugyanazt tudja, mint az alapértelmezett. Hát normálisak vagyunk?!

Persze volt mindennek értelme, hiszen így már tesztre szabható a delegáltunk. Kezdetnek oldjuk meg, hogy a két oszlopban más-más legyen a szerkesztő háttérszíne. Ezt úgy tudjuk elérni, hogy az objektum példányosítását követően megvizsgáljuk, hogy az adott indexű elem melyik oszlopban van, és más-más értéket adunk meg a 0, illetve az 1 sorszámmal. Szűrjük be ezt a pár sort a `return editor` utasítás elé:

```
if(index.column() == 0)
    editor->setStyleSheet("QLineEdit {background: yellow;}");
else
    editor->setStyleSheet("QLineEdit {background: lightgreen;}");
```

Távolítsuk el még a `Q_UNUSED(index);` sort, és gyönyörködjünk a színekavalkádban.

Nekifoghatunk az automatikus kiegészítés megvalósításának. A kiegészítést egy `QCompleter` osztályú objektum végzi. Használtunk már ilyet, talán emlékszünk is rá, hogy egy modellt is meg lehetett adni neki forrásul. Hát most épp ezt tesszük majd: megadjuk neki magát a `ToDoListModel`-t, annak is a második oszlopát. Persze a `Delegate` osztály nem ismeri a `ModelManager` osztályt, így a modellről sem tud. Egyelőre. Alakítsuk át a `Delegate` osztály konstruktorát úgy, hogy paraméterként kérje el a modellre mutató mutatót. Ha már nekifogunk, gyorsan példányosíthatunk a konstruktorban `QCompleter` osztályú objektumot is, meg be is állítjuk neki a modell második oszlopát:

```
Delegate::Delegate(QStandardItemModel *model, QObject *parent) :
    QStyledItemDelegate(parent)
{
    this->model = model;
    completer = new QCompleter(model, this);
    completer->setCompletionColumn(1);
    completer->setCaseSensitivity(Qt::CaseInsensitive);
    completer->setModelSorting(QCompleter::CaseInsensitivelySortedModel);
}
```

Beállítjuk még a kis-nagybetűkre való érzéketlenséget, illetve azt, hogy a modelltől kapott adatokat rendezze is a `completer` objektum. Ne felejtkezzünk el a `model` és a `completer` mutató deklarálásáról, illetve a `<QStandardItemModel>` és a `<QCompleter>` fejléc felvételéről. Változtatnunk kell a főablakban a `delegate` objektumot példányosító soron is, alakítsuk ilyenné:

```
delegate = new Delegate(modelManager->ToDoListModel, this);
```

Van már szép `completer` objektumunk, épp csak annyi vele a gond, hogy senki nem használja. A `Delegate` osztály `createEditor()` tagfüggvényében kell az `else` ágon elhelyezni még az

```
editor->setCompleter(completer);
```

utasítást. Megy a kiegészítés, bár az első örömmünket hamar elrontja, hogy ha már többször van megadva egy kategória, akkor többször is felajánlja a `completer`. Így jártunk.

12.2. A keresőfül és a QSortFilterProxyModel osztály

Vajh mi lehet az a proxymodell? Olyan modell, amelyik automatikusan változik, kapcsolatot tart fenn az eredeti modellel – ez lesz a proxymodell *forrásmodellje* –, de az eredeti modellek nem minden elemét tartja meg, vagy másképp mondja el őket. A proxymodellek a Qt-ban a `QAbstractProxyModel` osztály leszármazottai, de ez az osztály, mint a neve is mutatja, elvont, azaz rengeteg függvényét meg kell valósítani ahhoz, hogy használhassuk. A lehetőségek végtelenek: megoldható például, hogy egy számokat tartalmazó modell proxymodellje szavakként mondja el nekünk az eredeti modell számait, vagy csak a páratlan számokat tartalmazza.

A dolog azonban macerás, és egy strandkönyvben nem fogunk bele ilyesmibe. Szerencsére arra, amire nekünk kellene a proxymodell, nevezetesen, hogy csak bizonyos keresési feltételeknek megfelelő sorokat tartson meg az eredeti modell tartalmából, már van beépített leszármazott osztály, a `QSortFilterProxyModel`. Rendez és főleg: szűr, hát ez kell nekünk!

A proxymodellt is a `ModelManager` osztályban helyezzük el. Adjuk meg a fejlécek között a `<QSortFilterProxyModel>`-t, majd deklarálunk publikus mutatót `searchModel` néven. Az osztály konstruktorában példányosítsunk hozzá objektumot és ejtsük meg a szükséges beállításokat:

```
searchModel = new QSortFilterProxyModel(this);
searchModel->setSourceModel(todoListModel);
searchModel->setFilterKeyColumn(1);
searchModel->setFilterCaseSensitivity(Qt::CaseInsensitive);
```

Felhívánk a figyelmet a harmadik sorra: itt dől el, hogy a keresést a kategóriák oszlopában végzi majd a modell, amely immáron teljesen működőképes, épp csak semmi nem használja. Kullogjunk át a főablak `preparationAtStartup()` tagfüggvényébe, és csempésszük bele az alábbi sorokat:

```
ui->searchView->setModel(modelManager->searchModel);
ui->searchView->horizontalHeader()->setSectionResizeMode(QHeaderView::Stretch);
connect(ui->searchEdit, SIGNAL(textChanged(QString)),
        modelManager->searchModel, SLOT(setFilterFixedString(QString)));
```

Az első sorban a keresőfülön elhelyezett nézet modelljéül a proxymodellt adjuk meg. Vegyük észre, hogy épp úgy adjuk meg, mint ha igazi modell lenne. A nézetnek fogalma nincs, hogy ez a modell nem olyan modell. A második sor szerepe az, hogy a modell kitöltse a nézetet – ezt a „rendes” modellünk nézete esetében is így oldottuk meg. A harmadik sor pedig azért felel, hogy a proxymodell csak azokat a sorokat tartalmazhassa, amelyek második oszlopában megtalálható a `searchEdit` objektumban lévő karakterlánc.

Most, hogy meggy a keresés, nincs kibúvó: keressük meg a hörcsögös feladatainkat, és lássuk el kis kedvencünket.

Tartozunk egy vallomással: kész a `ToDoList` modelles változata. A jobb kezünkkel megveregethetjük a bal vállunkat – balkezesek természetesen használhatják ellenkező oldali felső végtagjaikat a gyakorlat elvégzésére.

És így, a tornamutatvány végére következzen még egy megszívlelendő jó tanács: a valós életben a programjainkba kemény ellenőrzést kell építenünk a bemeneti fájl integritását illetően. Ha például a feladatlistafájlból kivesszük a teendőt a kategóriától elválasztó cső („|”) karaktert, úgy elszáll a programunk, mint őszi viharban a műanyagzacskó.

13. HÁLÓZATI SZINKRONIZÁCIÓ: A ToDoList FELHŐCSKÉJE

Ebben a fejezetben a modell-nézet minta lehetőségeit kihasználó ToDoList-változatból kiindulva olyan alkalmazást készítünk, amelyik képes az adatfájlját webszerverre szinkronizálni. Az elképzelés a következő: ha a felhasználó szeretné, hogy a programja a feladatlista-fájlt a megadott webszerverre szinkronizálja, akkor a program mentéskor

- menti a fájlt a helyi adattárolóra
- feltölti a megadott webszerverre
- feltölt mellé egy időbélyeget is.

A fájl megnyitásakor:

- letölti az időbélyeget a webszerverről
- megnézi, hogy az időbélyeg szerint a webszerveren lévő fájl az újabb, vagy a helyi
- ha a webszerveren lévő az újabb, akkor letölti azt is
- megnyitja, amelyiket kell.

A programunk nem fog semmiféle hitelesítést tartalmazni a webszerver felé. Természetesen semmilyen titkosítás nem jön szóba. Komolyabb probléma, hogy a megoldásunk csak akkor működik jól, ha a felhasználó összes eszközén be van kapcsolva a hálózati szinkronizáció. Természetesen semmiféle konfliktuskezelést nem valósítunk meg. A legnagyobb baj azonban az, hogy a megoldásunk nem nyújt lehetőséget arra, hogy az egyik eszközünkön elkészült fájl létét a másik, újonnan használatba vett eszközünkön észrevegyük. Ha az egyik gépünkön létrehoztuk a `cucc.tdolist` állományt és felszinkronizáltuk a webszerverünkre, a másik gépünkön csak úgy tudjuk majd letölteni, ha ott már eleve volt egy régebbi `cucc.tdolist`. Hogy miért lesz ilyen korlátozott tudású a programunk? Az a helyzet, hogy a való életben igen komoly szerveroldali háttér – szerveroldali alkalmazás – kellene ahhoz, hogy megbízhatóan működjön egy ilyen szolgáltatás, és ez csak egy strandkönyv.

Ennyi hátulütő mellett a fejezet még mindig jó lesz arra, hogy a hálózatok felé tett kis kirándulással súroljuk a Qt hálózatkezelés-tengerének felszínét.

13.1. A webszerver előkészítése

Hálózati alkalmazás programozásába fogni saját, különbejáratú hálózat nélkül manapság dőreségnek számít, amit nem is fogunk elkövetni. Főleg, hogy a hálózat a mi alkalmazásunk esetében annyira egyszerű, hogy egy gépen elfér a kliens is és a szerver is.

Olyan webszerverre van szükségünk, amely képes PHP-kódot futtatni. Kivételt képeznek azok az Olvasók, akik túltéve magukat a szerveroldali rész meglehetősen összetettségén, vállalkoznak a hamarosan olvasható, PHP-nyelvű kód más szerveroldali környezetbe való átültetésére.

Akár Windows-on, akár Linux-on, akár OS X-en dolgozunk, alighanem a legkézenfekvőbb választás egy Apache 2.x webszerver telepítése. Hogy miért pont ezt ajánljuk, mikor annyi más szép webszerver van? Azért, mert ez a legelterjedtebb nyílt forrású, azaz bárki számára beszerezhető webszerver. És, lévén a legelterjedtebb, a legjobban dokumentált is – fordulhatunk a hivatalos forrásokhoz, de a YouTube is segítségünkre lehet.

Mikor az Apache 2.x fenn van, akkor telepítenünk kell mellé PHP-t is, és az Apache-nak szólni kell, hogy vegye használatba a PHP-t, és kezelje, futtassa le a **.php** fájlokat.

A PHP-telepítés ellenőrzésére egy **akarmi.php** nevű fájlt szokás elhelyezni ott, ahol a telepítésünk alapértelmezés szerint a webszerver fájljait tárolja (Windows-on ez alapértelmezés szerint az Apache telepítés alatti **htdocs** könyvtár, Debian alapú Linuxokon, azaz Ubuntu-n is a **/var/www** mappa). A fájl tartalma legyen a következő:

```
<?php
phpinfo();
?>
```

Ha ezek után a böngészőnkben a **http://localhost/akarmi.php** címre navigálunk (a **localhost** minden gépen maga a helyi gép, belső, úgynevezett visszacsatoló, angolosan: *loopback* hálózati interfésze, IP-címe: 127.0.0.1), akkor a következőhöz hasonló képnek illene fogadni bennünket:

Nézzünk még utána, hol találjuk meg rendszerünkön az Apache naplófájljait (angolul: log files). Ha a beállítófájlok és a dokumentáció alapján végképp nem találjuk, akkor érdemes fájlkereséssel próbálkoznunk. Keressünk **error.log**, illetve **access.log** nevű fájlokat. A naplófájlok helye Windows rendszeren alapértelmezés szerint az Apache telepítési mappája alatti **logs** könyvtár, Debian alapú Linux-okon a **/var/www/apache2** mappa. A naplófájlok egyszerű szövegfájlok, bármilyen egyszerű szövegszerkesztővel megnézhetők, de a parancssorból is olvashatók. Windows-on érdemes lehet a **more** programocskát használnunk a megfelelő mappában:

```
more < error.log
```

Linuxokon a naplófájlok folyamatosan frissülő tartalmát a **tail** paranccsal tudjuk úgy figyelemmel kísérni, hogy nem kell a fájlokat újra és újra megnyitni:

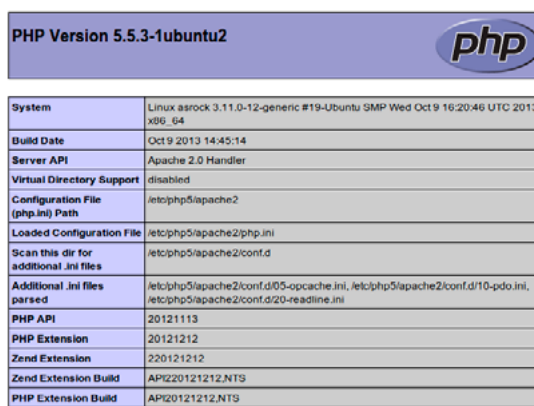
```
tail -f /var/log/apache2/*.log
```

A fejlesztés során a naplók vizsgálata sok fejfájástól kímélheti meg az embert (azért e fajspecifikus kijelentés, mert a hörcsögök *általában* nem szoktak naplófájlokat nézni). Innen fog kiderülni, ha rossz helyre tettük a szerveroldali PHP-kódot, vagy ha azért nem megy a fájlfeltöltés, mert a PHP-nak nincs engedélye az adott könyvtárba írni.

Pár bekezdéssel feljebb utaltunk rá, hogy hihetetlenül összetett és kifinomult PHP-kódot kell fejlesztenünk a fájlfeltöltés lehetőségét megteremtendő. Ilyen az eleje:

```
<?php
    $targetName = "/var/www/" . basename( $_FILES['uploaded']['name']);
    move_uploaded_file($_FILES['uploaded']['tmp_name'], $targetName);
?>
```

És ilyen a vége is, merthogy figyelmetlenségünkben máris végigolvastuk mind a négy sorát. A **/var/www** részt cseréljük ki a mi rendszerünk megfelelő mappájára, majd a fenti négy sort pótyogjuk be egy szövegfájlba – persze letölthetjük a könyv webhelyéről is –, és mentjük az



PHP Version 5.5.3-1ubuntu2	
System	Linux asrock 3.11.0-12-generic #19-Ubuntu SMP Wed Oct 9 16:20:46 UTC 2013 x86_64
Build Date	Oct 9 2013 14:45:14
Server API	Apache 2.0 Handler
Virtual Directory Support	disabled
Configuration File (php.ini) Path	/etc/php5/apache2
Loaded Configuration File	/etc/php5/apache2/php.ini
Scan this dir for additional .ini files	/etc/php5/apache2/conf.d
Additional .ini files parsed	/etc/php5/apache2/conf.d/05-opcache.ini, /etc/php5/apache2/conf.d/10-pdo.ini, /etc/php5/apache2/conf.d/20-readline.ini
PHP API	20121113
PHP Extension	20121212
Zend Extension	220121212
Zend Extension Build	API220121212.NTS
PHP Extension Build	API20121212.NTS

25. ábra: Ha ilyet látunk, akkor sikeresen beüzemeltük a PHP-t

előbbi `akarmi.php` fájl mellé, `upload.php` néven. Hogy mit csinál ez a négy sor? Elfogadja a feltöltött fájlokat, és a megadott helyre menti őket. Ennél hosszabb magyarázatra nem vállalkozunk, elvégre ez nem PHP-strandkönyv.

Nézzük meg, hogy a webszerverünket futtató felhasználónak van-e joga írni a megadott mappába, és elvileg a szervertől beállításokkal megvagyunk. Nem győzzük hangsúlyozni, hogy fejlesztés közben „Fél szem, fél fül a figyelésé”²⁶, mármint a naplófájlok figyeléséé.

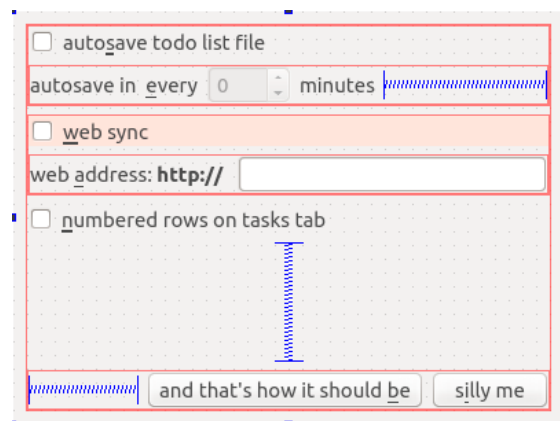
Akkor hát vissza a Qt-hoz.

13.2. Kiegészítjük a beállítás-ablakot

A grafikus felülettervezőben alakítsuk ilyenre a `settingswindow.ui` fájl tartalmát:

A két új elem neve: `webCheckBox` és `webLineEdit`. Igen, tudjuk, hogy van még egy `QLabel` osztályú felirat is, de azzal a programban nem beszélgetünk, ilyenformán a neve lényegtelennek minősítettük. Állítsassuk elő a `webCheckBox` slot-ját, majd egészítsük ki a kódot.

Az osztály konstruktorában beolvasunk még két beállítást:



26. ábra: Az új beállítás-ablak

```
ui->webCheckBox->setChecked(settings.value("Web/SyncEnabled",
"false").toBool());
ui->webLineEdit->setText(settings.value("Web/Address", "").toString());
```

Az OK gomb lenyomásakor a többivel együtt ezeket is tároljuk:

```
settings->setValue("Web/SyncEnabled", ui->webCheckBox
->isChecked());
settings->setValue("Web/Address", ui->webLineEdit->text());
```

A `webCheckBox` slot-ja nagyon hasonló lesz a másikhoz – mármint, ha jól írjuk meg. A slot törzse a következő:

```
if(arg1 == Qt::Unchecked)
    ui->webLineEdit.setEnabled(false);
else if(arg1 == Qt::Checked)
    ui->webLineEdit.setEnabled(true);
```

Kész is vagyunk. A próba-futtatás során tároltathatjuk a webcímet – a `webLineEdit`-ben a `localhost` szót kell megadnunk, ha a webszerver ugyanazon a gépen üzemel, ahol a programunkat fejlesztjük.

13.3. Feltöltés a WebSynchronizer osztállyal – a QNetworkAccessManager és egyéb hálózati örömeik

Hogy is szeretnénk mi ezt pontosan? Készítsünk egy `WebSynchronizer` osztályt, amelynek két publikus tagfüggvénye van – mármint a konstruktoron kívül, persze. Az egyik feladata

²⁶ Cipúr, Vili a veréb, Pannónia Filmstúdió, 1989.

a fájl lemezre való mentését követően feltölteni a webszerverre, a másiké meg az, hogy a fájl megnyitása előtt letölti az időbélyeget, és ha az időbélyeg szerint a webszerveren lévő feladatlista-fájl az újabb, akkor letölti azt is, és felülírja vele a helyi példányt. A `WebSynchronizer` osztályt majd a `FileOperator` objektumban példányosítjuk. Mentéskor-betöltéskor megnézzük a beállításfájlt, és ha kell szinkronizálni, akkor hívjuk a megfelelő tagfüggvényt. (Ismét megjegyezzük, hogy a feladat jelenlegi megvalósítása sok kivetnivalót hagy maga után. Ha például a letöltés során félig jön csak le a fájl, azzal is felülírjuk a helyi példányt, és akkor aztán használható fájl nélkül maradunk, ami az álmoskönyv szerint semmi jót nem jelent. Amikor mindezt majd élesben csináljuk, sokkal gondosabban kell eljárunk.)

Adjuk hát hozzá a `WebSynchronizer` osztályt a projekthez. Az őse lehet egy mezei `QObject`, semmi bonyolultabbra nincs szükségünk. Deklaráljuk benne az alábbi két publikus tagfüggvényt:

```
void syncFileBeforeOpen(QString fileName, QString webAddress);
void syncFileAfterSave(QString fileName, QString webAddress);
```

Helyezzük el a megvalósításukat – egyelőre üres törzsszel – a `websynchronizer.cpp` fájlban, s ezt követően a `FileOperator` osztály konstruktorában példányosítsunk magunknak egyet az új osztályból, mégpedig `webSync` néven, a heap-re:

```
webSync = new WebSynchronizer(this);
```

Ezt követően battyogjunk le a `performFileOpenOperation()` tagfüggvényhez, és valahol az elején, de még mindenképp azelőtt, hogy a helyi fájlal babrálni kezdenénk, helyezzük el az alábbi sorokat:

```
//webSync
if(settings.value("Web/SyncEnabled", "false").toBool())
    webSync->syncFileBeforeOpen(fileName, settings.value("Web/
Address").toString());
//webSync done
```

Szívünk titkos zugai elárulják nekünk, hogy a következő teendők az lesz, hogy hasonló sorokat helyezünk el a `performFileSaveOperation()` tagfüggvényben is, de ezúttal olyan helyre, ahol *már* nem fogunk babrálni a helyi fájlal. Ilyen hely például az `if(success)` teljesülése esetén lefutó ág. Az elhelyezendő sorok a következők:

```
//webSync
if(settings.value("Web/SyncEnabled", "false").toBool())
    webSync->syncFileAfterSave(fileName, settings.value("Web/
Address").toString());
//webSync done
```

Aki most a `WebSynchronizer` osztály két meglévő tagfüggvényében elhelyez egy-egy `qDebug()` üzenetet, az figyelemmel kísérheti, hogy milyen remekül lefutnak a tagfüggvények. Jah, kérem, hogy semmi értelmet nem csinálnak? Kicsinység!

Illetve nem is. Talán mégis inkább meg kéne őket csinálni rendesen. Kezdjük a feltöltéssel, már csak azért is, hogy utóbb legyen mit letölteni.

A feltöltés során ugyebár két fájl kell majd feltöltenünk. Az egyik maga a feladatlista-fájl, a másik az időbélyeget tartalmazó párja. Ismétlődő feladatra leltünk, azaz mehetünk tagfüggvényt írni. A neve legyen a fantáziadús `uploadFile()`. Legyen mondjuk két argumentuma, a fájlnev és a webcím. Eddig tehát a `syncFileAfterSave()` hívja kétszer az `uploadFile()`-t.

Hogy is történik maga a feltöltés? Az úgynevezett űrlap-alapú feltöltés móddal dolgozunk, amelynek a módját a 1867-es RFC írja le.²⁷

Eszerint elküldünk egy kérést a webszervernek, ebben tudtára adjuk, hogy mi épp adatot akarunk küldeni neki. Eláruljuk neki, hogy milyen határ (*boundary*) fogja jelezni az adat elejét és végét, aztán az orrára kötjük azt is, hogy milyen hosszú lesz az adat. Az adat ebben az esetben nem csak magát a fájlt jelöli, mint az hamarosan nyilvánvalóvá válik.

Az adat hosszának megadását követően pedig jön az adat: művelet kezdetét jelölő *boundary*, majd megmondjuk, hogy mit kell vele csinálni – mi majd azt mondjuk, hogy oda kell adni az **upload.php**-nek. Ezt követi megint egy *boundary*, aztán eláruljuk a fájl nevét, a fájl típusát, majd jöhet maga a fájl, végül a lezáró *boundary*.

Az előző két bekezdés a Qt esetében is élesen elválik. A kérés (a kódunkban: `request`) után következik az adattömb (`array`). Az adattömb legyártását megint külön tagfüggvényre bizzuk, így jobban elkülönül a dolog logikája. Azaz a függvényhívási lánc úgy egészül ki, hogy az `uploadFile()` tagfüggvény minden futása alkalmával egyszer meghívja a `createArray()` tagfüggvényt.

Régen nyúlkáltunk már a `ToDoList.pro` fájlban, itt az ideje némi turkálásnak ismét. Keressük meg azt a sort, amelyben a Qt keretrendszer általunk használt részeit soroljuk fel, és egészítsük ki a `network` szóval. Szóval:

```
QT += core gui network
```

A Qt-ban a hálózati műveleteket egy, a `QNetworkAccessManager` nevű osztályból példányosított objektummal szoktuk elvégeztetni. Egy példány elég az egész alkalmazásnak. A `WebSynchronizer` osztály konstruktorában példányosítsunk magunknak egyet:

```
networkManager = new QNetworkAccessManager(this);
```

A `networkManager` objektum `post()` tagfüggvényét használjuk arra, hogy elküldjük a kérést, illetve az adattömböt. A kérés `QNetworkRequest` osztályú objektum lesz, amely konstruktorának kötelező paramétere az URL, ahova a kérés irányul. Ha készen vagyunk a kérés példányosításával, be kell állítanunk a tartalomtípusra és az adattömb hosszára fejléceket, majd elküldeni az egészet a webszervernek. A webszerver válasza egy `QNetworkReply` osztályú objektumként érkezik meg.

Akkor lássuk mindezt megvalósítva:

27 A hálózati műveletek mikéntjeit RFC-nek nevezett kváziszabványokban szokás megfogalmazni. RFC-ből sok van, számmal azonosítják őket. A bennünket érdeklő RFC szövege a következő címen (is) olvasható: <http://tools.ietf.org/html/rfc1867>

```

void WebSynchronizer::uploadFile(QString fileName, QString webAddress)
{
    QByteArray array = createUploadArray(fileName);
    if(!array.isEmpty()){
        QUrl URL = QUrl("http://" + webAddress + "/upload.php");

        QNetworkRequest request(URL);
        request.setHeader(QNetworkRequest::ContentTypeHeader,
"multipart/form-data; boundary=margin");
        request.setHeader(QNetworkRequest::ContentLengthHeader,
QString::number(array.length()));

        QNetworkReply *uploadReply = networkManager->post
(request,array);

        QEventLoop loop;
        connect(uploadReply, SIGNAL(finished()),
                &loop, SLOT(quit()));
        loop.exec(); //wait until upload finished
        if(uploadReply->error())
            qDebug() << uploadReply->errorString() << uploadReply->
error();

        delete uploadReply;
    }
}

```

A tagfüggvény első sorában elkészítjük az adattömböt, benne a feltöltendő fájlal (mindjárt megmutatjuk, hogy miként). Ha a tömb nem üres – ami akkor következhet be, ha a helyi fájl nem sikerült megnyitni –, nekikezdünk a feltöltés előkészítéséhez. Legyártjuk az URL-t, majd ennek felhasználásával példányosítjuk az a `QNetworkRequest` osztályú objektumot, amelyiknek a példányosítást követő két sorban beállítjuk a fejléceit. A már sokat emlegetett *boundary* neve lesz a *margin*. Lehetne épp bármi más, az RFC csak annyit köt ki, hogy az adatban ne forduljon elő. Hát, a mi tesztadatainkban nem fog, és punctum. Figyeljük meg, ahogy az adattömb hosszát is elhelyezzük a megfelelő helyen.

Az `uploadReply` nevű mutató jelzi a webszerver válaszát a `networkManager` objektum által kezdeményezett beszélgetésre, melynek során az előkészített kérést és az adattömböt HTTP POST kérésként küldtük el a webszervernek.

A válasz nem érkezik meg azonnal, és lehet, hogy nagyon soká jön majd, mert például lassú a hálózat. Több módszer is volna arra, hogy csak akkor nyúljunk a válaszhoz, amikor már megérkezett. Mindegyik azon alapul, hogy amikor visszaért a válasz, a `QNetworkReply` objektum egy `finished()` signal-t emittál. Ezt a signal-t köthetnénk ahhoz a slot-hoz, amelyik feldolgozza a választ, de talán a Qt hálózati programozásában még nem profik számára könnyebben követhető, ha inkább indítunk egy eseményhurkot, amit az `uploadReply` objektumból érkező `finished()` signal megérkeztekor megszakítunk. Magyarán: *megvárjuk*, amíg visszaér a válasz.

Ha visszaért, akkor feldolgozzuk. Ha hibát tartalmaz, akkor mind a hibát, mind a hibakódot kiírjuk. A kiírásnak később még keresünk jobb helyet, egyelőre megteszi a `qDebug()`.

Végül töröljük a szükségtelenné vált `QNetworkReply` osztályú objektumot az `uploadReply` mutató végéről. Kifacsartuk, elvettünk mindent, amit adhatott, és most eldobjuk, igazi szívtipró módjára.

Ha értjük, hogyan működik majd a feltöltésnek ez a része, akkor lássuk a `createUploadArray()` tagfüggvényt. Lényegesen egyszerűbb lesz, ígérjük.

```
QByteArray WebSynchronizer::createUploadArray(QString fileName)
{
    QFile file(fileName);
    QByteArray array;

    if (file.open(QIODevice::ReadOnly)){
        array.append("--margin\n");
        array.append("Content-Disposition: form-data; name=\"action\"");
        array.append("\n\n");
        array.append("upload.php\n");
        array.append("--margin\n");
        array.append("Content-Disposition: form-data;");
        array.append("name=\"uploaded\"; filename=\"" + fileName + "\"\n");
        array.append("Content-Type: text/tolist\n\n");
        array.append(file.readAll());
        array.append("\n");
        array.append("--margin--\n");
    }

    return array;
}
```

A `QByteArray` remek kis osztály, de a mi szempontunkból jelenleg csak annyi az érdekes belőle, hogy tulajdonképp bájt sorozatokat tárolhatunk a belőle példányosított objektumokban. Ha sikerült megnyitni a fájlt, akkor beírunk mindenféle okosságokat az adattömb elejére, majd a `QFile::readAll()` tagfüggvény hívásával az adattömb közepére bepakoljuk az egész fájlt. Utóbb még pár okosságot fűzünk az adattömböz, és az egészet visszaadjuk a hívónak. Ha nem sikerült megnyitni a fájlt, akkor üres objektumot adunk vissza, ezt az információt az `uploadFile()` tagfüggvényben ki is használjuk.

A `syncFileAfterSave()` tagfüggvény törzse egyelőre egyetlen sor:

```
uploadFile(fileName, webAddress);
```

Mikor mindezzel elkészültünk, futtassuk a művünket. A mentés ikonra kattintva elvileg megtörténik a feltöltés – látnunk kell a webszerver naplófájljaiban is, illetve a megfelelő könyvtárban meg kell jelennie a feltöltött fájlnek.

Azért ez elég sok buktatós feladat volt, főleg azok számára, akik először telepítettek webszerveret, először futtatnak PHP-parancsfájlt. Úgyhogy ha tényleg felment a fájl, egy bátoratlan és kételkedő „Hurrrááááá!!!” igencsak helyénvaló.

Most pedig... Nem engedünk a csábításnak, nem nyargalunk letöltést írni, hanem először úgy istenigazából rendberakjuk a feltöltést.

Azzal kezdjük a nagy rendberakást, hogy a `syncFileAfterSave()` tagfüggvényben legyártjuk és feltöltetjük az időbélyeget is. A tagfüggvény egysoros törzsét egészítettük ki az alábbi néhány sorral:

```

QString timeStampFileName = fileName + ".timestamp";
QFile file(timeStampFileName);
if (file.open(QFile::WriteOnly | QFile::Truncate | QFile::Text)) {
    QTextStream out(&file);
    QFileInfo fi(fileName);
    out << fi.lastModified().toString(Qt::ISODate);
}
if(!file.error())
    uploadFile(timeStampFileName, webAddress);

```

Elvileg egyetlen újdonságot találunk a fentiekben. Az időbélyeget tartalmazó fájl nevét az eredeti fájlnev kiegészítésével képezzük. Előbb legyártjuk itt helyben a fájlt, amelybe a QFileInfo osztály lastModified() tagfüggvényével lekérdezett QDateTime osztályú választ írjuk bele, de a beleírás előtt a választ karakterlánccá alakítjuk, mégpedig az ISO által meghatározott formátumúra. Ha a fájl adathordozóra való írása sikeres volt, az uploadFile() tagfüggvény második hívásával utánaküldjük a feladatlistát tartalmazó fájlunk.

A renderakást ott folytatjuk, hogy kiépítünk egy mechanizmust, melynek használatával a felhasználót a főablak állapotsorában értesítjük a hálózati műveletek sikeréről, illetőleg balsikeréről.

A WebSynchronzier osztályú webSync objektum a fileOperator objektum privát objektuma, azaz a főablak nem látja, így a singal-jait sem hallhatja. Készítsük hát fel a FileOperator osztályt az üzenet továbbítására. Definiálunk egy tulajdonságot, úgy, ahogy azt nem is olyan rég megtanultuk:

```

Q_PROPERTY(QString lastWebSyncMessage READ lastWebSyncMessage
WRITE setLastWebSyncMessage NOTIFY lastWebSyncMessageChanged)

```

Elhelyezzük az m_lastWebSyncMessage nevű, QString osztályú objektumot tárolni képes provát változót, deklaráljuk a void lastWebSyncMessageChanged(QString message); signal-t, majd megírjuk a tagfüggvényeket:

```

QString FileOperator::lastWebSyncMessage()
{
    return m_lastWebSyncMessage;
}

```

```

void FileOperator::setLastWebSyncMessage(QString message)
{
    m_lastWebSyncMessage = message;
    emit lastWebSyncMessageChanged(message);
}

```

Fontos, hogy a beállítófüggvényt ne sima függvényként, hanem publikus slot-ként adjuk meg, mert majd ezt a tagfüggvényt fogja hívni a webSync objektum.

A főablakot készítsük fel a fileOperator objektum felől érkező üzenet vételére, illetve az állapotsoron való megjelenítésére. Szükségünk lesz egy publikus slot-ra:

```

void MainWindow::lastWebSyncMessageChanged(QString message)
{
    ui->statusBar->showMessage(message, 4000);
}

```

A szám az üzenet után azt mondja meg ezredmásodpercben, hogy mennyi idő után tűnjön el az üzenet az állapotsorról. Ha 0-t adunk meg, akkor a következő üzenetig ott marad. A főablak `preparationAtStartup()` tagfüggvényében elhelyezzük a `connect` utasítást:

```
connect(fileOperator, SIGNAL(lastWebSyncMessageChanged(QString)),
        this, SLOT(lastWebSyncMessageChanged(QString)));
```

Mostanra a `fileOperator` objektum és a főablak között zavartalan az információáramlás. A `WebSynchronizer` osztályban ténykedünk tovább. Deklarálunk egy signal-t:

```
void syncReport(QString);
```

Eddig az `uploadFile()` tagfüggvényben a hálózati hibát egy `qDebug()`-kiírással jeleztük. Ezt cseéljük le most, ráadásul akkor is beszélünk, ha minden rendben ment. Íme a teljes `if`-utasítás:

```
if(uploadReply->error())
    emit syncReport(uploadReply->errorString() + " (code:" +
        QString::number(uploadReply->error()) + ")");
else if(fi.suffix() != "timestamp")
    emit syncReport("Network upload (" + fi.fileName() + ")
just happened.");
```

Ha volt hiba, akkor kiírjuk, ha nem, akkor megmondjuk, hogy mit töltöttünk fel, de csak akkor, ha feltöltött fájl kiterjesztése nem „`timestamp`”. Azért hallgatunk mélyen az időbélyeg-fájl feltöltéséről, mert rögtön a feladatlista-fájl feltöltése után történik meg, és így az üzenetek olyan gyorsan követnék egymást, hogy nem volna időnk megnézni az állapotsoron az első, számunkra fontosabb üzenetet.

A `fi`, amelyet használunk, egy `QFileInfo` objektum, amit persze a tagfüggvény elején inicializálni kell:

```
QFileInfo fi(fileName);
```

A kiterjesztés ellenőrzésén felül azért van rá szükségünk, hogy a `fileName` változóból gyorsan és fájdalommentesen le tudjuk csipptenteni az elérési utat – ha nem csipptentenék le, nem férne el az állapotsoron az üzenet.

Az egész hálózati művelethez csak akkor fogunk hozzá a tagfüggvényben, ha nem üres a `createUploadArray()` tagfüggvénytől visszkapott tömb. Erre szolgál a tagfüggvény eleje felé található

```
if(!array.isEmpty())
```

ellenőrzés. Említettük már, hogy üres tömböt akkor kapunk vissza, ha nem sikerült a helyi fájl megnyitása. Ha ilyen malőr történne, azt is tudjuk közölni a felhasználóval, mégpedig úgy, hogy megírjuk a fenti `if` utasítás `else`-ágát:

```
}else
    emit syncReport("Local file error while syncing " +
        fi.fileName());
```

Kicsit tesztelhetjük a programot: érdemes rossz URL-t megadni a beállítások ablakban, vagy lekapcsolni a webszerveret, vagy eltenni az `upload.php`-t máshova. Ha minden jól megy, sok szép új hálózati hibaüzenetet és -kódot ismerünk majd meg. Az izgalmasabbakat felolvashatjuk a hörcsögnek is, miközben a félig becsukott markunkból figyeli a monitort, és csak a bajszocskája mozog izgalmában. A feltöltéssel végeztünk, kezdődhet a

13.4. Letöltés a WebSynchronizer osztállyal – a QNetworkReply osztály és a fájlok kicsomagolása

Az a tervünk, hogy a `syncFileBeforeOpen()` tagfüggvénnyel letöltetjük az időbélyeg-fájlt, s a benne tárolt adatot összehasonlítjuk a helyi fájléval. Ha a hálózaton lévő fájl az újabb, akkor letöltjük, és felülírjuk vele a helyit. A `fileOperator` osztály erről mit sem tud: egyszerűen megnyitja az ott lévő fájlt, amit közben vagy újabbra cseréltünk, vagy sem.

```
void WebSynchronizer::syncFileBeforeOpen(QString fileName, QString
webAddress)
{
    QString timeStampFileName = fileName + ".timestamp";
    downloadFile(timeStampFileName, webAddress);
    QFile file(timeStampFileName);
    if (file.open(QIODevice::ReadOnly | QIODevice::Text)){
        QTextStream in(&file);
        QString timestamp = in.readAll();
        if(!file.error()){
            QFileInfo fi(fileName);
            if(fi.lastModified() < QDateTime::fromString(timestamp ,
Qt::ISODate))
                downloadFile(fileName, webAddress);
        }
    }
}
```

A szokásos módi szerint előállítjuk az időbélyeg-fájl nevét, majd letöltjük – mindjárt megmutatjuk, hogyan. Ha leért, megnyitjuk, és beolvassuk a tartalmát. A `QDateTime::fromString()` tagfüggvény a fájl tartalmából állít elő `QDateTime` osztályú objektumot, amit össze lehet hasonlítani a `QFileInfo::lastModified()` tagfüggvény ugyanilyen osztályú kimenetével.

A letöltést végző tagfüggvény az alábbi:

```
void WebSynchronizer::downloadFile(QString fileName, QString
webAddress)
{
    QFileInfo fi(fileName);
    QUrl URL = QUrl("http://" + webAddress + "/" + fi.fileName());

    QNetworkRequest request(URL);
    QNetworkReply *downloadReply = networkManager->get(request);

    QEventLoop loop;
    connect(downloadReply, SIGNAL(finished()),
            &loop, SLOT(quit()));
    loop.exec(); //wait until download finished
    if(downloadReply->error())
        emit syncReport(downloadReply->errorString() + " (code:" +
QString::number(downloadReply->error()) + ")");
    else{
        if(fi.suffix() != "timestamp")
            emit syncReport("Network download (" + fileName + ") just
happened."); //we don't want succesful timestamp download reports
        QFile file(fileName);
        if(file.open(QIODevice::WriteOnly))
            file.write(downloadReply->readAll());
        else
            emit syncReport("Local file error while syncing " +
fi.fileName());
    }

    delete downloadReply;
}
```

A legtöbb része már ismerős. Előállítjuk az URL-t, és a használatával példányosítunk magunknak `QNetworkRequest` osztályú kérést. Ezúttal a `networkManager` objektum `get()` tagfüggvényét használjuk, amelynek egyetlen paramétere a kérés. A szerver válasza ezúttal is `QNetworkReply` osztályú objektumként érkezik. Igyekszünk okosabbak lenni az esetleges hibaüzenetek alapján, de ha egy mód van rá, akkor kisedjük a fájlt a válaszból. Ez meglepően egyszerű dolog: a `readAll()` tagfüggvényre rá lehet bízni az egészet.

Elkészültünk: tudunk fájlt webszerverre feltölteni, és onnan letölteni. Amennyiben azt szeretnénk, hogy valós életben is használható hálózati szinkronizációnk legyen, még rengeteg fejlesztenivalónk van, de erről a fejezet elején már elméltünk.

14. TÖBBSZÁLÚ PROGRAMOT ÍRUNK

Ebben a fejezetben kivételesen nem rögtön a ToDoList-tet megyünk csinosítgatni, hanem előbb elfilózzgatunk a többszálúság szépségeiről, aztán készítünk egy példaalkalmazást, és csak mindezek befejeztével térünk vissza kis kedvencünkhöz, a feladatlista-nyilvántartó alkalmazásunkhoz.

Manapság a többszálúság, a többszálú programozás ismerete olyannyira alapkövetelménnyé vált, hogy még a strandkönyvekbe is bekerül a téma. Meggondolandó azonban, hogy a mi programunkban tényleg szükség van-e rá, ugyanis a többszálú programok hamar válnak nagyon bonyolulttá.

Egy többmagos processzoron – és napjainkban nehéz olyat találni, amelyik nem ilyen – a szálak futása tényleg történhet párhuzamosan. Ha ilyen esetben két szál is nyúlkal ugyanabban az objektumban, az könnyen bajhoz vezet. A probléma illusztrálására tekintsünk egy olyan osztályt, amelyikben van egy karakterlánc adattag. Ha a két szál egyszerre matat a karakterlánccal, nem biztos, hogy mire az egyik szálban újra hozzányúlunk a karakterlánchoz, azt találjuk ott, amit ott hagytunk. A helyzet lehet még ennél is rosszabb, ugyanis a karakterlánc módosítása nem elemi művelet. Mi történik, ha az egyik szál épp a karakterlánc módosításának kellős közepén tart, amikor a másik is elkezdi módosítani ugyanazt a karakterláncot? Leginkább talán még a jócska tudják elmondani, de a programozók aligha. Így aztán feltalálták az osztálytagok zárolását – ezt végzi a `QMutex` és a `QMutexLocker` osztály – ami az adattagokhoz, tagfüggvényekhez való hozzáféréseket sorba állítja, és vigyáz, hogy egyszerre csak egy matatás történjen.

A Qt biztosít olyan osztályokat is, amelyek szálbiztosak (angolul: thread-safe). A nagy többség azonban nem ilyen, ugyanis a folyamatos zárolgatás-feloldozgatás a legtöbb esetben csak fölösleges plusz terhelést jelentene. Ilyen osztályok használatakor magunknak kell gondoskodni a zárolásokról, vagy olyan programot kell írunk, amelynek a szálai nem turkálnak ugyanabban az objektumban. Vagy olyat, amelyik nem többszálú.

Többszálú programot gyakran írunk olyankor, amikor a háttérben valamilyen komoly adatfeldolgozás zajlik, de azt szeretnénk, hogy a grafikus felület ne merevedjen le az adatfeldolgozás időtartamára. Persze ilyenkor sem kell mindig többszálú programot írunk. Az adatfeldolgozó ciklusban ugyanis időről időre hívható a `QEventLoop::processEvents()` tagfüggvény, ami az időközben felgyűlt események feldolgozására ad lehetőséget.

Még egy fontos megemlítenivalónk van: a `QThread` osztály használatát sok esetben mind internetes források, mind szakkönyvek a most ismertetésre kerülő módszerhez képest teljesen másképp írják le – a lényeg ezekben a forrásokban többnyire az lesz, hogy ha több szálat akarunk használni, akkor leszármazott osztályt kell készítenünk a `QThread`-ből, és újra meg kell valósítanunk a `run()` tagfüggvényt. Ez a megoldás mára elavult és ellenjavallt.

Nos, ha ennyi bevezető és eltántorító dolog után még mindig többszálú program írására adnánk a fejünket, akkor most nekikezdünk, de előtte még indítsunk egy külön hörcsögsimogató folyamatot: egy darabig most nélkülünk kell ellennie.

14.1. Tanulmányprogram a szálak tanulmányozására

A programunk először természetesen egyszálú lesz, még akkor is, ha már most a `threads` nevet adjuk neki. Helyezzünk el a főablakon `QLabel`-t `showNumber` néven, két `QLineEdit`-et

`startEdit` és `stopEdit` néven, valamint két nyomógombot, `startButton` és `stopButton` néven. Talán sejtjük, hogy valami számlálós dolog lesz benne, aminek megtudjuk adni a kezdő és befejező értékét. Nos, nem is csatlakozunk sejtésünkben.

A számlálót külön osztályban helyezzük el. Az osztály egy tagváltozóból, két slot-ból és egy signal-ból áll. Az egyik slot indítja a számlálót, és a benne lévő ciklus a főablak két `QLineEdit` elemében megadott érték között fut. A ciklus olyan gyors volna, hogy nem látnánk, ahogy a számok változnak, így némi késleltetést helyezünk el benne egy `QEventLoop` osztályú objektummal – emlékszünk még rá? Ilyen objektummal oldottuk meg azt is, hogy a `ToDoList` megvárja a hálózati művelet végét. A `QEventLoop`-ot akkor a `QNetworkReply` osztályú hálózati válasz megérkeztekor szakítottuk meg, most pedig egy `QTimer` osztályú objektum `timeout()` signal-ja lesz a jel számára, hogy elég volt belőle.

A másik slot a tagváltozót állítja igaz értékre. Ha a tagváltozó igazzá válik, még azelőtt megszakítjuk az első slot ciklusát, hogy az elszámolt volna a célértékig.

Íme a két slot:

```
void Counter::startCounter(int start, int stop)
{
    QTimer *timer = new QTimer(this);
    timer->setSingleShot(true);
    m_stop = false;
    for(int i = start; i < stop; i++){
        if(m_stop)
            break;
        timer->start(100);
        QEventLoop loop;
        connect(timer, SIGNAL(timeout()),
                &loop, SLOT(quit()));
        loop.exec();
        emit numberChanged(i);
    }
    delete timer;
}
```

```
void Counter::setStop(bool stop)
{
    m_stop = stop;
}
```

A főablak lényegében három slot-ból áll. Az egyik a számláló `numberChanged()` signal-ja hatására módosítja a `showNumber` nevű objektum kiírását. A másik a `stop` gomb lenyomását érzékeli, és kibocsátja azt a signal-t, amelynek hatására a `Counter::setStop()` fut le. Ezt a két slot-ot gyorsan meg is mutatjuk, aztán rátérünk a harmadikra, a lényegre.

```
void MainWindow::onNumberChanged(int number)
{
    ui->showNumber->setText(QString::number(number));
}
void MainWindow::on_stopButton_clicked()
{
    emit stopCounter(true);
}
```

Szóval a harmadik slot, a lényeg. Íme:

```
void MainWindow::on_startButton_clicked()
{
    Counter *counter = new Counter(this);
    connect(this, SIGNAL(startCounter(int,int)), counter,
    SLOT(startCounter(int,int)));
    connect(this, SIGNAL(stopCounter(bool)), counter,
    SLOT(setStop(bool)));
    connect(counter, SIGNAL(numberChanged(int)), this,
    SLOT(onNumberChanged(int)));
    emit startCounter(ui->startEdit->text().toInt(), ui->stopEdit->
    text().toInt());
    delete counter;
}
```

Létrehozunk benne egy számlálót. Úgy teszünk, mint ha nem volna elegendő a stack-en létrehoznunk, és a heap-re tesszük. A következő három utasítással bekötögetjük az új objektum slot-jait és signal-ját, majd elindítjuk a számlálót. Az utolsó utasítás pedig törli a számlálót. A mű kész, de jobban megfigyelhető a működése, ha az emit utasítás elé és mögé elhelyezünk egy-egy qDebug() utasítást, amelyik a signal emittálását, illetve az objektum törlését jelzi.

Közben – Linux operációs rendszeren – van értelme kiadni a

```
watch -n 1 "ps H -C threads -o 'pid tid cmd comm'"
```

parancsot. A **threads** a futó bináris fájlunk neve, a **ps** parancs emígyen felparaméterezve megmutatja többek között a folyamatazonosítót (pid) és a szárazonosítót (tid). A **watch** parancs azt oldja meg, hogy másodpercenként automatikusan újra lefusson a **ps** parancs, így a világképünk is, legalábbis, ami a szálatkat illeti. Némileg megütődve láthatjuk, hogy a programunk máris több szál megjelenésével jár, de hogy azokat nem mi követtük el, az tuti.

Ha kitanulmányoztuk magunkat, itt az ideje, hogy a számlálót üzemeltető objektumot külön szádra helyezzük. A szál számunkra egy QThread osztályú burkoló (angolul: wrapper) objektumon keresztül lesz elérhető. Az egyik problémánkat az okozza majd, hogy van módunk arra, hogy egy objektumot másik szádra tegyünk át, de onnan *visszavenni* már nem annyira egyszerű. A másik nyavalyánk meg azzal kapcsolatos, hogy a másik szálon lévő objektumot mikor kell törölni? És mikor törölhető maga a szál?

No, majd meglátjuk. Helyezzük el a `mainwindow.cpp` elején a `<QThread>` fejléctet, majd a számláló példányosítását követően példányosítsunk magunknak szalat is:

```
QThread *thread = new QThread;
```

Ahogy ezzel megvagyunk, helyezzük át a számlálót az új szádra:

```
counter->moveToThread(thread);
```

Ezután következhet a három connect, majd a szál elindítása:

```
thread->start();
```

amit a már eddig is meglévő, `startCounter()` signal-t emittáló sor, illetve a `counter` objektumot törölő sor követ. Tudomásul vesszük, hogy a `thread` objektumot nem töröltük, és ez memóriaszivárgással jár, de egyelőre kibírjuk. Indulhat a program. Lefordul, és futni kezd, de a számláló indításakor kapunk egy olyan üzenetet, hogy:

QObject::moveToThread: Cannot move objects with a parent

és a counter vigyorgva marad az eredeti szálon, amit egyébként megfelelően elhelyezett `qDebug() << counter->thread();` utasításokkal magunk is ellenőrizhetünk: a `QObject::thread()` tagfüggvény való a használt szál kiderítésére. Akkor hát a counter objektumot példányosító utasításból töröljük a `this` szülőt, ami amúgy is csak arra kellett volna, hogy ne kelljen kézzel törölni az objektumot.

A counter objektum szépen áthelyeződik a másik szálra – ez látható a `QDebug()` sorok kimenetéből –, de utóbb úgy elszáll a programunk, hogy Lindbergh repülőgépe sem különben. Miért is? Mert azáltal, hogy az objektumunk másik szálra került, kikerült ennek a szálnak a főciklusából, és a `delete` nem várja meg, amíg lefut a számláló slot, hanem törli az objektumot. Ha mi meg a `delete` sort töröljük, akkor nincs gáz a program futásával, de van egy olyan objektumunk (a counter), aminek mostanra se szülője, se törlése, azaz itt van még egy memória-szivárgás – és akkor még mindig nem törődtünk a thread objektum hasonló nyűgjeivel.

Szerencsére azonban létezik néhány remek kis signal és slot, amivel minden, a fentiekben vázolt problémánkon úrrá lehetünk. A `QObject::deleteLater()` slot dokumentációja szerint:

A Qt 4.8 óta, amennyiben a `deleteLater()` tagfüggvényt egy futó eseményciklus nélküli számban élő objektumon hívjuk, a szál befejeződésekor az objektum megsemmisül.

Eddig pöpec. Most az a kérdés, hogyan állítjuk le a számban az eseményciklust. Erre lesz jó a `QThread::quit()` slot, amely:

A szál eseményciklusát arra kéri, hogy 0 (sikert jelző) visszatérési értékkel lépjen ki.

Azaz eddig úgy áll a dolgunk, hogy leállítjuk a szál eseményciklusát, aztán hívjuk a `counter->deleteLater()` tagfüggvényt, ami előkészíti a counter objektum megsemmisítését. Már csak a szál megsemmisítéséről kell gondoskodni. Erre megint a `deleteLater()` tagfüggvényt használjuk, de ezúttal a thread objektumon hívva, ugyanis a dokumentáció szerint:

Az objektum törlődik, amikor a vezérlés visszatér az eseményciklushoz.

Mármost ahhoz az eseményciklushoz, amelyikből a `deleteLater()`-t hívtuk.

Már csak megfelelő signal-okat kell találnunk a slot-jainkhoz. A `QThread::quit()` slot hívását végző signal-t nekünk kell előállítanunk, úgyhogy a Counter osztályban deklarálunk egyet magunknak, `counterFinished()` néven. A signal emittálását időzítsük akkorra, amikor a `Counter::startCounter()` slot-ban lévő ciklus már lefutott, vagy kiléptünk belőle, azaz legyen ez a slot törzsének utolsó sora. Kiadjuk a megfelelő connect utasítást is a főablakban, olyankor, amikor már van szál is, meg counter is:

```
connect(counter, SIGNAL(counterFinished()), thread, SLOT(quit()));
```

Mi legyen az a signal, ami a két objektum `deleteLater()` tagfüggvényét hívja? Nos, szál emittál magából egy `finished()` signal-t, közvetlenül a végrehajtás leállítását megelőzően. Ezt fogjuk használni mind a counter, mind a thread törlésére:

```
connect(thread, SIGNAL(finished()), counter, SLOT(deleteLater()));
connect(thread, SIGNAL(finished()), thread, SLOT(deleteLater()));
```

A művünk így már rendben van, de ha szeretnénk látni is, hogy tényleg kipuштul a két

objektum, érdemes az objektumok `destroyed()` signal-jához írni egy akkora slot-ot, amelyik egy `QDebug()` üzenettel értesít bennünket az objektum elhalálzásáról. Csak a `connect`-utasításokat közöljük:

```
connect(thread, SIGNAL(destroyed()), this, SLOT(onThreadDestroyed()));
connect(counter, SIGNAL(destroyed()), this, SLOT(onCounterDestroyed()));
```

A programunk szépen fut, objektumok születnek és pusztulnak. Van egy implementációs hibánk, nevezetesen az, hogy a számlálót akárhányszor elindíthatjuk. Azonban, tekintve, hogy ez a program csak illusztráció, nem is bájuk. A fenti `ps` parancsot futtatva azt látjuk, hogy többszöri számláló-indításkor szépen szaporodnak a `QThread` nevű szálak. Ha van kedvünk, a `thread->setObjectName("worker");` utasítással elnevezhetjük a szálakat, esetleg használhatjuk a `thread->setObjectName("worker" + QString::number((quintptr) thread).toLatin1());` utasítást, amellyel a szál címéből képzünk a szálaknak egyedi nevet.

14.2. Szálakat építünk a *ToDoList-be*

Talán nem meglepő, ha frissen megszerzett tudásunkat szeretnénk dédelgetett projektcskénkbe is beépíteni. Nem igazán kívánczik hosszú és kemény munkát igénylő feladat, olyan, ami igazán külön szálnak való. Sebj, majd külön szálra tesszük a webes szinkronizációt – így ha az urambocsá' hosszan eltart, mert mondjuk ezermillió feladat van felvéve a listába, és lassú a net, akkor mehet a háttérben, mi pedig dolgozhatunk tovább a a programban.

Elsőként a `WebSynchronizer` osztályon ejtjük meg a szükséges átalakításokat. Megfogadva egy jótanácsot²⁸, a konstruktorából eltávolítjuk azt az utasítást, amely a heap-en hoz létre objektumot, ugyanis az ilyen objektum még azelőtt létrejön, hogy a `webSync` objektum átkerülne az új szálra. A tényállásból pedig az következik, hogy így a konstruktorban létrehozott objektum marad a főszálon, azaz a két objektumunk két külön szálon lesz. Az álmoskönyv szerint ez nem szerencsés. Persze `QNetworkAccessManager`-re szükségünk van, de majd készítünk magunknak az adott tagfüggvényben. Olvastuk azt is, hogy alkalmazásonként egyetlen ilyen osztályú objektum épp elég, de végső soron egy szálaban így is csak egy lesz. Írjuk hát ki a `WebSynchronizer` osztály konstruktorát, szüntessük meg a fölöslegessé vált deklarációt, cserébe a `downloadFile()` és az `uploadFile()` tagfüggvényben helyezzük el az alábbi sort:

```
QNetworkAccessManager networkManager;
```

A két szükséges helyen cseréljük le a nyilat (`->`) pontra, hiszen immáron nem a heap-re példányosítunk magunknak hálózatkezelőt.

Következő feladatunk a `syncFileBeforeOpen()` és a `syncFileAfterSave()` tagfüggvényt slot-tá alakítani, ami nem jelent mást, mint a két sor áthelyezését a `websynchronizer.h` fájlban. Szükségünk lesz még egy `syncFinished()` signal-ra, deklaráljuk hát, és helyezzük el az emittálásra való sort a két, újonnan slot-tá avasztált függvény törzsének végén. Ezzel a `WebSynchronizer` osztály módosításaival végeztünk is. Az osztály működésében a `FileOperator` osztály szempontjából egyelőre semmi nem változott, azaz ezen a ponton érdemes lefordítanunk és kipróbálnunk a programunkat – már most derüljön ki, ha elrontottunk valamit.

28 <http://mayaposch.wordpress.com/2011/11/01/how-to-really-truly-use-qthreads-the-full-explanation/>

Ha minden rendben fut, akkor elkezdjük a `FileOperator` osztály módosítását is. Hasonló módon közelítünk a problémához, mint a `threads` nevű példaprogramunknál, azaz a megfelelő tagfüggvény, illetve újabban slot hívását signal-hoz kötjük. Még a signal emittálása előtt létrehozzuk az új `WebSynchronizer` osztályú objektumunkat, áttesszük az új szálra, és kialakítjuk a megfelelő signal-slot kapcsolatokat. A megfelelő slot indítását végző signal ugyebár ugyanolyan paraméterlistájú, mint a slot, azaz esetünkben `QString`, `QString`. Az első paraméter a szinkronizálandó fájl neve, a második a webcím, ahova-ahonnan szinkronizálunk.

A két slot hívását egyazon signal-lal is elvégezhetjük, csak arra kell majd odafigyelni, hogy az épp aktuális, `WebSynchronizer` osztályú objektumunk melyik slot-jához kötjük a signal-t. Deklaráljuk hát a `FileOperator` osztály `startSync(QString, QString)` signal-ját.

Ahol eddig a `webSync` objektum `syncFileAfterSave()`, illetve `syncFileBeforeOpen()` tagfüggvényét hívtuk, helyezzünk el egy új hívást a `FileOperator` osztály saját tagfüggvényére. Az első esetben a

```
startWebSyncThread("save", fileName, settings.value("Web/Address").toString());
```

a második esetben pedig a

```
startWebSyncThread("open", fileName, settings.value("Web/Address").toString());
```

hívással cseréljük le. Írjuk meg gyorsan ezt a remek kis szálkezelőt:

```
void FileOperator::startWebSyncThread(QString job, QString fileName,
    QString webAddress)
{
    QThread *thread = new QThread;
    WebSynchronizer *webSync = new WebSynchronizer();
    webSync->moveToThread(thread);

    connect(webSync, SIGNAL(syncFinished()), thread, SLOT(quit()));
    connect(thread, SIGNAL(finished()), webSync, SLOT(deleteLater()));
    connect(thread, SIGNAL(finished()), thread, SLOT(deleteLater()));

    connect(webSync, SIGNAL(syncReport(QString)), this,
        SLOT(setLastWebSyncMessage(QString)));
    if(job == "open")
        connect(this, SIGNAL(startSync(QString, QString)), webSync,
            SLOT(syncFileBeforeOpen(QString,QString)));
    else if(job == "save")
        connect(this, SIGNAL(startSync(QString, QString)), webSync,
            SLOT(syncFileAfterSave(QString,QString)));
    thread->start();

    emit startSync(fileName, webAddress);
}
```

A szokásos módon létrehozuk a szálát, majd példányosítunk magunknak `WebSynchronizer` osztályú objektumot (ne feledjük a konstruktorban lévő példányosítást, törölni, az ottani `connect` pedig jól jön majd itt). A kész objektumot áttesszük az új szálra, a

következő három `connect` pedig már ismerős: ezek az utasítások gondoskodnak a `webSync` és a `thread` objektum törléséről, ha eljön az ideje.

Ezután a konstruktorból áthelyezett `connect` következik, majd a tagfüggvény első paramétere (`open` vagy `save`) alapján eldöntjük, hogy melyik slot-ot indítjuk el majd a `startSync()` signal emittálásával.

Végül elindítjuk az új szálat és emittáljuk signal-t.

Megvagyunk, kész a többszálú `ToDoList`. Már a fejezet elején jeleztük, hogy nem minden esetben éri meg többszálúvá alakítani a programunkat: az egészen szélsőséges helyzetektől eltekintve most is csak lassítottunk rajta. Cserébe viszont van elképzelésünk, hogy mi a teendő, ha egy szép napon olyan programot kell írunk, amelyik szép méretes mappákban számol majd minden fájlra külön SSHA-ellenőrzőösszeget.

15. XML-FÁJLBA MENTJÜK A FELADATLISTÁT

A legtöbb programozó megegyezik abban a kérdésben, hogy előfordulnak olyan esetek, amikor célszerű lehet ezt-azt XML formátumban menteni. Az, hogy *pontosan melyek* ezek az esetek, már késhegyig menő viták tárgyát képezi. A vita természetesen nem dőlhet el addig, amíg a legfőbb hozzáértő – a hörcsög – állást nem foglalt, azaz nem most várható, hogy mi, egyszerű földi halandók is megtudjuk a tutit.

Ebben a röpfeszletben a `FileOperator` osztályunk átírásával megvalósítjuk azt, hogy a `ToDoList` ne egyszerű szövegfájlba mentse a feladatlistát, hanem egy remekbe szabott XML-dokumentumba. Ahogy az egy jó strandkönyvhöz illik, nem terheljük az Olvasó lelkivilágát az XML, illetve a Qt XML-lehetőségei teljes mélységével. Az XML akkora téma, hogy több száz oldalas könyveket írnak róla, nekünk meg csak pár oldalunk van a témára.

Amit mindenképp tudnunk kell az XML-ről, azt most gyorsan összefoglaljuk. Az XML fájl lényegében szövegfájl, amelyben többnyire „<” és „>” jel közé kerülnek a *jelölők*, ami meg nem jelölő, az sima szöveggént van jelen. Az XML dokumentum elemekből áll, amelyek bizonyos esetekben startcímkevel kezdődnek és stopcímkevel végződnek, az úgynevezett üres elemek viszont egyetlen címkéből állnak. Minden elemnek lehetnek jellemzői. Minden XML dokumentum pontosan egy gyökérelemet tartalmaz. Minden elemnek lehetnek gyermekelemei.

A mi feladatlista-fájlunk így fog kinézni:

```
<Tasks>
  <Task job="egyik teendő" category="egyik kategória"/>
  <Task job="másik teendő" category="másik kategória"/>
  <Task job="harmadik teendő" category="egyik kategória"/>
</Tasks>
```

Ebben a dokumentumban a `<Tasks>` a gyökérelem nyitócímkeje, a `</Tasks>` a gyökérelem zárócímkeje. A gyökérelem gyermekelemei a `<Task />` elemek, amelyek üres elemek – nincs bennük igazi tartalom, szöveg. Üres elemek lévén egyetlen címkéből állnak. Bár tartalmuk nincs, vannak jellemzőik, mégpedig mindegyiknek egy `job` és egy `category` jellemzője. A jellemzők értéke az a szöveg, amelyet aztán a programunk felhasználói felületén mi is látunk, megadunk, módosítunk, törölünk.

A DOM, azaz Document Object Model – kissé pongyola, de számunkra most használható megfogalmazásban – az XML dokumentumok memóriabeli reprezentációja.

XML-gyorstalpalónk végére érve még elmondjuk, hogy a fejezetben a `ToDoList`-nek abból a változatából indulunk ki amikor már modellben tároljuk az adatokat, de még sem webes szinkronizáció, sem többszálúság nem került a programba. Aki időközben elhányta ezt a változatot, az természetesen letöltheti a könyv webhelyéről.

Mi sem mutatja jobban, hogy az XML mekkora téma, hogy ismét a `ToDoList.pro` fájl megnyitásával kezdjük a munkát, és elhelyezzük az xml bejegyzést:

```
QT += core gui xml
```

Eltételezve három nüansznyi módosítástól, a `FileOperator` osztálynak abban két tagfüggvényében fogunk piszkálni, amelyek a tényleges fájlműveletekért felelősek, azaz a

`performFileOpenOperation()` és a `performFileSaveOperation()` függvényben. Szokás szerint a mentéssel kezdjük, hogy legyen mit betölteni.

Eddig a modellből olvastuk ki az adatokat, és a kiolvasást követően fájlba írtuk őket. Ezt a feladatot látja el a mentésért felelős tagfüggvény közepén trónoló ciklus. Egyesével végignyálazza a modell sorait, és amennyiben a teendő nem üres, mind a teendőt, mind a kategóriát beírja a fájlba.

Az XML-dokumentumok mentése másképp történik: egy teljes utasítás szükséges a kiírásukhoz. Ha feltételezzük, hogy a mentendő dokumentum DOM-ja, a mi olvasatunkban memóriabéli reprezentációja a `documentToSave` objektumban rendelkezésre áll, akkor a mentést végző utasítás ennyi:

```
out << documentToSave.toString();
```

Le is cserélhetjük ezzel az utasítással az előbb emlegetett ciklust. Csak az egyértelműsítés kedvéért jelezzük meg, hogy az új sor a `QTextStream out(&file);` sor utánra és az `if(!file.error())` sor elé kerül.

Persze ezzel még nincs kész a dolog, hiszen az imént abból a feltételezésből indultunk ki, hogy a `documentToSave` objektum rendelkezésünkre áll, holott nem ez a helyzet. Valahol az imént beszúrt sor előtt helyezzük el a következő értékadást, de csak a `<QDomDocument>` fejléc megadását követően:

```
QDomDocument documentToSave = createXmlDocumentFromModel();
```

Azaz amit nyertünk a réven, veszítjük a vámon: nem kell ugyan pizmognunk a mentéssel, de elő kell állítanunk a modellben lévő adatok alapján a DOM-ot. Milyen lépésekből áll a folyamat? Létrehozunk egy üres DOM-ot, azaz `QDomDocument` osztályú objektumot, felvesszük a gyökérelemét, majd a gyökérelem gyermekeiként megadjuk az alkalmazásban használt modellünk egyes sorai alapján előállított elemeket. Mindez Qt-ul:

```
QDomDocument FileOperator::createXmlDocumentFromModel()
{
    QDomDocument xmlDocument;
    QDomElement rootElement = xmlDocument.createElement("Tasks");
    xmlDocument.appendChild(rootElement);
    for(int i = 0; i < model->rowCount(); i++){
        QString job = model->index(i,0).data().toString().simplified();
        if(!job.isEmpty()){
            QString category = model->index(i,1).data().toString().simplified();
            QDomElement task = xmlDocument.createElement("Task");
            task.setAttribute("job", job);
            task.setAttribute("category", category);
            rootElement.appendChild(task);
        }
    }
    return xmlDocument;
}
```

Figyeljük meg, hogy az elem létrehozása sem a gyökérelem, sem az egyes `task` elemek esetében nem egyenértékű az elem felvételével a DOM-ba. Az elem előállítás a `createElement()`, a felvétele az `appendChild()` tagfüggvény feladata. Látható az is, hogy a kész `task` elemre még a DOM-ba való felvételét megelőzően rábiggyesztjük a jellemzőket. Az

XML-be való mentés működik, de mielőtt kipróbáljuk, idézzük eszünkbe, hogy szó volt három nüansznyi módosításról. A kiterjesztéseket kell átírogatnunk, a mentés esetében kétszer is – mert a `QFileDialog` statikus tagfüggvényén felül még az automatikus kiterjesztésadást is meg kell változtatni – illetve a betöltésnél. Kipróbálhatjuk a mentést. Ha nem akarjuk, hogy a tárolt beállítások közötti `LastFileName` gondot okozzon, használhatunk épp `LastXMLFileName` kulcsot is, nem csak itt, de a `mainwindow.cpp` és a `modelmanager.cpp` fájlban is.

Talán nem vagyunk meglepve, ha kiderül, hogy a betöltés hasonló szépségekkel és csúnyaságokkal jár: ezúttal is borzasztó egyszerű dolog a fájl betöltése a DOM-ba, de jön a vám: ránk vár a DOM bejárása és a modell feltöltése. A `performFileOpenOperation()` tagfüggvény közepén lévő `QTextStream`-sort és a `while`-ciklust cseréljük fel a következő utasítással:

```
documentLoaded.setContent(&file);
```

Ennyit a betöltésről. Persze kicsit – mondjuk két sorral – korábban példányosítani kell magunknak DOM-ot:

```
QDomDocument documentLoaded;
```

Sikeres fájlművelet esetén már eddig is elvégeztünk négy sorban három műveletet: beállítottuk a fájlnevet, szóltunk, hogy a modell és a fájl tartalma mostantól megint azonos, és új sort helyeztünk a modell végére. Most még, talán a második helyen, fel kell töltenünk a modellt. Helyezzük el a következő függvényhívást:

```
populateModelFromXml(documentLoaded);
```

Íme a modellt benépesítő tagfüggvény:

```
void FileOperator::populateModelFromXml(QDomDocument xmlDocument)
{
    QDomNode rootElement = xmlDocument.documentElement();
    QDomNode node = rootElement.firstChild();
    while(!node.isNull()){
        QDomElement element = node.toElement();
        if(!element.isNull()){
            QStandardItem *item = new QStandardItem(element.
attribute("job"));
            model->setItem(model->rowCount(), 0, item);
            item = new QStandardItem(element.attribute("category"));
            model->setItem(model->rowCount()-1, 1, item);
        }
        node = node.nextSibling();
    }
}
```

A függvény első sorában beolvassuk a gyökérelemet. A második sorban beolvassuk a gyökérelem első csomópontját (angolul: node), s a negyedik sorban elkezdjük bejárni a többi. Figyeljünk meg, hogy a bejárást végző ciklus bennmaradási feltétele, illetve a ciklus léptetését végző `node.nextSibling()` utasítás együtt nagyjából így fordítható: „amíg van még csomópont ezen a szinten”. Lévéen a mi XML-ünk eléggé lapos struktúrájú, nem kell bajlódnunk más szintek ellenőrzésével.

A ciklus első sorából nagyjából kiderül, hogy az elem és a csomópont valahogy úgy van egymással, mint a bogár meg a rovar: nem minden csomópont elem, de minden elem

csomópont. Így aztán konvertálnunk kell, és a konvertálást végző lépés előtt nagyvonalúan nem nézzük meg, hogy ez a csomópont tényleg elem-e – egyébiránt a `QDomNode::isElement()` tagfüggvénnyel ejthetnénk meg ezt az ellenőrzést.

Az elem jellemzőiből `QStandardItem` osztályú objektumokat képzünk, amiket aztán végre tud kezelni a modellünk, el is helyezzük benne.

Ezt volt hát az XML-formátumot kezelő `ToDoList` meséje. Ahogy az a jó mesékhez illik, ennek is van tanulsága – bár a tanulságot pár fejezettel ezelőtt is levontuk –, nevezetesen: éles projektben ellenőriznünk kell, hogy a bemeneti fájl tényleg nekünk való-e, különben a programunk esetleg nem pont azt teszi majd, amit várunk tőle.

16. A ToDoList, MINT SQL-KLIENS

Ha már az előző fejezetben a feladatokat XML-fájlban tároltuk, lépjünk még egy nagyot: tároljuk az adatainkat SQL-ben. Ezúttal a fájl tárolás megváltozása nem egyszerű felszíni változás, hanem az egész alkalmazásunkat érinti.

A mi adatbázisunk egyetlen táblát, a `tasks` nevűt tartalmazza, amelyben nem meglepő módon két mező – „`job`” és „`category`” – bujkál, egyelőre még az elsődleges kulcsról is lemondunk. Az adatbázis adatait egy `QSqlTableModel` osztályú objektumban helyezzük el, amit természetesen ismét olyan nézethez kapcsolunk majd, amely képes az adatok változásának megfelelően frissíteni magát, illetve képes arra, hogy a megváltozott adatokat visszahelyezze a modellbe.

Azt ígértük, hogy ezúttal nem felszíni, hanem az alkalmazásunk mélységeit érintő lesz a változás. Ha ezt ígértük, így is lesz, de mi ennek az oka? Az, hogy a `QSqlTableModel` osztályú objektumok nagyon szorosan együtt élnek az adatbázis-kapcsolatukkal – a modell adatváltozásai azonnal megjelennek magában az adatbázisban is. Nem lesz hát külön osztályunk, amely a mentést és a megnyitást végzi, és maga a mentés nem különül majd el időben az adatmodell megváltozásától – a felhasználó szempontjából egy művelet lesz egy új feladat felvétele és a változás tárolása, egy meglévő feladat módosítása és a változás tárolása, egy szükségtelenné vált feladat törlése és a változás tárolása. Sőt, nem csak a felhasználó szempontjából lesz egy művelet, hanem a Qt-ban fejlesztő fejlesztő számára is.

Azért is változik az alkalmazásunk, mert az adatbázisok tábláiban a legkritikább esetben van jelentősége a rekordok sorrendjének. Így értelmét veszítik a ToDoList azon funkciói, amelyek a sorok beszúrásával, fölfelé-lefelé mozgatásával kapcsolatosak. (Ha nagyon kell, természetesen megvalósítható az, hogy ezek a gombok visszanyerjék létjogosultságukat, de ez csak egy strandkönyv.)

A Qt adatbázis-szerverek egész sorához képes kapcsolódni – és természetesen írhatunk saját csatolót a saját fejlesztésű SQL-szerverünkhöz, ha nem elegendő a választék. Mi ebben a fejezetben az elterjedt SQL-adatbázisok sok tekintetben legkezdetlegesebb, mégis nagyon sok helyütt használt megvalósításával, az SQLite-tal ügyködünk.

Mi hát az, ami az SQLite-ot olyan vonzóvá teszi sok jelentős alkalmazás számára is? Az, hogy nem kell hozzá SQL-szerver, és szerver alatt ebben a szövegösszefüggésben számítógépet és szoftvert egyaránt érthetünk. Az SQLite az SQL olyan megvalósítása, amely lényegében nem más, mint egy fájl a háttértáron – egy SQLite fájl ilyen értelemben igen hasonlít a Microsoft Access, illetve a LibreOffice Base saját adatfájljaira.

16.1. Gyomlálunk

Térjünk vissza azokhoz a régi szép időkhez, amikor még épp csak belekezdünk a modelles ToDoList-változat kialakításába. A könyv webhelyén is megtaláljuk azt az állapotot, amikor már ki van alakítva a felhasználói felület, de még csak a névjegy működik. Ebből az állapotból indulunk ki, és még törölünk is belőle.

Nem fog kelleni a `buttonNewBefore`, a `buttonUpSelected` és a `buttonDownSelected` gomb, a `buttonNewAfter` gombot meg kereszteljük át `buttonNewTask`-ra és a feliratát is ilyen értelemben változtassuk meg. Az `Action Editor`-ban hajigáljuk ki a mentés, mentés másként

és a beállítások QAction-jét, illetve az ez utóbbi QAction-höz tartozó slot-ot is pusztítsuk. Említettük már, hogy a mentés lényegében értelmét veszti. A mentés másként nem volna több egyszerű átnevezésnél. A beállítások ablakban meg azt az automatikus mentést tudtuk szabályozni, ami úgymint megvalósul.

Utolsó mozzanatként a szükségtelenné vált ikonokat is eltávolítjuk az erőforrásfájlból, illetve a mentés ikonra való hivatkozást az `about.html`-ből. Ekkora tarolás után készen állunk az új feladatra.

16.2. Első QSqlTableModel-ünk

Az első fül nézetével ezúttal egy QSqlTableModel osztályú modellt fogunk összekapcsolni. Ha megnézzük az osztály dokumentációját, azt látjuk hogy a konstruktorának szinte kötelező paramétere az adatbáziskapcsolat. A „szinte” kötelező azt jelenti, hogy ha nem adnánk meg paraméterül egy kapcsolatot, akkor az alapértelmezettet fogja használni a tábla. S bár még nem beszéltünk egy szót sem erről az adatbáziskapcsolat-dologról, azt értenünk kell, hogy adatbázis nélkül nincs modell. Azaz az új modell kialakítását megelőzi az adatbázishoz való kapcsolódás, és ezen az sem változtat, hogy a mi esetünkben az adatbázis egy fájl.

Az osztály dokumentációjában semmi nem utal arra, hogy meg tudnánk változtatni a modell alatt lévő táblát. Ha pedig ez nem lehetséges, akkor szembesülnünk kell azzal hogy az eddigiekkel ellentétben nem egy modellt fogunk ürítgetni, például mielőtt új fájlt nyitnánk meg, vagy amikor üres feladatlistát kezdenénk, hanem kidobjuk a régi modellt és újat fogunk használni. A nézet a főablakban lakik, a modellkezelőt meg önálló osztályba tesszük, tehát amikor új modellünk van, szólnunk kell róla a nézetnek is. Erre kell majd egy signal, meg az ő slot-ja.

A fentiekből az is következik, hogy nem járható út az, hogy elindítjuk az alkalmazást, és majd egyszer mentjük a táblánkat²⁹, hanem az új feladatlista az új fájl nevének megadásával, illetve a fájl mentésével kezdődik. Sőt, az a helyzet, hogy ez a mentés valójában nem is mentés, hanem először létrehozunk egy SQLite-fájlt, aztán abban a táblát, és megnyitjuk a fájlt, benne a táblát.

Mindebből pedig az szűrhető le, hogy a létező fájl megnyitása és az új fájl kialakítása sok szempontból azonos dolog, csak az új fájl megnyitása bonyolultabb. Megjegyezzük még, hogy e hosszas gondolkodás azért volt szükséges, hogy hamarabb megértsük, miért pont úgy írjuk a programunkat, ahogy írjuk. És akkor most végre írjuk!

A projektfájlban helyezzük el az sql bejegyzést:

```
QT += core gui sql
```

A projektben nyissunk egy TableManager osztályt, mégpedig olyat, amelyiknek a QWidget osztály az őse. Megint pusztán azért pont az, mert akarunk belőle QDialog-okat nyitogatni, és a QDialog-ok ősoosztályának QWidget-leszármazottnak kell lennie. A főablak konstruktorában máris példányosíthatunk magunknak belőle, de előtte még rójuk le a kötelező köröket a QSettings osztály könnyebb használata érdekében:

²⁹ Ha nagyon akarjuk, éppen megoldható. Az SQLite adatbázisok lakhatnak egy speciális fájlban, a `:memory:`-ban, és át lehet őket költöztetni őket igazi fájlba, de a megoldás nem nyilvánvaló, és inkább SQLite-ismereteket igényel, semmint általánosan használható Qt-SQL ismereteket. Így aztán ez (is) kimarad strandkönyvünkől.

```

QCoreApplication::setOrganizationName("Sufni & Co.");
QCoreApplication::setOrganizationDomain("example.com");
QCoreApplication::setApplicationName("ToDoList");
tableManager = new TableManager(this);

```

A `tablemanager.h` fájlban deklaráljunk publikus mutatót az modellünknek:

```

 QSqlTableModel *tmodel;

```

Az osztály konstruktorában adjunk értéket az új mutatóknak, de még nem igazi objektumot:

```

tmodel = NULL;

```

Ugyanis, mint említettük, több esetben lesz majd új modellre szükségünk, ami ráadásul új fájl előkészítésével is jár. Így a munkát érdemes volna külön tagfüggvényre bízni. A tagfüggvény meg ne borítson be bennünket `QSqlTableModel` osztályú objektumokkal: érdemes előtte ellenőrizni, hogy van-e már régi, és ha igen, azt töröljük, mielőtt újat példányosítanánk. A régi objektum meglétéről vagy nemlétéről úgy tudunk legegyszerűbben megbizonyosodni, hogy megnézzük a mutatójának értékét. Ha inicializálatlan mutató értékét nézzük meg, abból nagy elszállások lesznek, s ezért a mutató értékét az első lehetséges alkalommal inicializáljuk: `NULL` értékre. Ezek után tudni fogjuk, hogy amennyiben a mutató `NULL`, akkor nincs még modellünk, ha meg valami más, akkor van.

Munkálkodásunkat a konstruktorban azzal folytatjuk, hogy szétnézzünk, volt-e már fájlunk valaha, azaz a tárolt beállítások között.

```

QSettings settings;
QString fileName = settings.value("Files/LastSQLFileName", "").
toString();
if(fileName.isEmpty())
    newModel();

```

Egyelőre nem foglalkozunk azzal az esettel, ha már volt. Lássunk neki a `newModel()` függvény megírásának. Sőt, ne is modell legyen, hanem slot, és akkor máris köthetjük kevés megmaradt főablakbéli `QAction`-ünk közül az `actionNew()` nevezetűhöz.

Írjuk meg a `newModel()` slot elejét:

```

if(tmodel != NULL){
    delete tmodel;
}
QString fileName = QFileDialog::getSaveFileName(this,
                                                tr("New Database"),
                                                QDir::homePath(),
                                                tr("SQLite Database Files (*.sqlite)"));
QFile file(fileName);
if(file.exists())
    file.remove();

```

Azaz ha már van modell, azt kukázzuk, kérdezzük meg a felhasználót hogy mi legyen az új fájlnev, ha már van ilyen fájl, azt kukázzuk... és?

Most jön az a rész, hogy az új fájl felhasználásával létrehozzuk az új modellt. Ne feledjük azonban, hogy a modellnek feltétlen kellene SQLite-fájlhoz kapcsolódnia, hanem nyugodtan használhatnánk valami nagyagyú adatbázisszerveret is. Azaz a modell adatbázis-kapcsolatot

vár paraméterül. Adatbázis-kapcsolatot meg már meglévő fájl megnyitása esetén is ki kell építenünk, azaz most, hogy félig sem vagyunk kész a slot-unkkal, megírjuk a privát `openDb()` tagfüggvényt

```

QSqlDatabase TableManager::openDb(const QString fileName)
{
    if(db.isOpen())
        db.close();
    else
        db = QSqlDatabase::addDatabase("QSQLITE");
    db.setDatabaseName(fileName);
    db.open();
    QSettings settings;
    settings.setValue("Files/LastSQLFileName", fileName);
    return db;
}

```

A függvény kezdetét olvasgatva feltűnhet, hogy ilyen változónk még nincs is – a fejlécfájl privát változói között helyezzük el a

```
QSqlDatabase db;
```

sort, és ne feledjük az `#include <QSqlDatabase>` sort a fájl elejéről. Szólunk és hangsúlyozzuk: a `QSqlDatabase` osztályú objektumok *nem* adatbázist, hanem adatbázis-kapcsolatot jelentenek. Ennek tudatában értelmet nyer, ahogy a fenti tagfüggvény kezdődik: ha a kapcsolat nyitva volna, csukjuk be. Ha nincs nyitva kapcsolat, az a mi programunkban csak azért lehet, mert még csak most fogjuk kiépíteni az elsőt, azaz még adatbáziskapcsolat-objektumunk sincs. Ilyenkor példányosítunk egyet magunknak, és a program során mostantól mindig ezt fogjuk használni. Az adatbázis-kapcsolat SQLite adatbázishoz fog kiépülni, azaz ezt a csatolót illesztőprogramot, magyarul *driver*-t adjuk meg.

A következő cselekedetünk a fájlnev megadása. A `setDatabaseName()` tagfüggvény paraméterül egész csúnya dolgokat is elfogad, főleg, ha komolyabb, többféle adatbázist is kezelő csatolóval példányosítottuk a kapcsolat-objektumot. Ha az adatbázis nevét is tudja már a kapcsolat, elkézelhető, hogy felhasználónevet, jelszót, miegymást is be kell állítanunk, de nem a mi esetünkben, mi vigyorogva folytatjuk a megnyitást. Az `open()` tagfüggvény visszatérési értéke logikai, azaz kiderül, hogy sikerült-e megnyitnunk az adatbázist. Mi e pillanatban annyira győztesnek érezzük magunkat, hogy meg sem vizsgáljuk a visszatérési értéket – ebből is látszik, hogy nem ipari projekten dolgozunk.

Még gyorsan eltároljuk a fájlnevet a sikeres(?) megnyitás öröme, majd visszaadjuk a hívónak az adatbáziskapcsolat-objektumot.

Visszaoldaloghatunk a `newModel()` slot-ba, és most hogy van ilyen jó kis adatbázisfájl-nyitogató tagfüggvényünk, megírhatjuk a modellt példányosító sort:

```
tmodel = new QSqlTableModel(this, openDb(fileName));
```

Persze ez a fájl még üres, nincs benne egy incifinci picike kis tábla sem. A slot következő két sorában ezen segítünk, és közben megismerünk egy új osztályt (ne feledjük elhelyezni a fejlécét):

```
QSqlQuery query(db);
query.exec("CREATE TABLE tasks (job TEXT, category TEXT);");
```

A QSqlQuery osztály nem egyszerű darab, többek között azért sem, mert nem csak CREATE, hanem SELECT utasítás is megadható benne, és így az eredmény tárolása is feladata lehet, ráadásul eszközöket biztosít az eredmény bejárására. Tovább bonyolítja a helyzetet, amiről az adatbázis-szakemberek – vagy a lassú hálózati kapcsolattal rendelkezők – tudnának mesélni, hogy nem minden lekérdezés lefutása pillanatszerű, és akkor még nem is beszéltünk az egyéb nyálánkságokról. Mi most épp csak ízelítőt kaptunk a használatából. Határtalan önbizalmunkban megint nem vizsgáljuk, hogy rendben lefutott-e a lekérdezés, de végre megemlítjük a QSqlDatabase::lastError() tagfüggvényt, ami egy kis qDebug()-gal kombinálva sok fejfájástól kímélhet meg bennünket.

Itt az idő, hogy szóljunk a QSqlTableModel-nek, hogy kész az új tábla, tessék használatba venni. Azonban ugyanígy a szájába kell rágnunk a használandó tábla nevét akkor is, amikor már létező fájlt nyitunk majd meg, ráadásul a modellen még számos egyéb beállítanivalónk van, azaz már megint új privát tagfüggvényt írunk:

```
void TableManager::setTableModelProperties()
{
    tmodel->setTable("tasks");
    tmodel->setEditStrategy(QSqlTableModel::OnFieldChange);
    tmodel->setHeaderData(0, Qt::Horizontal, tr("job"));
    tmodel->setHeaderData(1, Qt::Horizontal, tr("category"));
    tmodel->select();
}
```

Az első dolgunk a megfelelő tábla használatba vétele. A második annak megadása, hogy mikor mentse a modell az SQL-táblába az adatokat. A setEditStrategy() tagfüggvény paramétere – a Qt jó szokása szerint – egy enum, amely ezúttal három értéket vehet fel. Amit mi használunk, az akkor menteti a változásokat, ha befejeztük a mező szerkesztését. Figyelem! Ez nem adatbázis-mező, hanem modell-mező, azaz tulajdonképp egy adott érték szerkesztéséről van szó. A két másik lehetőség közül az egyik, hogy akkor mentjük a változásokat, ha kiléptünk a rekordból, a harmadik meg az, hogy majd kézzel mentjük a változásokat, amikor épp nincs jobb dolgunk.

A munkát a fejlécek beállításával folytatjuk. Ha nem adunk meg fejlécet, akkor az adatbázismező nevét használja a modell, ami esetünkben történetesen megegyezik az általunk beállítottal. Akkor meg minek megadni? Azért, mert így lefordíthatóvá válik a két oszlopcím.

Végül a select() tagfüggvény hívása következik, amely szól a modellnek, hogy most már átveheti az adatokat a táblából – eddig üresen állta a modell.

A newModel() slot törzsének végére helyezzük el a fenti tagfüggvény hívását:

```
setTableModelProperties();
```

Az új modell használatra kész, már csak szólni kéne a főablaknak, hogy beállíthassa a nézete modelljéül. Deklaráljuk a fejlécfájlból a newTmodelReady(); signal-t, és itt, a slot végén emittáljuk, közben pedig elmagyarázhatjuk a hörcsögnek, hogy ez a tmodel nem az a T-Model, ezt mi egyedül csináltuk, semmi köze Henry Ford-hoz.

Kullogunk át a főablakba, és írjuk meg az előbbi signal-t fogadó slot-ot.

```
void MainWindow::onNewTmodelReady()
{
    ui->tableView->setModel(tableManager->tmodel);
}
```

A konstruktorban pedig, a `tmodel` objektum példányosítása alá helyezzük el a `connect` utasítást:

```
connect(tableManager, SIGNAL(newTmodelReady()),
        this, SLOT(onNewTmodelReady()));
```

Felmerül persze – persze! – egy hangyányi probléma. A példányosító sor lefutásával létrejön a `tmodel` objektum, annak meg kvázi a konstruktorából hozzuk létre az új modellt. Emittáljuk a modell elkészültét jelző signal-t, és *utána* hozzákötjük a signal-hoz a slot-ot. No bueno. Természetesen *később* jó lesz még ez a kapcsolat, de most épp semmire sem megyünk vele., úgyhogy kézzel hívjuk a slot-ot, rogtón a `connect` utasítás után:

```
onNewTmodelReady();
```

Ha más ügyis kötögetünk, mint a nagy, adjuk még ki az új feladatlistát készítő gombot a `newModel()` slot-hoz kapcsoló `connect` utasítást:

```
connect(ui->actionNew, SIGNAL(triggered()),
        tableManager, SLOT(newModel()));
```

Itt az idő, hogy ennyi rengeteg meló után végre futtassuk az alkalmazásunkat. Ha felvettük az összes kihagyott fejléct, akkor elindul az alkalmazás, és szépen bekéri a fájl nevét, aztán ott a nagy nesze semmi, fogd meg jól. Működik a „New” menüpont is, hasonló eredménnyel – lehet, hogy azért, mert ugyanaz a slot fut le mindkét esetben?

Pusztán a szépség mián helyezzünk el két további sort a főablak konstruktorában – nem is kommentáljuk őket, ismerősek, csak még nem jutottak az eszünkbe:

```
ui->tableView->horizontalHeader()->setSectionResizeMode(QHeaderView::Stretch);
ui->tableView->verticalHeader()->hide();
```

Jól volna talán, ha végre beírhatnánk egy teendőt. Az új modellünkbe azonnal elhelyezhetünk új sort, de a „new task” gombhoz is hozz kellene az új sor hozzáadását. Alighanem slot-ot fogunk írni:

```
void TableManager::appendEmptyRecord()
{
    tmodel->insertRecord(-1, tmodel->record());
}
```

A sor jelentése a következő: az utolsó sor után (negatív indexek ezt jelentik ebben a tagfüggvényben) szúrj be egy olyan rekordot, amelyet egyébiránt a `tmodel`-ben is talál az ember. Azaz nem kellett szütyögnünk az új rekord mezőinek beállításával – de senki ne izguljon, lesz még olyan is, amikor fogunk. Bár elsőre nem olyan feltűnő, de a slot a `QSqlRecord` osztályból példányosítja a beszúrandó rekordot, így helyezzük el ezt a fejléct is a többi között. Ezt a slot-ot kell hívunk, még mielőtt emittálnánk a `newTmodelReady()` signal-t. A főablak konstruktorában pedig megírjuk a megfelelő `connect` utasítást, így innen is hívható a slot:

```
connect(ui->buttonNewTask, SIGNAL(clicked()),
        tableManager, SLOT(appendEmptyRecord()));
```

Talán még emlékszünk, hogy szívesen állítottuk kijelöltnek az első cellát, ezzel is hívogatva a felhasználót, hogy írjon már végre egy feladatot. A kijelöléseket a `QItemSelectionModel` osztályból példányosított objektum tartalmazza, és ha véletlen lustán arra gondolnánk, hogy túl nagy macera ez, akkor emlékeztetnénk arra, hogy a törlés gomb slot-jának valahonnan úgy is meg kell tudnia, hogy melyik sor a törlendő. Így aztán deklarálunk publikus mutatót magunknak a `tablemanager.h` fájlban:

```
QItemSelectionModel *selectionModel;
```

A kijelölésmodell konstruktorának kötelező paramétere a forrásmodell, azaz a kijelölésmodellt együtt töröljük és együtt példányosítjuk a `tmodel` objektummal. Így alakult, hogy a `newModel()` slot elején a `delete tmodel` sor alá egy

```
delete selectionModel;
```

sor is kerül, és a `tmodel` objektumot példányosító sort az alábbi követi majd:

```
selectionModel = new QItemSelectionModel(tmodel, this);
```

Az `appendEmptyRecord()` slot-ba írjuk bele azt a két sort, amely az utolsó sor első oszlopár állítja a kijelölést:

```
selectionModel->clearSelection();
selectionModel->setCurrentIndex(tmodel->index(tmodel->rowCount()-1,0),
                                QItemSelectionModel::SelectCurrent);
```

Végül a főablak `onNewTmodelReady()` slot-ját egészítjük ki egy sorral:

```
ui->tableView->setSelectionModel(tableManager->selectionModel);
```

Mielőtt kipróbálnánk a fejlesztés eredményét, a tárolt beállítások közül töröljük az utolsó SQL-fájltra vonatkozót, ugyanis csak akkor lesz érvényes kijelölésmodellünk, ha új fájlt kezdtünk, fájlt megnyitni egyelőre nem tudunk. Az `onNewTmodelReady()` slot viszont mindenképp beállítja a `selectionModel` mutatót mint kijelölésmodellt, azt meg már a hörcsög is tudja, hogy az inicializálatlan mutató olyan, mint a romlott répa: aki babrál vele, rosszul jár.

Ha kipróbáltuk a programunkat, és elégedettek vagyunk vele, tegyünk egy kicsit azért, hogy ne kelljen már kitörölgetni a tárolt fájlnévet: valósítsuk meg az automatikus betöltést, privát tagfüggvényként. A gondos tervezésnek köszönhetően meglepően kevés dolgunk lesz vele, nem utolsósorban azért, mert, ahogy említettük a fejezet elején, az új fájl és a létező fájl megnyitása a mostani esetben nagyon hasonló lépésekből áll. Lényegében egy lebutított `newModel()` slot-ot kell írunk:

```
void TableManager::modelFromExistingFile(const QString fileName)
{
    if(tmodel){
        delete tmodel;
        delete selectionModel;
    }
    tmodel = new QSqlTableModel(this, openDb(fileName));
    selectionModel = new QItemSelectionModel(tmodel, this);
    setTableModelProperties();
    emit newTmodelReady();
}
```

Használatba venni úgy tudjuk, hogy a TableManager osztály konstruktorában lévő elágazást kiegészítjük egy else-ággal:

```
else
    modelFromExistingFile(fileName);
```

Oldjuk meg még gyorsan azt is, hogy működjön a megnyitás menüpont is. Lényegében arról van szó, hogy a felhasználótól bekérjük annak a fájlnek a nevét, amit meg szeretne nyitni, majd a fájlnevével felparaméterezve hívjuk a modelFromExistingFile() tagfüggvényt. A feladatot a következő publikus slot valósítja meg:

```
void TableManager::openFile()
{
    QString fileName = QFileDialog::getOpenFileName(this,
                                                    tr("Open Database"),
                                                    QDir::homePath(),
                                                    tr("SQLite Database Files
(*.sqlite)"));
    if(!fileName.isEmpty())
        modelFromExistingFile(fileName);
}
```

A főablak konstruktorában a kiadjuk a connect utasítást:

```
connect(ui->actionOpen, SIGNAL(triggered()),
        tableManager, SLOT(openFile()));
```

Az alapvető funkciók működnek, úgyhogy megfelezzhetünk egy almát a hörcsöggel. Elég kicsik ezek az almák, adjuk neki a nagyobb felét, jó?

16.3. Törlés és visszavonása – barkácsoljunk kézzel QSqlRecord osztályú objektumot!

A törléskor megszüntetett rekord értékei szokás szerint a privát undo objektumba kerülnek. Aki véletlen nem emlékszik rá, íme az objektum deklarációja:

```
QStack<QPair<QString, QString> > undo;
```

Magát a törlést publikus slot-ként valósítjuk meg:

```

void TableManager::onDeleteSelectedTask()
{
    int row = selectionModel->currentIndex().row();
    if(row >= 0){
        undo.push(QPair<QString, QString>(
            tmodel->index(row,0).data().toString(),
            tmodel->index(row,1).data().toString()));
        tmodel->removeRow(row);
        tmodel->submit();
        tmodel->select();
        int rowcount = tmodel->rowCount();
        selectionModel->clearSelection();
        if(rowcount > 0){
            if(row <= rowcount-1){
                selectionModel->setCurrentIndex(
                    tmodel->index(row,0),
                    QItemSelectionModel::SelectCurrent);
            }else{
                selectionModel->setCurrentIndex(
                    tmodel->index(rowcount-1,0),
                    QItemSelectionModel::SelectCurrent);
            }
        }
    }
}

```

A kijelölésmodelltől a szokásos módon megtudjuk az épp aktuális sor számát, amely masszív törlési műveleteket követően lehet akár negatív is – ezért az ellenőrzés. A sor tartalmát elhelyezzük az undo veremben, és utána töröljük a sort. Aztán jön két érdekes sor:

```

tmodel->submit();
tmodel->select();

```

Ha kihagyjuk őket, rögtön meglátjuk mire kellenek – főleg akkor látszik szépen a helyzet, ha ideiglenesen megjegyzéssé alakítjuk a főablaknak azt a sorát, ami a nézet függőleges fejlécét vakarja ki. A QSqlTableModel osztályban a törölt sor csak törlésre jelöltetik – a függőleges fejlécen a száma helyett felkiáltójelet látunk – és csak akkor történik meg a törlés, ha elküldjük a kérést, majd újra betöltjük az immár megváltozott táblát.

A következő sorok már csak azzal bajlódnak, hogy pontosan hova kerüljön a kijelölés a törlést követően.

A szokásos connect utasítás a főablak konstruktorába:

```

connect(ui->buttonDelete, SIGNAL(clicked()),
        tableManager, SLOT(onDeleteSelectedTask()));

```

A törlés megy. A visszavonást végző publikus slot-ban ismét az egyszer top(), másszor pop() módszert használjuk:

```
void TableManager::onUndoLastDelete()
{
    if(!undo.isEmpty()){
        QSqlRecord record;
        record.append(QSqlField("job", QVariant::String));
        record.append(QSqlField("category", QVariant::String));
        record.setValue("job", undo.top().first);
        record.setValue("category", undo.pop().second);
        tmodel->insertRecord(-1, record);
    }
}
```

Az újdonság az, ahogy a beszúrandó rekordot kézzel összeállítjuk. Használhatnánk azt a módszert is, ahogy az `appendEmptyRecord()` slot dolgozik, azaz a modelltől elkért rekordot szűrjük be a modell végére, utóbb pedig beállíthatnánk a rekord értékeit. Azért nem használjuk azt, mert így szép új módszerrel gazdagodunk. Figyeljük meg, hogy az új rekordnak elsőként a mezőit adjuk meg – minden mező `QVariant` típusú, de azért teszünk utalást arra nézvést, hogy mi a mező igazi adattípusa. A mezők megadását követően értéket adunk az egyes mezőknek, majd az utolsó utáni helyre (negatív index) beszúrjuk a rekordot. Adjuk meg a `<QSqlField>` fejléctet.

A szokásos connect-sor:

```
connect(ui->buttonUndo, SIGNAL(clicked()),
        tableManager, SLOT(onUndoLastDelete()));
```

Most jutottunk el arra a pontra, hogy az első fül kész – így aztán a keresőfülrel folytatjuk a munkát.

16.4. Keresés és kiegészítés a `QSqlQueryModel` osztállyal

A `QSqlTableModel` osztály egy adott tábla tartalmának megjelenítésére alkalmas. Bizonyos szempontból lényegesen ügyesebb a `QSqlQueryModel` osztály, ez ugyanis – nevéhez méltóan – egy SQL-lekérdezés eredményét jeleníti meg modellül. A lekérdezés lehet többtáblás, és használható benne a teljes SQL-nyelvi repertoár – már amit a programunkban használt SQL-megvalósítás is ismer.

Persze ott van az a fránya „bizonyos szempontból” kitétel. A `QSqlQueryModel` osztályú objektumok által nyújtott modellek ugyanis csak olvasható modellek.

Mi most mindenesetre szeretnénk magunknak egy ilyet – jó, jó kettőt, de nem akarunk annyira mohónak tűnni, úgyhogy a másikat majd később. Deklarálunk neki publikus mutatót a `TableMaganer` osztályban:

```
QSqlQueryModel *searchModel;
```

Ne feledjük megadni a `<QSqlQueryModel>` fejléctet. A konstruktorban példányosíthatjuk:

```
searchModel = new QSqlQueryModel(this);
```

A főablak konstruktorában adjuk meg, mint a `searchView` (a második fülön lévő nézet) objektum modelljét, s ha már úgyis arra járunk, elhelyezhetjük a szokásos kozmetikázó sort is:

```
ui->searchView->setModel(tableManager->searchModel);
ui->searchView->horizontalHeader()->setSectionResizeMode(QHeaderView::Stretch);
```

És hogy mire kell keresni? Természetesen a második fülön lévő `searchEdit` tartalmára, a `category` mezőben. A `searchEdit` tartalmának változását egy jó kis signal folyton emittálja, már csak annyi a dolgunk, hogy írunk egy publikus slot-ot a `TableManager` osztályba, ami amjd elkapja ezt a signal-t:

```
void TableManager::updateSearchModel(const QString searchString)
{
    searchModel->setQuery("SELECT job, category FROM tasks WHERE
category LIKE \"%\" + searchString + \"%\";", db);
}
```

Figyeljünk a lekérdezésben szereplő idézőjelek levédésére. A slot-hoz való `connect` utasítás a főablak konstruktorába:

```
connect(ui->searchEdit, SIGNAL(textChanged(QString)),
        tableManager, SLOT(updateSearchModel(QString)));
```

Szinte tökéletesen működik. Egyetlen érdekes problémánk adódhat még: ha kerestünk valamire, majd felveszünk egy olyan újabb feladatot a feladatlistába, amelyiknek a kategóriájára illeszkedik a keresési feltétel, akkor visszatérve a keresőfülre, a keresés még nem frissül. A nehézség leküzdhető, ha a `tabWidget` fülszám-változását jelző signal-hoz írunk egy új slot-ot. Az új slot a `QSqlQueryModel::query` tagfüggvény használatával megtudja a jelenlegi lekérdezést, majd a már használt `setQuery` tagfüggvénnyel újra lefuttatja.

Még egyetlen dolgot szeretnénk megvalósítani a ToDoList SQL-es változatában, azt, hogy működjön az automatikus kiegészítés. A kiegészítést végző `QCompleter` osztályú objektumot az első fül táblázatában nem fogjuk használni, legalábbis itt, a könyvben - nem fogunk ugyanis delegáltat írni. Semmi nem akadályoz meg azonban bennünket abban, hogy a keresőfül `searchEdit` objektumánál be ne állítsuk a kiegészítő használatát.

Hozzunk létre hát a `TableManager` osztályban egy publikus mutatót a kiegészítést végző objektumnak, meg egy privátot, az előző objektum modelljének:

```
QCompleter *completer;
```

illetve

```
QSqlQueryModel *completerModel;
```

Az osztály konstruktorában mindkettőhöz példányosítunk objektumot:

```
completerModel = new QSqlQueryModel(this);
completer = new QCompleter(completerModel, this);
```

És megírjuk a `completerModel` objektum frissítését végző slot-ot:

```
void TableManager::updateCompleterModel(const int tabIndex)
{
    if(tabIndex == 1)
        completerModel->setQuery("SELECT DISTINCT category FROM
tasks", db);
}
```

A slot-ot a főablakban a `tabWidget` fülszámváltozását jelző signal-hoz kötjük, a `completer` objektumot pedig beállítjuk a `searchEdit` objektum segítőjeként:

```
connect(ui->tabWidget, SIGNAL(currentChanged(int)),
        tableManager, SLOT(updateCompleterModel(int)));
ui->searchEdit->setCompleter(tableManager->completer);
```

A ToDoList SQL-es változata ezzel elkészült. Klassz, igaz?

17. EGYSZERŰ QT-PROGRAM CSOMAGOLÁSA LINUX-RENDSZEREKHEZ

Ebben a fejezetben telepítőt készítünk kedvenc projektünkhöz, de kizárólag Linux-rendszereken. Windows-ra ugyanis annyiféle módon tudunk telepítőt készíteni, hogy elmondani is nehéz – ez cserébe azt is jelenti, hogy a sok-sok egységesítési törekvés után sincs egy üdvöztető megoldás.

Ezzel szemben a Linux rendszereken jellemzően létezik az egy üdvöztető módszer. Bár néha markáns eltérésekkel, de alapvetően azt az elképzelést szokás megvalósítani, hogy az alkalmazások *csomagokba* vannak szervezve, amelyek tudják magukról, hogy mely egyéb csomagoktól függenek. Ha például egy alkalmazás tartalmaz alapvető fájlokat, de kiegészítéseket is, akkor azokat sok esetben külön fájlokba csomagolják, és a kiegészítések függenek az alapsomagtól. Például amennyiben egy napon megírjuk kedvenc bicikliszimulátorunkat, akkor az alapsomag esetleg csak a játék motorját és egy gyakorlópályát tartalmazza, a többi pályát külön-külön csomagoljuk majd. Természetesen minden pálya függ az alapsomagtól.

De mit jelent az, hogy „függ”? Nos, a gyakorlatban azt, hogy a csomag a telepítésekor megkérdezi az épp futó rendszert, hogy telepítve vannak-e azok a fájlok, amelyekről ő maga függ? Ha igen, akkor ő is telepedik, ha nem, akkor a telepítés leáll, és megtudjuk, hogy miért nem hajlandó a rendszerünkre az adott csomag felmászni: kiírja, hogy kéri még ezt-és-ezt telepíteni, aztán próbáljuk újra. Szokott még egy ennél erősebb telepítési módszer is lenni, amelyik, mikor az előző példában felhozott játék valamelyik pályáját telepítenénk úgy, hogy maga a játék még nincs telepítve, udvariasan érdeklődni fog, hogy akkor most telepíteni akarjuk a játékot is, vagy hagyjuk a francba az egészet.

Miért időztünk el ennyit a függésénél? Azért, mert a mi alkalmazásunk is függeni fog a Qt programkönyvtáraitól, a bennük tárolt objektumoktól. Azaz, telepítéskor meg fogja majd kérdezni a gépünk, hogy „Ha már telepítenéd ezt az alkalmazást, jöhetnek a Qt megfelelő részei is”?

A másik megoldás az volna, hogy a használt Qt-részeket statikusan linkeljük az alkalmazásba. Ez azonban részint gazdaságtalan, részint licenclési problémákat vethet fel.

Mielőtt nekikezdenénk a két legelterjedtebb Linuxos csomagformátum (**deb** és **rpm**) valamelyikével foglalkozni, még pár gondolat a Qt-projektek fordításáról. A fordítás alapértelmezés szerint úgy történik, hogy a **qmake** segédprogram előállít egy *makefile*-t, aminek alapján a gépünkön használt **make** program fel tudja úgy paraméterezni a fordítót, hogy a végén azt kapjuk, amit szeretnénk. A **qmake** pedig alapvetően a projektfájlból (projektnév.pro) álmodja meg, hogy mit írjon a *makefile*-ba.

A példánkban egy rém egyszerű alkalmazást fogunk lefordítani, olyat, amelyik nem egyéb, mint egy főablak. Ha Qt Creatorral végeztük a fejlesztést, a következő fájlok találhatók majd a projekt könyvtárában:

```
akarmi.pro
akarmi.pro.user
main.cpp
mainwindow.cpp
```

```
mainwindow.h  
mainwindow.ui
```

Ezek közül az **akarmi.pro.user** a szempontunkból teljesen fölösleges, a továbbiakban nem is használjuk – törölhetjük is, ha még kelleni fog, a Qt Creator majd csinál magának. Az **akarmi.pro** fájlba viszont helyezzünk el két újabb sort:

```
target.path = /usr/bin  
INSTALLS += target
```

Ezt követően futtassuk le a mappánkban a **qmake** parancsot. A futás eredménye jó esetben egy új, **Makefile** nevű fájl a mappában.

17.1. Telepítőcsomag készítése Debian alapú rendszerekre

Az **akarmi** mappát lássuk el verziószámmal is, nézzen ki így: **akarmi-0.2**. Eztán telepítsünk mindent, amire szükség lehet a fordításhoz-csomagkészítéshez:

```
$ sudo apt-get install build-essential dpkg-dev
```

Lépjünk be a mappába:

```
$ cd akarmi-0.2
```

Debianizáljuk a mappát:

```
$ dh_make -s -c gpl -e en@email.nincsilien -createorig
```

Ezzel elkészül egy **debian** mappa az **akarmi-0.2** alatt, benne mindenféle megértenivaló. A README kezdetű fájlokban benne avn, hogy mit kéne írni beléjük, törölhetjük őket. Sem az **ex**, sem az **EX** végű fájlokra nincs szükség, és kivághatjuk a **docs** fájlt is. A **control** fájlba kerül néhány, a felhasználók számára szóló információ, de van ár fontosabb dolog: ha a csomagunk függ egytől-mástól, az is ide kerül. Beszéltünk már arról, hogy a Qt programkönyvtáraitól most épp függeni fogunk, de a beírást megússzuk, mert automatikusan ki fog tölteni ez a rész is, ha a használt Qt-unk hivatalos csomagokból származik. Ha a qt-project.org honlapról töltöttük le a Qt-ot, akkor – lévén az egy disztribúciófüggetlen változat – erre nem sok esély van. A **copyright** fájl nagyja a parancssori paraméter miatt megvan, kipótolhatjuk benne, amit szükséges.

Utolsóként kiadhatjuk a

```
$ dpkg-buildpackage
```

parancsot. Kisvártatva panaszkodni fog, hogy nem volt kulcs, amivel aláírhatta volna a fájlt, de attól még megcsinálta – az **akarmi-0.2** mappával egy szinten. Használható, kedves egészségünkre.

18. KIRÁNDULÁS A QT QUICK-SZIGETEKRE (MEG A QML TENGEREÉBE)

Ennyi kemény munka után itt az ideje, hogy bepakoljuk a fürdőrucit és a hörcsögöt, és elutazzunk. Szóba se jöhet Skandinávia, pedig (vagy épp azért, mert) a Qt-nak sok köze van hozzá. Amikor a repülők a sarkon balra fordul, és bemondják, hogy „a Qt Quick-szigetek fölött az ég kissé zavaros, a hőmérséklet forró, néhol ködszítalással”, sejtjük, hogy nem lesz egyszerű ez a nap sem, de se baj, mert itt van kedvenc strandkönyvünk, majd segít.

A Qt Quick, hasonlóan a Qt-hoz, egy keretrendszer, kicsit ugyanaz, de inkább mégis teljesen más. A magyarázatot a kijelentés végén kezdjük, az eleje meg talán kiderül a fejezet végére.

A Qt egy régi jól bevált nyelvet, a C + ++-t terjeszti ki, tupírozza fel. A Qt Quick meg teljesen új nyelvet kapott, a QML³⁰-t, ami annyira hasonlít a C + ++-ra, mint a szilvalekvár a rósejbnire: mind a kettő kaja, de aki együtt tudja enni őket, annak erős a gyomra. A hagyományos, widget-ekre alapuló Qt többé-kevésbé „kész”, ráadásul ma már nem trendi a szó hagyományos értelmében vett számítógépekre programot fejleszteni, programozási környezetet még kevésbé. Így a Qt háza táján azt a Qt Quick-et fejlesztik erősen, amely jelen állapotában főként felhasználói felületek írására jó, és a jósnőkön kívül senki nem tudja biztosan, hogy ez mindig így lesz-e. A Qt Quick fejlesztése e könyv írásával párhuzamosan is gyors nyelvi változásokkal jár. A dokumentáció szokás szerint a fejlesztés nyomában kullog, és ez még akkor is igaz, ha olyan, meglehetősen korrektül dokumentált projektről van szó, mint a Qt.

Az internet teli van elavult példákkal (amelyek még a Qt 4 és a Qt Quick 1 korából származnak), és a gyors fejlődést mi sem illusztrálja jobban annál, hogy ebben a fejezetben is lesz olyan példaprogram, amihez nem elég a Qt 5.0 sem, hanem Qt 5.1 kell. Aki el akar merülni a témában, figyeljen oda, hogy Qt Quick 2.x-re épülő példákat nézegessen. A két verzió között nem csak nyelvi, hanem alapvető, belső motorbéli különbségek is vannak: az 1.0 még a QPainter-rel rajzol, a 2.x pedig OpenGL-re épül.

A QML alkalmas a mobileszközök elterjedésével követelménnyé vált izgó-mozgó, zizegő-zsebzegő felhasználói felületek elkészítésére. A nyelv deklaratív, azaz csak annyit kell benne megmondani, hogy mi legyen, a hogyan egész ritkán érdekes. Amit nem tud a QML, azt tudja a JavaScript – igen, már itt is JavaScript –, amit meg egyik sem tud, azt tudja a C + ++, meg a jó öreg Qt – a komolyabb háttérfeladatok elvégzésére a QML nyelvű programok is őket használják. Vannak, akik a widget-ekre épülő Qt-ot lassan temetni kezdik, és azok szerint, akik meg akarják vigasztalni a gyászolókat, még néhány évig biztos lesznek widget-ek. Nyilván a piac dönt majd: amíg használják a widget-eket, addig lesznek.

18.1. QML-programot írunk

Ebben a részben bemutatjuk, miből épül fel egy tiszta QML-program. Szokás szerint nem törekszünk teljességre, részint, mert nincs rá hely – külön könyvet érne a dolog –, részint, mert olyan gyors a fejlődés, hogy erre a legerősebb törekvés mellett sem volna esély, részint

30 Az XML, az SQL és a QML között sok-sok különbség van, és alighanem a legkevésbé fontos, hogy a QML betűszó jelentése nem tisztázott. A különféle forrásokban találkozni lehet például „Qt Markup Language”, „Qt Quick Markup Language”, „Qt Modelling Language” és „Qt Meta-language” variációval. Mindenki kiválaszthatja a neki leginkább tetszőt.

pedig azért, mert a kombinációk száma végtelen. A komoly projektek esetében eddig is külön ember vagy részleg foglalkozott a felhasználói felület tervezésével, de a Qt Quick eljövételével ismét várható egy rétegszakma kialakulása. Az új szakma képviselőinek a feladata sok tekintetben a webdesignerek feladatához lesz hasonló: *a nyelv képességeinek pontos ismeretében* képesek lesznek különlegesebbnél különlegesebb hatások elérésére – olyanokra, amelyekre a nyelv fejlesztői soha nem gondoltak volna. A Qt Quick igazán jó használata a hagyományos értelemben vett programozói szemlélettől némileg eltérő logikát kíván.

QML-programot Qt Creatorban is írhatunk – ha így akarunk tenni, akkor most a projektnyitáskor felajánlott sok Quick-es lehetőség közül a Qt Quick 2 UI lehetőséget érdemes választanunk. Két fájlt kapunk, de a projektfájl csak arra való, hogy együtt kezelhessük az esetleges képeket és JavaScript-fájlokat. Nekünk egyelőre ilyesmink nem lesz, lényegében az az egy `qml`-fájl lesz a programunk, amit megnyit a Qt Creator.

Akinek úgy tartja kedve, dolgozhat egyszerű szövegszerkesztővel is. A program értelmezését-futtatását fejlesztés közben legegyszerűbben a `qmlscene` nevű alkalmazással végeztetheti a fejlesztő – az alkalmazás parancssori paramétere a fájl, amit futtatunk. A `qmlscene` az alapértelmezett Qt-telepítések alkalmával a Qt Creatorral együtt kerül a gépünkre, azaz nem kell külön le vadásznunk valahonnan.

A QML-fájlok mindig importálással kezdődnek:

```
import QtQuick 2.0
```

A régebbiekénél a verziószám 1.0 volt, a könyv írásakor a legújabb esetekben 2.1 is előfordulhat.

A Qt Creator által felkínált sablon esetében a program többi része ilyen:

```
Rectangle {  
    width: 360  
    height: 360  
    Text {  
        anchors.centerIn: parent  
        text: "Hello World"  
    }  
    MouseArea {  
        anchors.fill: parent  
        onClicked: {  
            Qt.quit();  
        }  
    }  
}
```

Nézzük meg lépésről lépésre. A program fő eleme egy négyszög (rectangle), amelynek két jellemzőjét adjuk meg: a szélességét és a magasságát. A programban a fő elem két gyermekelemét is deklaráljuk.

Az egyik egy szöveg, aminek megint csak két jellemzőjét deklaráljuk: az egyik maga a szöveg, a másik pedig az, hogy hova horgonyzandó (anchor) a szöveg. Esetünkben a szülő (parent, a négyszögről van szó) közepére. A horgonyzás azért remek eljárás, mert az objektumunk a szülő átméretezésekor is középen marad, ki sem lehet robbantani onnan (na jó, ki lehet).

A második gyermekelem nem látható típusú. Egérterületet deklarálunk vele, azaz olyan részt, ahol a program érzékeny az egérgattintgatásra. Az egérterület horgonyzását úgy végezzük,

hogy a szülő teljes területét kitöltse, azaz bárhova kattinthatunk a négyszögünkben. Ennek a gyermekelemnek signal-jai is vannak (na végre, csak kiderül, hogy Qt lapul a háttérben), a dokumentáció³¹ szerint az egyik a `clicked()`. A slot-ot meg már látjuk is: az `onClicked()` nevet viseli. Ha kattintunk, a program kilép.

Éles szemünkkel rögtön észrevesszük, hogy a pontosvesszőket kukázták a nyelvből, de nem merészkedtek a Python egyszerűségéig³²: a kódblokkok határát itt kapcsos zárójelek jelzik.

Az elemeknek nem adtunk nevet, ami azzal jár, hogy amit egyszer odatettünk, az ott is marad, mégpedig úgy, ahogy letettük. Ha némi dinamizmust szeretnénk csempészni a programunkba, akkor az egyes elemeket azonosítóval (`id`) kell ellátnunk. Tekintsük a következő programot:

```
import QtQuick 2.0
import QtQuick.Controls 1.0

Rectangle {
    id: rectangle
    width: 360
    height: 360
    Button {
        anchors.centerIn: parent
        text: "push me"
        onClicked: {
            rectangle.color = "brown"
        }
    }
}
```

Ezúttal nem egy szöveg dekkol a négyszög közepén, hanem egy gomb (`Button`). Gombunk vagy akkor lehet, ha leesett a nadrágról, vagy importálnunk kell a vezérlőket (`Controls`) is. (Emlékszünk még, amikor jó pár fejezettel ezelőtt arról elméltünk, hogy miként fordítható a `widget` szó? Ott említettük, hogy vannak környezetek, ahol az elemeket vezérlőknek hívják. Ez esetben is kézenfekvőnek tűnik a választás, már csak azért is, hogy ezeket az elemeket véletlenül se keverjük össze a Qt widget-eivel.) Sok vezérlő létezik már a Qt Quick-ben, nézzünk csak körül a dokumentációban.

Ha még nem tűnt fel, most enyhe utalást teszünk arra, hogy az elemek neve nagybetűvel kezdődik, cserébe a jellemzőké kicsivel, nehogy elkopjon a `Shift`. Aki megpróbált nagy kezdőbetűs `id`-t adni a négyszögnek, az meg már azt is tudja, hogy azt sem szabad. Újabb megfigyelni való, hogy nem mindig kettősponttal adunk értéket, mert a fejlesztők nem szeretnék volna megbántani az egyenlőségjelet.

A következő változatban újabb négyszöget alakítunk ki (az `import`-sorokat nem írjuk ide újra):

31 A Qt Creatorban a szokásos módon, a megfelelő szón állva `F1`-et nyomunk. Ha az internetet jobb szeretjük, akkor a könyv írásakor legfrissebb dokumentáció a <http://qt-project.org/doc/qt-5.1/qtquick/qmltypes.html> webhelyen olvasható.

32 A Python nyelvben a sorok behúzása nyelvi elem: a különböző mélységű behúzásokból derül ki, hogy meddig tart a kódblokk. Egymásba ágyazott kódblokkok esetében a mélyebben lévő nagyobb behúzást kap.

```

Rectangle {
    id: rectangle
    width: 360
    height: 360
    Rectangle {
        width: parent.width / 2
        height: parent.height / 3
        color: "yellow"
        anchors.centerIn: parent
        Button {
            anchors.centerIn: parent
            text: "push me"
            onClicked: {
                rectangle.color = "brown"
            }
        }
    }
}

```

Két dolgot nézzünk meg jól. Az egyik, hogy ezek szerint a végtelenségig folytatható az elemek egymásba ágyazása. A másik, hogy a QML él az úgynevezett tulajdonságkötés, magyarul: *property binding* használatával. A belső négyzet szélessége és magassága mindig a külső adataihoz képest lesz értelmezett, azaz az alkalmazásunk vigyorogva átméretezhető.

Az utolsó tiszta QML-példánk következik:

```

Rectangle {
    id: rectangle
    width: 360
    height: 360
    Rectangle {
        id: innerRectangle
        width: parent.width / 2
        height: parent.height / 3
        color: "yellow"
        anchors.centerIn: parent
        Button {
            anchors.centerIn: parent
            text: "push me"
            onClicked: {
                rectangle.color = "brown"
            }
        }
        PropertyAnimation {
            id: myAnimation
            target: innerRectangle
            property: "rotation"
            from: 0
            to: 360
            duration: 5000
            loops: Animation.Infinite
        }
    }
    Component.onCompleted: myAnimation.start()
}

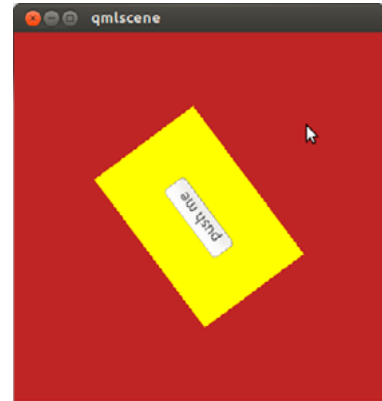
```

A fenti példa a QML rengeteg animációtípusa közül használja az esetleg már a jó öreg Qt-ből is ismert `PropertyAnimation`-t (a Qt-ban `QPropertyAnimation` osztály). Beállítjuk, hogy mettől meddig tartson az animáció, meg, hogy hányszor ismétlődjék, és mehet is a dolog. Kell azonban még egy esemény, meg az ő slot-ja, hogy indulhasson a buli. Mi a komponens – esetünkben az egész QML program – elkészültéhez, kialakításának végéhez kötjük az animáció indítását.

Ahogy ígértük, a következő példában már nem csak QML lesz a `qml`-fájlban. Teszünk hozzá némi JavaScript-et is, egy véletlenszám-előállító függvény képében:

```
Rectangle {
    id: rectangle
    width: 360
    height: 360
    Rectangle {
        id: innerRectangle
        width: parent.width / getRandomInt(2, 4)
        height: parent.height / 3
        color: "yellow"
        anchors.centerIn: parent
        Button {
            anchors.centerIn: parent
            text: "push me"
            onClicked: {
                rectangle.color = "brown"
                getRandomInt()
            }
        }
    }
}

function getRandomInt (min, max) {
    return Math.floor(Math.random() * (max - min + 1)) + min;
}
```



27. ábra: A ferdén álló négyszög valójában forog, csak ugye az állókép...

A JavaScript elkülönülése nem valami látványos: az egyik sorban még QML, aztán meg már nem, aztán meg már megint lehetne, de nem lesz, mert véget ér a fájl.

A következő, és ebben az alfejezetben utolsó példánkban bemutatjuk, hogy miként helyezhető az alkalmazás logikája, a belbecs külön JavaScript `.js`-fájlba, hogy a QML-ben csak a külső maradjon. Ha van kedvünk, a JavaScript-fájlt a Qt Creatorral is elkészíttethetjük, de kézzel is a `qml`-fájl mellé tehetjük. Tegyük bele az iménti függvényt – egyszerű másolás-beillesztés –, majd alakítsuk ilyenné a kódot:

```

import QtQuick 2.0
import QtQuick.Controls 1.0
import "random.js" as RObject

Rectangle {
    id: rectangle
    width: 360
    height: 360
    Rectangle {
        id: innerRectangle
        width: parent.width / RObject.getRandomInt(2, 4)
        height: parent.height / 3
        color: "yellow"
        anchors.centerIn: parent
        Button {
            anchors.centerIn: parent
            text: "push me"
            onClicked: {
                rectangle.color = "brown"
                getRandomInt()
            }
        }
    }
}

```

Szándékosan írtuk ki újra az `import`-sorokat: a JavaScript-fájl az importálás során önálló objektummá avanzsál, aminek ráadásul kötelezően nagy kezdőbetűs a neve. Az új „objektum” tagfüggvényének nevét híváskor a szokásos pont köti az objektum nevéhez.

18.2. Oda-vissza Qt és Qt Quick között: így beszélgetnek egymással a hibrid programok részei

Ebben az alfejezetben egy olyan programot valósítunk meg, amely egyik ablakában egy Qt Quick-ben írt widget csücsül. A csacsogást – természetesen – signal-okkal és slot-okkal végzi a két rész. A sikeres munkához most mindenképp legalább 5.1-es Qt-ra van szükségünk.

Kezdjünk új hagyományos Qt C++ projektet, amelynek némileg megtévesztően a `qml1` nevet adjuk. A főablakban helyezzünk el egy nyomógombot. Adjunk hozzá egy, a `QDialog` osztályból származtatott ablakot is a projekthez (Emlékszünk még? Qt Designer Class a kulcsszó.). Kereszteljük `Dialog` névre, a belőle példányosított ablakot pedig a következő slot nyitja meg:

```

void MainWindow::on_pushButton_clicked()
{
    Dialog dialog;
    dialog.exec();
}

```

Adjunk `qmldialog.qml` néven QML-fájlt a projektünkhöz, és helyezzük el benne az alábbi kódot:

```

import QtQuick 2.1
import QtQuick.Controls 1.0

Rectangle{
    id: main
    width: 100
    height: 100

    CheckBox {
        id: cb
        text: "check me"
        anchors.centerIn: parent
        PropertyAnimation {
            id: cbAnimation
            target: cb
            property: "rotation"
            from: 0
            to: 360
            duration: 5000
            loops: Animation.Infinite
        }
    }

    Component.onCompleted: cbAnimation.start()
}

```

A kód elvileg mostanra könnyen megfejthető: egy `CheckBox`-ot pörgetünk majd benne, amely – természetesen – pörgés közben is pipálható. A `qml`-fájlt helyezzük el egy erőforrás-fájlba, hogy ne kelljen kézzel másolgatni a programmal együtt. Ha mindezzel megvagyunk, térjünk vissza a `Dialog` osztályhoz, és a felhasználói felületet alakítsuk ilyenre:

Az *unchecked* felirat egy `QLabel`, csak jó nagy betűkkel. A nevének sokat gondolkodtunk, majd a hörcsögöt is bevonva a tanácskozásba, alapos megfontolás után a `label` mellett döntöttünk. A `label` objektum alatt egy `QCheckBox` van, amellyel majd gyorsabb pörgést lehet kapcsolni, és amit a hangzatos és a jellemét jól tükröző `checkBox` névvel indítottunk el az élet nagy országútján.

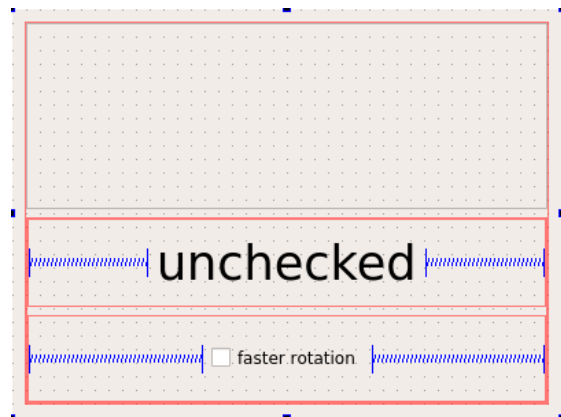
Az ablak felső részén lévő nagy semmi egy `QFrame` osztályú, `frame` nevű keret. Abban a keretben fog majd futni a QML-kódunk.

A `Dialog` osztály kódját szerkesztve a fejlécfájlban a már megismert módon helyezzük el a `duration` nevű tulajdonság definícióját (a `duration` érték az, amely a QML `PropertyAnimation` animáció sebességét szabályozza: ezredmásodpercben adjuk meg, hogy mennyi idő alatt kell lejátszódnia az animációnak):

```

Q_PROPERTY(int duration READ duration WRITE setDuration NOTIFY
durationChanged)

```



28. ábra: A párbeszédablakunk

Deklaráljuk a privát tagváltozót:

```
int m_duration;
```

Deklaráljuk a signal-t:

```
void durationChanged(int animationDuration);
```

Figyeljük meg, hogy szokásunktól eltérően megadtuk a signal-ban lévő változó nevét is – máskor csak a típusát szoktuk. Deklaráljuk és valósítsuk meg a két publikus slot-ot:

```
int Dialog::duration() const
{
    return m_duration;
}

void Dialog::setDuration(const int &d)
{
    if(d != m_duration){
        m_duration = d;
        emit durationChanged(d);
    }
}
```

Ha mindezzel megvagyunk, rátérhetünk a programunk lényegi részére, amely elsősorban a Dialog osztály konstruktorában helyezkedik majd el. Írjuk be a konstruktorba az alábbi sorokat, aztán majd megbeszéljük, hogy melyik mire jó:

```
QQuickView *qView = new QQuickView();
QWidget *qmlScene = QWidget::createWindowContainer(qView, this);
QVBoxLayout *layout = new QVBoxLayout();
layout->addWidget(qmlScene);
ui->frame->setLayout(layout);
qView->setSource(QUrl("qrc:/stuff/qmldialog.qml"));
qView->setResizeMode(QQuickView::SizeRootObjectToView);
qView->show();
```

Két fejléc is kellene fog: a <QQuickView> és a <QVBoxLayout>. És akkor következhet a magyarázat.

Az első sorban elsősorban egy olyan objektumot példányosítunk, amely alkalmas arra, hogy hagyományos Qt programon belül futtathassunk Qt Quick alkalmazást. Figyeljük meg, hogy ez is egy View, azaz nézet. A modellje maga Qt Quick program, illetve annak kimenete. A második sorban szereplő statikus tagfüggvény az, amelyik miatt 5.1-es Qt kellett. Ez a tagfüggvény alkalmas arra, hogy a QQuickView osztályú, a QWidget osztályú objektumok közé nem illeszkedő, azok egymásba-pakolhatóságáról mit sem sejtő objektumoknak egy olyan tartalmazót, angolul container-t adjon, amely már QWidget osztályú, azaz el tudjuk helyezni a hagyományos elemeink között.

A qmlScene objektumunkat remekül elhelyezhetnénk például a Dialog felületén, de a méretének szabályzásával gondjaink lennének. Létezik ugyan, és mi is használjuk pár sorral lejjebb a setResizeMode() tagfüggvényt, de közvetlenül a Dialog felületére helyezve a qmlScene-t bajos volna a méret beállítása. Így aztán a QFrame osztályú frame osztályban próbáljuk elhelyezni az újdonsült widget-ünket, de a QFrame osztály nem képes widget-eket közvetlenül fogadni. Elrendezés (layout) viszont beállítható neki, és ezt kihasználjuk: először

példányosítunk magunknak elrendezést, majd a `qmlScene` objektumunkat ráhelyezzük az elrendezésre. Végül a `frame` objektum gondjaira bízunk az elrendezést. A `setLayout()` tagfüggvény dokumentációja szerint a függvénnyel beállított elrendezés a `QFrame` osztályú objektum tulajdonává válik, ezért nem törtük magunkat az elrendezés példányosításakor szülőobjektum megadásával.

Készen áll hát az ablak, ami a QML-nyelvű programot futtatja majd, így a `QQuickView::setSource()` tagfüggvénnyel az erőforrásfájlból megadjuk a futtatandó `qml`-fájlt. Megadjuk, hogy a QML forrásban megadott gyökérobjektum (a `Rectangle`) mérete alakuljon át a `QFrame`, pontosabban a benne lévő `QVBoxLayout` méreteinek megfelelően. Ha mindez megvan, akkor `show()`-time!

A programunk megyeget, épp csak azt nem tudja még, amiért írni kezdtük: nem tud még beszélgetni egymással a Qt és a Qt Quick. Először az oldjuk meg, hogy a Qt Quick értesüljön a Qt mondanivalójáról.

Elsőként írjuk meg a „faster rotation” feliratú objektum slot-ját:

```
void Dialog::on_checkBox_stateChanged(int arg1)
{
    if(arg1 == Qt::Checked)
        setDuration(500);
    else
        setDuration(5000);
}
```

A `setDuration()` tagfüggvény hívása ugye együtt jár a `durationChanged()` signal emittálásával. Ezt a signal-t igyekszünk majd elkapni a Qt Quick-en belül. A Quick a *környezetéből* érkező signal-okat figyeli, meg kell adnunk hát, hogy mi számít a környezetének. A `Dialog` osztály konstruktorában, akár még a QML-forrásfájl megadását megelőzően helyezzük el az alábbi sort (meg a fájl elejére a `<QQmlContext>` fejléceket):

```
qView->rootContext()->setContextProperty("dialog", this);
```

Ezzel azt mondtuk meg a `QQuickView` osztályú nézetnek, hogy a környezete maga a `QDialog` osztályú objektum (`this`), azaz az ebből az objektumból érkező signal-okat kell figyelnie. Azt is beállítjuk, hogy a környezetre "dialog" néven szeretnénk hivatkozni a QML-nyelvű programban.

A `qml`-fájlban kevés dolgunk volna, ha csak egy egyszerű tulajdonság értékét szeretnénk Qt-ból változtatni. A `CheckBox` szövegének változtatgatása annyival elintézhető lenne (jó megírt `Q_PROPERTY`-t feltételezve), hogy a `text:` jellemző értékéül `dialog.text`-et adunk meg – azonnal át is íródik. Nekünk ez esetben nem elegendő az animáció `duration` jellemzőjét megváltoztatni, mert az a *már futó* animáción nem változtat. (Ha ilyen esetekben nem érezzük a nyelv deklaratív mivoltát teljesen konzekvensnek, nos, akkor..)

Így aztán a `qml`-fájl végére, de még az utolsó záró kapcsos zárójelt megelőzően helyezzük el az alábbi blokkot:

```

Connections {
    target:dialog
    onDurationChanged: {
        cbAnimation.stop()
        cbAnimation.duration = animationDuration
        cbAnimation.start();
    }
}

```

Ez a rész felel meg a Qt connect utasításának. Megadjuk, hogy kitől várjuk a signal-t: a `dialog` nevű környezettől. A signal nevének megfelelő `onDurationChanged:` blokkba helyezzük el a tennivalókat – a slot-ok nevének megválasztásakor nem élvezzük azt a fajta szabadságot, mint a Qt-ban, ami talán nem is baj. Talán még rémlik, hogy a Qt-ban a signal deklarálásakor kivételesen megadtuk a signal-ban lévő változó nevét is. Hát, most látjuk, hogy miért: így tudunk rá hivatkozni a QML-nyelvű alkalmazásunkban. Természetesen a `Connections` blokkban több környezet több signal-ját is kezelhetnénk, de hát most nekünk csak ennyi van. Mindenesetre a program lefuttatásakor a Qt-ban lévő `checkBox` pipálásával a Qt Quick-ben lévő `CheckBox` forgása mostantól gyorsítható és lassítható. Ha a hörcsög szédülős, a program kipróbálása előtt fordítsuk másfelé a monitort.

Folytassuk azzal, hogy tudatjuk a nagyvilággal, hogy a `qml`-fájlban bujkáló `CheckBox`-nak milyenre változott az állapota. A `CheckBox` tulajdonságai között (talán az `anchors` sor alá, de végső soron mindegy) helyezzük el a két következő sort:

```

signal checkStateChanged(bool checked)
onClicked: cb.checkStateChanged(cb.checked)

```

A signal változójának nevét kötelező megadnunk, bár használni nem fogjuk. A signal-t az `onClicked:` slot-ból bocsátjuk ki. És akkor miért nem használjuk közvetlenül a `CheckBox` QML-típus `clicked()` signal-ját? Mert abból nem derül ki, hogy most épp odatette a felhasználó a pipát, vagy kivette onnan.

Battyogjunk vissza a `dialog.cpp` fájlba, és kicsit törjük a fejünket. A `connect` utasításnak kellene fog egy slot is, ezt gyorsan megírjuk:

```

void Dialog::onQmlCheckBoxStateChanged(bool checked)
{
    if(checked)
        ui->label->setText("checked");
    else
        ui->label->setText("unchecked");
}

```

De kellene fog neki egy `QObject` is, amiből jön a signal. Ilyen honnan vadásszunk? Hát, majd kérünk a nézettől (de előtte elhelyezzük a fájl elején a `<QQuickItem>` fejléctet)!

```

QObject *qmlRoot = qView->rootObject();

```

Ilyen egyszerű? Nem, persze nem. A nézet visszaadja nekünk a gyökérobjektumát, ami a QML-ben lévő `Rectangle`. Nekünk viszont a `CheckBox` kell, az előző gyermekobjektuma. Szerencsére a `QObject`-ek gyermekei jól nevelt kölykök: ha mondod a nevüket, előjönnek. Ha van nevük, ami nem azonos az őket tároló változónévvel. Kullogjunk vissza a `qml`-fájlba, és a `CheckBox` elemnek adjunk nevet az

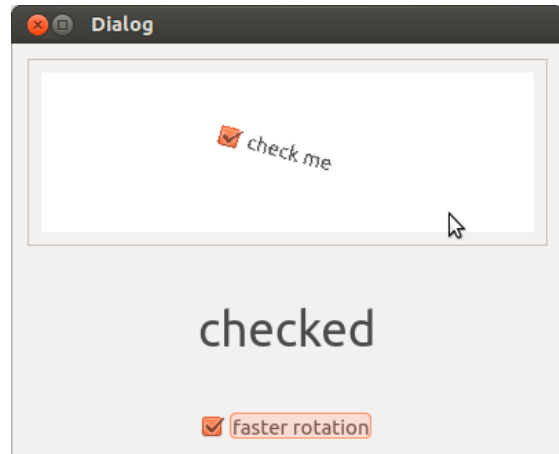
```
objectName: „cbName“
```

utasítással. Éles helyzetben alighanem okos gondolat névként és id-ként azonos karakterláncot megadni, de ezúttal hangsúlyozni szeretnénk, hogy `id` és `objectName` nem ugyanaz. A boldog keresztelő után slattyogjunk vissza ismét a `dialog.cpp`-be, és az előző sor alá helyezzük el a következő két utasítást:

```
QObject *qmlCheckBox = qmlRoot->findChild<QObject* >(„cbName“);  
connect(qmlCheckBox, SIGNAL(checkStateChanged(bool)),  
        this, SLOT(onQmlCheckBoxStateChanged(bool)));
```

Az első előszólítja nekünk a gyermeket, akinek immáron van neve, a második megkialakítja a várva várt kapcsolatot signal és slot között.

A programunk működik, és lassabb egerészeknek kihívást jelenthet a gyorsan forgó CheckBox-ba pipát tenni.



29. ábra: Vaku nélkül sem mozdult be a kép!

Búcsú

Ennyi együtt töltött oldal után búcsút vesz egymástól Olvasó, hörcsög és író. Az író reményét fejezi ki, hogy a könyv elolvasása és a benne lévő programok megírása során az Olvasó nem csak szebb lett, de okosabb is. Az Olvasó alighanem hasonló reményeket táplál, és mind a ketten táplálják a hörcsögöt: hol némi zöldséggel, hol dióval, hol citromos nápolyival. Így szép az élet, igaz?

MILYEN OSZTÁLYOKRÓL, MAKRÓKRÓL ÉS EGYÉB SZÉPSÉGEKRŐL VAN SZÓ A KÖNYVBEN?

B

Button 138, 139, 140

C

Component 140

Connections 146

M

moc 7, 15, 90

MouseArea 137

P

PropertyAnimation 140

Q

QAbstractItemModel 75

QAbstractProxyModel 100

QAction 28, 50, 59, 60, 62, 63, 74, 75, 89

QApplication 10, 47

QBrush 57

QByteArray 107

QCheckBox 68

QColor 57

QCompleter 46, 48, 53, 72, 99, 132

QCoreApplication 49, 61, 85

QDateTime 108, 110

QDebug 15, 16

QDialog 70

QDir 29

QDomDocument 120, 121

QDomElement 120, 121

QDomNode 121

QEventLoop 112, 113

QFile 31, 32, 107

QIODevice 32

QFileDialog 28, 29, 30, 33, 120

QFileInfo 29, 30, 108, 109, 110

QFont 57

QFrame 144

QGraphicsRectItem 58

QGraphicsScene 54, 72

QGraphicsTextItem 58

QGraphicsView 53, 72

QHash 55, 56

QItemSelectionModel 79

QItemSelectionModel 79, 128

QLabel 11, 12, 13, 14, 18, 20, 23, 68, 75

QLinearGradient 57

QLineEdit 15, 20, 23, 47, 52, 75, 97, 98

QList 24, 37, 38, 40, 41, 51, 76

QListView 72, 73, 75

QListWidget 20, 21, 22, 24, 25, 34, 39, 42, 44, 52, 72

QListWidgetItem 24, 25, 27, 45

QMap 15, 17, 18, 32, 55, 56, 57

QMapIterator 58

QMessageBox 28, 31, 92

QMutex 112

QMutexLocker 112

QNetworkAccessManager 104, 105

QNetworkReply 105, 106, 109, 111, 113

QNetworkRequest 106, 111

QObject 16, 29, 37, 53, 70, 75, 87, 115, 146

Q_OBJECT 15

QPair 83, 84

QPointF 58

Q_PROPERTY 92, 93, 144, 145

QPushButton 11, 12, 13, 20, 39, 66, 74

QQuickView 144, 145

QSettings 49, 50, 69, 70, 85, 86

QSharedPointer 37, 39, 40, 41, 83

QSortFilterProxyModel 100

QSpinBox 68

QSqlDatabase 126, 127

QSqlField 131

QSqlQuery 126

QSqlQueryModel 131, 132

QSqlRecord 130

QSqlTableModel 123, 124, 125, 127, 131

QStringList 32

QStack 83, 84

QStandardItem 75, 76, 78, 121

QStandardItemModel 75, 76, 78, 97, 99
QStatusBar 18
QString 13, 15, 17, 18, 22, 28, 37, 38, 55, 56, 83, 84, 116
QStringList 27, 28, 32, 33, 34, 35, 37, 38, 40, 41, 46, 49, 53
QStringListModel 49, 72
QStyledItemDelegate 97
QTableView 73, 74, 78
QTabWidget 53, 75
QTextBrowser 64, 66, 67
QTextCodec 18
QTextEditor 66, 67
QTextStream 31, 121
QThread 112, 114, 115
QTimer 67, 70
QToolBar 62
QTranslator 60
Q_UNUSED 29, 98, 99
QVariant 50, 69
QVBoxLayout 145
QVBoxLayout 144
QWebView 67
QWidget 26, 29, 47, 66, 87, 97, 98, 144

R

Rectangle 137, 138, 139, 140

S

SIGNAL() 17

SLOT() 17

static_cast 98

T

Text 137

ISBN 9 789630 882552 >

E-közigazgatási Szabad
Szoftver Kompetencia Központ



MAGYARY
PROGRAM

Nemzeti Fejlesztési Ügynökség
www.ujszecsenyiterv.gov.hu
06 40 638 638



MAGYARORSZÁG MEGÚJUL



A projekt az Európai Unió támogatásával, az Európai
Regionális Fejlesztési Alap társfinanszírozásával valósul meg.