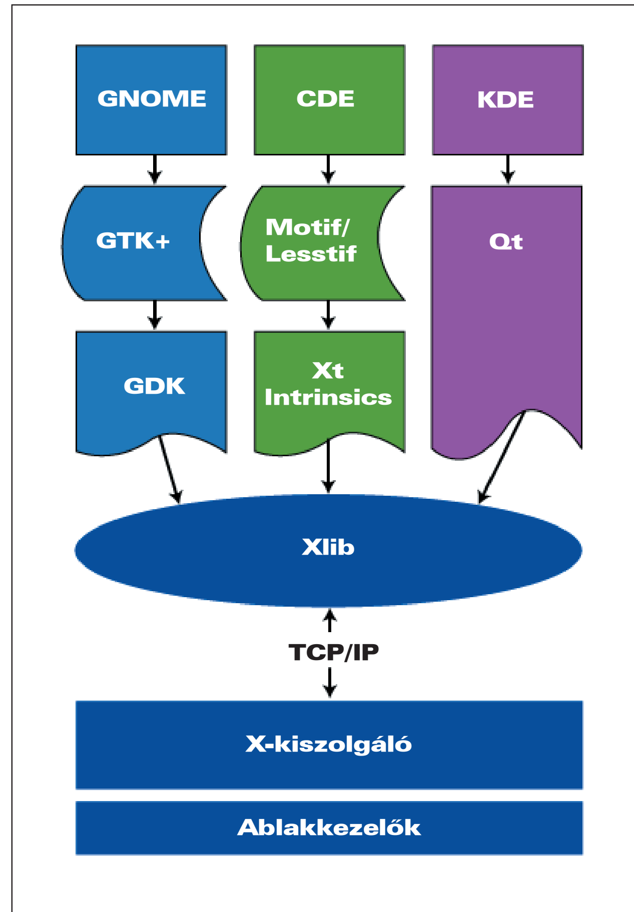
© Kiskapu Kft. Minden jog fenntartva

Könnyed programfejlesztés KDE alá!

Aki e cikket elolvassa, és rendelkezik némi programozói tapasztalattal, minden szükséges információhoz hozzájuthat, ami egy linuxos asztali alkalmazás elkészítéséhez kívánatos.

Számos linuxos alkalmazásfejlesztő eszközkészletből (toolkits) választhatunk. Néhányan úgy gondolják, ez már a Linux bukásának kezdetét jelenti, mások szerint éppen ez a legnagyszerűbb képessége. Én az arany középutat választom, és úgy vélem, jó megoldás, de csak akkor, ha olyat választunk, amely testreszabottan felel meg igényeinknek. Linux alatt a legtöbb grafikus munkafelület (GUI) az X-en alapul, azaz egy olyan kiszolgálóalapú szerkezeten, amely a hálózati gépek számára grafikus alkalmazásaik megosztását teszi lehetővé. Az X szempontjából az ügyfél az az alkalmazás, amely grafikus kimenetét az X-kiszolgálónak továbbítja. Az X-kiszolgáló az alkalmazások kimenetét saját helyi eszközökről (avagy virtuális eszközökről, de írásomban erre terjedelmi korlátok miatt nem térek ki) fogadhatja. A legtöbb esetben az X-kiszolgáló és az X-ügyfél ugyanazon a gépen fut, ennek ellenére ilyenkor is a kiszolgálóalapú modellt használja.
Az X-ügyfelek készítéséhez felhasználható alapszintű eszközkészlet neve Xlib. Az Xlib azonban túlságosan alapszintű és nehézkesen használható ahhoz, hogy önmagában alkalmazások felépítésére használhatnánk fel. Ennek eredményképpen rengeteg Xlibre épülő eszközkészlet készült, amelyekkel X alatt könnyebben lehet GUI-alkalmazásokat létrehozni. Ráadásul ha alkalmazásunkat ilyen magas szintű eszközkészletben készítjük el, arra sem kell figyelnünk, hogy valójában hálózati alkalmazást készítünk (ami grafikus kimenetét egy kiszolgálóhoz küldi). A két legnépszerűbb Xlibre épülő nyílt forrású eszközkészlet a Qt és a GTK+ – a KDE és a Gnome is ezekre épül. A Motif szintén népszerű eszközkészlet (ez ugyan nem nyílt forrású, de Lesstif néven ingyenes változata is létezik). Az *ábrán* bemutatott diagramon ezeket (és néhány más) rendszert, valamint egymás közötti kapcsolataikat láthatjuk. Minél lejjebb helyezkedik el valami a diagramon, annál alacsonyabb szintű API-ról van szó. A magam részéről több okból is a KDE/Qt megvalósítást kedvelem, legfőképpen azért, mivel a jó felhasználói felületre összpontosít, ugyanakkor tiszta és jól megtervezett API.



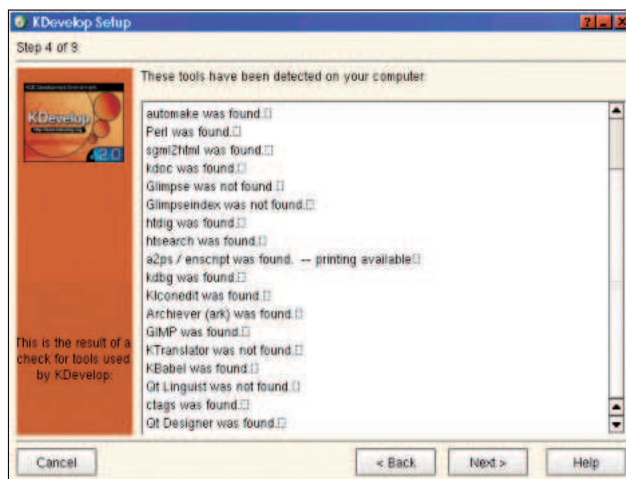
Rokonsági kötelékek a KDE, a Gnome és egyéb eszközök között

Előfeltételek

Ebben a cikkben egy alkalmazást (számológép) fogunk az alapoktól felépíteni, hogy megmutassam, milyen gyorsan és egyszerűen létre lehet hozni. A lépések követéséhez szükségünk lesz néhány eszközre. A legfontosabb a KDevelop, a 2.0.2-es változatot használtam KDE 2.2.2 alatt. Ha a KDevelop telepítő varázslóját használjuk, a rendszer figyelmeztet az esetlegesen hiányzó függőségekre. Látogassuk meg a KDevelop honlapját (➲ http://www.kdevelop.com), vagy nézzük át a terjesztésünkhöz adott programok listáját, és ha a KDE szerepel a terjesztésünkben, telepítsük a KDevelopot.

KDevelop

A legtöbb alkalmazásfejlesztési útmutató megpróbál fejlesztőkörnyezet-független maradni. Én ezt a hagyományt két okból is figyelmen kívül hagyom: a KDevelop ingyenes és a KDE-vel együtt érkezik; a KDdevelop egy már eleve könnyen kezelhető API-t még egyszerűbbé tehet, ráadásul a KDevelop varázslói segítik a fejlesztőket, hogy betarthassák a KDE felhasználói felületekre vonatkozó szabványait. Azoknak, akik nem nagyon szeretnek `Makefile`-okkal bíbelődni vagy beállításfájlokat készíteni (esetleg az egyszerűbb `automake` és `autoconf` sincs ínyükre), nagy segítség lehet a KDevelop, hiszen mindezt megteszi helyettük. Azon szakik számára, akik mégis szeretnek `Makefile`-okkal bíbelődni, a KDevelop ezt is lehetővé teszi. KDevelop-alkalmazásunk a hagyományos `./configure; make; make install` megoldás szerint is elkészíthető. Más szóval a KDevelop használatát tetszés szerint bármikor lehetőségünk nyílik megszakítani.
Amikor első ízben indítjuk el a KDevelopot, az végigvezet bennünket a telepítővarázslón. A varázsló legfontosabb feladata a



1. kép A KDevelop telepítésvarázslója

függőségek vizsgálata (*1. kép*). Vegyük a fáradtságot, és nézzük végig a kimenetet – ha hibajelentéseket találunk benne, segítsünk a gondon, és próbáljuk meg a hiányzó könyvtárakat feltelepíteni. Ezt legkönnyebben a ➲ http://www.rpmfind.com segítségével tehetjük meg (amennyiben RPM-alapú rendszert használunk). Minden hiányzó könyvtárhoz végezzünk keresést az rpmfind honlapon. A találatokból válasszuk ki a terjesztésünknek megfelelő RPM-csomagot, majd telepítsük. Ha ezzel végeztünk, futtassuk újra a KDevelop varázslóját (a `kdevelop --setup` parancsot gépeljük be, vagy a K menüből a ***KDevelop Setup*** nevű menüpontot keressük ki). Ha még mindig lennének hiányzó elemek, a fenti lépéseket szükség szerint ismételjük meg.

Kalculate

Az általam felépített számológép igen egyszerű, és csak a számtani alapműveletekre képes. Az egész elkészítése mindössze két órát vett igénybe (na jó, három órát, de nem siettem). Úgy döntöttem, hogy Kalculatornak nevezem el, de mivel már létezik ilyen nevű alkalmazás, végül a Kalculate név mellett döntöttem. Nem használtam, sőt meg sem néztem a KDE KCalc nevű beépített számológépének forráskódját, amely természetesen több matematikai szolgáltatást nyújt, mint az enyém. Mindig is úgy éreztem, jó lenne, ha a KDE tartalmazna egy egyszerű számológépet, ezért szívesebben látnék néhány felhasználóbarát képességet a Kalculate-ben a legújabb matematikai csodák helyett. Miután elolvastad a cikket, meghívlak egy élő nyílt forrású fejlesztés boszorkánykonyhájába – jöjj, és csatlakozz a csapathoz, és adj újabb képességeket a programhoz! Különleges előnyöket élvezhetsz, ha megemlíted, hogy e cikk közvetítésével jutottál hozzánk (➲ http://sourceforge.net/projects/kalculate).

A Document-View Model

Nem fogunk új projektet kezdeni, hanem kiaknázzuk az Internet nyújtotta lehetőségeket, és elkezdjük használni a Kalculate-et. A következő parancsok kiadásával töltsük le a Kalculate kódját a hivatalos CVS-kiszolgálóról (feltételezve, hogy írási joggal rendelkezünk, készítsünk egy */usr/local/src* nevű könyvtárat, amennyiben esetleg még nem rendelkeznénk vele):

```
cd /usr/local/src
cvs -d:pserver:
➥anonymouscvs.kalculate.sourceforge.net:
➥/cvsroot/kalculate login
```

Ha a parancs jelszót kér, egyszerűen üssünk ENTER-t, ugyanis nincs szükség jelszóra.

```
cvs -z3 -d:pserver:anonymous@cvs.kalculate.
➥sourceforge.net:/cvsroot/
➥kalculate co kalculate
```

Amennyiben az alkalmazást mi magunk szeretnénk felépíteni, nincs szükségünk a forráskódra. Mivel azonban most nem szándékozom soronként leírni a folyamatot, lehetséges, hogy mégis célszerűbb a forráskódot letölteni, és frissen szerzett tudásunkat új képességek hozzáadásával vagy egy új alkalmazáson kipróbálni.

Az általunk alkalmazott tervezési minta az úgynevezett Document-View modell. Három alkalmazásspecifikus osztállyal nyitunk, ezek a következők: a `KalculateApp`, a `KalculateDoc` és a `KalculateView`. E három osztály jön létre, ha a projektet KDeveloppal frissen hozzuk létre.

A Kalculate esetében a sablon által felajánlottakhoz további osztályokat már nem adtam – azért tettem így, mert ily módon könnyebb bemutatni a tervezési mintát, illetve azt, mennyi mindent megold számunkra a KDevelop. A működő számológép előállításához mindössze három osztályt kellett módosítanom, néhány ikont hozzáadnom, és voil☐!

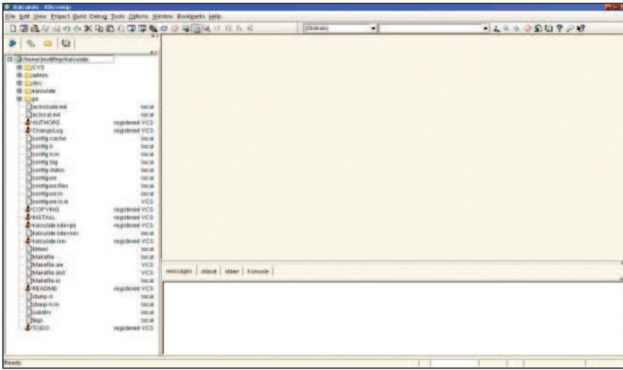
A Document-View legnagyobb hibája, hogy az üzleti logika a dokumentum végére kerül, és a felhasználói felület részt az utolsó nézet tartalmazza. Elméletben ugyanannak a dokumentumnak több nézete lehet. Általános értelemben véve a „dokumentum" valamilyen „dolog" adott nézete. Természetesen ezt a legkönnyebben valamilyen szövegszerkesztő-szerű dologgal tudjuk összeegyeztetni. Ezen elképzelés szerint a dokumentumobjektum jeleníti meg a szövegfájl minden jellemzőjét és tulajdonságát, amelyben a nézetobjektum (view object) felelős a szövegképernyő megjelenítésért és a felhasználói felületéért. A vázban azt is észrevehetjük, hogy a mentés, nyomtatás, megnyitás és lezárás eljárásai előre elkészültek a számunkra. E modellnek egyértelműen a szövegszerkesztő-elképzelés képezi alapját. Csakhogy például a Kalculate esetében ez a megközelítés nem mindig működik. A számológép nem igazán használ fájlokat: egy megadott képességkészlettel rendelkezik, és ezt teszi elérhetővé. Ezért – bár üzleti logikánkat a `KalculateDoc` osztály valósítja meg – nem lesz szükségünk a megnyitás, lezárás, mentés és nyomtatás képességekre. Egyelőre a Kalculate forrásában a szükségtelen gyári kódot (vagyis azt, amit a KDevelop írt be) megjegyzésbe tettem, ahelyett, hogy egyszerűen töröltem volna. Ezáltal, ha egyszer mégis szükség lenne rájuk, a vázlatok megmaradnak.

Az üzleti logika felől nézve a számológépnek lehetővé kell tennie a számok begépelését, e számokhoz műveletek hozzárendelését, és az eredményszámot vissza kell szerezni (illetve a műveleteket belül valóban meg kell valósítani). A felhasználói felület szemszögéből gombokra van szükségünk, amelyekkel számokat és műveleteket vihetünk be, illetve megjelenítő felületre, amelyen az eredményt írjuk ki. Feltételezve, hogy a `cvs checkout`-ot a */usr/local/src* könyvtárban végeztük, a ***/usr/local/src/kalculate*** könyvtár fog létrejönni. A következőket gépeljük be:

```
cd /usr/local/src/kalculate
kdevelop  kalculate.kdevprj &
```

© Kiskapu Kft. Minden jog fenntartva

© Kiskapu Kft. Minden jog fenntartva



2. kép KDevelop projektfájl

A csatolók

```
public slots:
  /**megh vja a repaint()-et minden nØzetre,
   * ami a document objektumhoz
   * kapcsol dik, illetve az a nØzet h vja
   * meg, amelyikben a dokumentum
   * megvÆltozott. Mivel ez a nØzet
   * normÆlisan œjrarajzolja magÆt,
   * kivessz k a paintEvent al l.
   */
  void slotUpdateAllViews(KalculateView
  ➥*sender);

  /** Minden szÆmol gØp szolgÆltatÆs,
   * egy-egy tagf ggvØny minden egyes
   * gombhoz, ahogy az egy val di gØpen
   * is lenne. ValamifØle setNum(int)
   * f ggvØnymegoldÆs helyett ezt a
   * tervezØsi m dot alkalmaztam, nemcsak
   * azØrt, hogy az objektumot
   * biztonsÆgossÆ, hanem hogy szabad
   * t pusœvÆ is tegyem.
   *  AzØrt ezt a megoldÆst rØszes tettem
   *  elınyben, mert  gy a hÆtteret
   *  bÆrmikor anØlk l vÆltoztathatjuk meg,
   *  hogy bÆrmit elrontanÆnk.
   */
  void pressZero();
  void pressOne();
  void pressTwo();
  void pressThree();
  void pressFour();
  void pressFive();
  void pressSix();
  void pressSeven();
  void pressEight();
  void pressNine();
  void pressPlus();
  void pressMinus();
  void pressTimes();
  void pressDivide();
  void pressToggleSign();
  void pressEquals();
  void pressDecimal();
  void pressClear();
```

A fenti parancs megnyitja a projektet. A KDevelop a projektadatokat a *.kdevprj* végződéssel kiegészített projektnév nevű fájlban tárolja. Beállításainktól függően valami olyasmit kell látnunk, amit 2. *kép* mutat. Ha a bal oldalon nem látjuk a projektfát, ellenőrizzük, hogy a ***View, Tree Tools, Views Files*** be van-e kapcsolva. A ***kalculate*** könyvtárra kattintva a projekt fájlszerkezetét láthatjuk (igen, az ott egy ***kalculate*** könyvtár a ***kalculate*** könyvtár alatt, nem kell megtisztítani a monitort!). Itt található az összes forrásfájl. Számunkra most a ***kalculatedoc.cpp*** és ***kalculatedoc.h*** fájlok a legérdekesebbek. Kattintsunk rájuk duplán. Ez a két fájl számológépünk magja: a `KalculateDoc` osztály kezeli az összes számítást. Ez egy olyan virtuális gép, amelyhez valakinek hozzá kell fűznie az irányítófelületet.
Ha a ***kalculatedoc.h***-t megvizsgáljuk, az osztálymegadásokban hamar felfedezhetünk valami furcsaságot, ugyanis akad köztük néhány, amelyik mintha hibás írásmódot követne. Ez vezet el bennünket a csatolók és jelek (signal) világába.

Csatolók és jelek

Az eseményvezérelt alkalmazásokban, mint amilyenek az asztali alkalmazások, egy olyan alrendszert kell felépíteni, amelynek adott részei bizonyos események bekövetkezésekor meghívhatók. Ez az esemény lehet valamilyen felhasználói cselekmény (az egérmutató mozgatása, egy egérgomb lenyomása, egy billentyű leütése stb.). Eseményt egy másik programelem is okozhat, amely valamilyen állapotot jelez (például az időt, a betelt fájlrendszert). Az alkalmazások egyik általános grafikus eleme (widget) a gomb, mellyel könnyen választhatunk ki eseményeket. A KDE világában az eseményeket jeleknek (signal), az eseményeket kezelő függvényeket pedig ***csatolóknak*** vagy eseménykezelőknek (slot) nevezzük. Fejlesztőként lehetőségünk nyílik jeleket és csatolókat készíteni, illetve megadhatjuk, hogy melyik jel melyik csatolóhoz csatlakozzon. Csatolóinkat beépített jelekhez is csatolhatjuk (és megfordítva), vagy beépített csatolókhoz beépített jeleket rendelhetünk.
A beépítettség jelen esetben azt jelenti, hogy a KDE-t vagy a Qt könyvtárosztályokat használjuk, amelyek előre meghatározott csatolókkal és jelekkel rendelkeznek.
A Qt (és így a KDE) a hibás írásmódot is elfogadja, mivel a szabványos C++ előfeldolgozó `#define` fordítói utasításaival a `signals:` megadásokat `protected:`-del helyettesíti, a `slots` és `emit` megadásokat pedig törli. A kiegészítő utasításformára az előfeldolgozó előtt futtatott `moc`-nak van szüksége. A Meta Object Compiler (moc) egy olyan program, amely a Qt könyvtárral együtt érkezik, és a `QObject` fejállományain kell lefuttatni, hogy az osztályainkhoz tartozó különleges tagfüggvényeket elkészítse, és így azok `QObject` alosztályokként felhasználhatók legyenek. Minden objektumnak, amely csatolókat és jeleket használ, kötelezően a `Qobject` alosztályának kell lennie, osztálymeghatározásában pedig szerepelnie kell a `Q_OBJECT` makrónak (amint azt a ***kalculatedoc.h*** fájlban láthatjuk is). A `QObject` szuperosztály egyik tagfüggvénye a `connect`. Ezzel a tagfüggvénnyel kapcsolhatjuk össze jeleinket és a csatolókat.
Jelek létrehozásához osztálymeghatározásunk `signals:` bejegyzése alatt meg kell adnunk őket. A jelek esetében valójában nincs szükség további tagfüggvények elkészítésére. Mindössze annyit kell még tennünk, hogy az `emit`-et a jel nevével meghívjuk, és máris az összes hozzácsatolt csatoló meghívódik – a `moc` az összes jeltagfüggvényt elkészíti nekünk.
A csatolókat ezzel szemben létre kell hozni. Csatolójainkat megadhatjuk a `public slot` vagy a `protected slot` alatt, majd a *.cpp* fájlban hagyományos tagfüggvényként létrehoz-

hatjuk őket. Ezek után már tetszés szerint csatlakozhatunk hozzájuk, valahogy így:

```
connect(this, SIGNAL( mySignal() ),
➥this, SLOT( mySlot() ) );
```

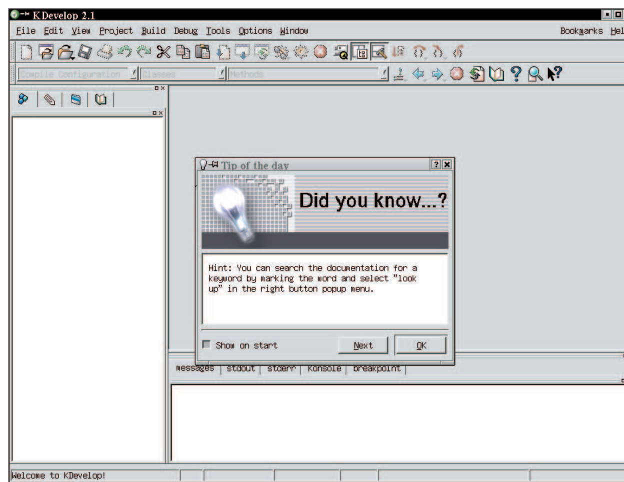
Feltételezve, hogy ugyanazon az osztályon belül van egy `mySignal()` nevű jelünk és egy `mySlot()` nevű csatolónk, a fenti `connect` hívás összefűzi a kettőt, így valahányszor kiadunk egy `emit mySignal()` utasítást, a `mySlot()` meghívódik. Ha csatolónkat valamilyen más objektum jeleihez szeretnénk kapcsolni, az első érték az adott objektum példánya lett volna, a második érték pedig a `SIGNAL()` makró zárójelek között a jel nevével. Ilyen egyszerű. Minden nehéz részt (ideértve fejállományainkon a `moc` futtatását) a KDevelop végzi el helyettünk. Tehát – számológépünkhöz visszatérve – a `KalculateDoc` osztály a csatolók halmaza. Minden csatoló egy-egy olyan műveletnek felel meg, amit a számológéppel elvégezhetünk. A fejállomány az összes csatolót a *listában* látható módon határozza meg. Az egyetlen dolog, ami ezeket a tagfüggvényeket különlegessé teszi, az az, hogy a `public slots:` címke alatt lettek megadva, így a `moc` némi metaadatot készít hozzájuk. A csatolókat ugyanúgy kell megadnunk, mint a hagyományos tagfüggvényeket (vizsgáljuk meg a ***kalculatedoc.cpp***-t, és figyeljük meg e csatolók meghatározásait). A jelek esetében ezt nem kell megtennünk, a `moc` mindenről gondoskodik.

3. kép Kalculate

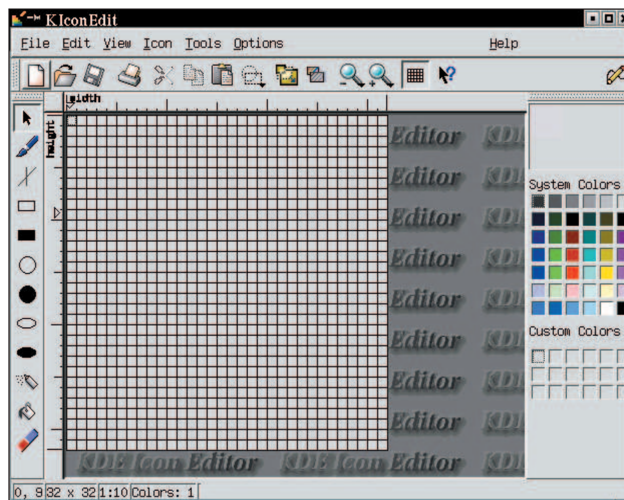
Most már csak egy GUI-ra van szükségünk, ami ezt az osztályt használni tudja. Lépjünk be a `KalculateView`-ba (reményeim szerint a forráskód megtekintése végett már amúgy is beléptél ide). A ***kalculateview.h*** fájl határozza meg a GUI-t létrehozó osztályt. A `protected` kulcsszó alatt láthatjuk, hogy néhány kinézetkezelő, LCD-megjelenítő és pár gomb már meg van adva. A KDevelop beépített sablonkódjához hozzáadtam egy fájlt, a neve ***kalculatesizes.h***, a tartalma pedig itt olvasható:

```
#ifndef KALCULATE_SIZES_H
#define KALCULATE_SIZES_H

#define BUTTON_WIDTH 35
#define BUTTON_HEIGHT 35

#define LAYOUT_SPACING 4

#define MAX_WIDTH (BUTTON_WIDTH * 5)
        + ((LAYOUT_SPACING *2) * 4)
#define MAX_HEIGHT (BUTTON_HEIGHT * 5)
        + ((LAYOUT_SPACING *2) * 4)

#endif // SIZES_H
```

Ez tulajdonképpen számológépünk méreteit állítja be. Nem szerettem volna újraméretezhető számológépet készíteni, ugyanakkor azt akartam, hogy a gombok méretét könnyedén lehessen változtatni. Ezért a méreteket ebben a fájlban adom meg. Ezek után mindössze ezt a fájlt kell módosítani, majd újrafordítani a programot, és máris eltérő gombméretű számológépet kaptunk



4. kép A KDevelop induló ablaka



5. kép Az ikonkészítő felülete

(a jövendőbeli változatok esetleg futásidőben is lehetővé tehetik, ezt azonban bizonyos okok miatt haboztam megtenni). A `KalculateView`-ban található létrehozó függvény (a ***kalculateview.cpp***) kelti életre az alkalmazást, a `setMaximumSize()` meghívása pedig átméretezhetetlenné teszi. A méretszabály szintén segít, ez azonban csupán ajánlás. Itt megállnék egy pillanatra, és ejtenék pár szót a kinézetkezelőkről (layout managers). Minden egyes GUI-elemnek vannak tagfüggvényei, amelyek a méreteit állítják (magasság, szélesség, relatív helyzet stb.). Amikor az alkalmazás elindul, illetve valahányszor átméretezik, nem lenne vicces, ha mindig nekünk kellene megírnunk azt a kódot, amely átméretezi és áthelyezi. A kinézetkezelő nélkül kezdeti állapotban az alkalmazás minden egyes gombja előre rögzített x:y koordinátákra kerülne. Órákat venne igénybe, ha a számológép méretét meg akarnánk változtatni. Ezért az elhelyezési tulajdonságok rögzítése helyett az elemkészleteket kinézetkezelőkben jegyezzük be, amelyek bizonyos szabályokat követnek, például a `QVBoxLayout` az elemeket függőleges oszlopokba rendezi. Valahányszor új elemkészletet adunk a hozzá az `addWidget()` tagfüggvénnyel

```
outerL0ayout->addWidget(output,1);
```

© Kiskapu Kft. Minden jog fenntartva

© Kiskapu Kft. Minden jog fenntartva

az az új elemkészletet önműködően az előző alá helyezi. A `QHBoxLayout` ezzel szemben az új elemkészletet mindig az előző jobb oldalára helyezné. A második érték az úgynevezett nyújtási arány (stretch factor). A nyújtási arány alapvetően azt határozza meg, hogy ez az elemkészlet mennyi helyet foglaljon el az ugyanebben a kinézetkezelőben található egyéb elemkészletekhez képest. Így tehát ha valamennyi `1`, mindannyian ugyanakkorák lesznek (feltéve, hogy a `setMaximumSize` ezt az értéket nem írja felül). A nyújtási arányok összege adja meg a mértéket. Így ha van két elemkészletünk, és az egyik `1`-es, a másik `2`-es nyújtásaránnyal bír, a második kétszer akkora lesz, mint az első.
Igazán nagyszerű benne, miként a `KalculateView` is példázza, hogy kinézetkezelőkbe másik kinézetkezelőt is helyezhetünk. Ez lehetővé teszi, hogy igen összetett kinézeteket alkossunk. Kalandra fel, próbálkozz egy kicsit a Kalculate kódjával – fedezd fel, mit tudsz alkotni!
Az alkalmazás futtatásához a következő lépéseket kell megtennünk: válasszuk az ***Autoconf*** és ***Automake*** pontot a ***Build*** menüből. Majd válasszuk a ***Build*** menü ***Configure*** pontját. Amikor értékeket kér, a `--prefix=`*`kde alapk nyvtÆrad`*-at adjuk meg. Ez lehetővé teszi, hogy az alkalmazást telepítsük is, ami elengedhetetlen, ha azt szeretnénk, hogy az alkalmazás ikonja megjelenjen. Mandrake-et futtató gépem KDE alapértelmezett könyvtára a */usr*. Más terjesztésnél eltérő lehet. Ezután válasszuk a ***Build*** menü ***Execute*** pontját, és az alkalmazás lefordul, majd lefut. A *3. képen* láthatóhoz hasonló látványnak kell fogadnia minket, bár a bal felső sarokban szerplő ikon – amíg fel nem telepítjük – valószínűleg hiányozni fog. A telepítéshez rendszergazdaként a következőket gépeljük be:

```
cd /usr/local/src/kalculate
make install
```

Remélem, mindenkit a helyes irányban indítottam el. Valójában éppen csak a felszínét kapargattuk meg a KDE-vel végezhető feladatok hatalmas tárházának, reményeim szerint azonban sikerült megmutatnom, hogy milyen gyorsan el lehet indulni a KDeveloppal végzett munka rögös útján. Csaknem az összes háttérmunkálatot átvállalja tőlünk, és olyan alkalmazást készít, amely parancssorból lefordul anélkül, hogy a KDevelop szükséges lenne hozzá (ami nagyon jó, ha a kódot terjeszteni szeretnénk). Szívesen várom leveleiteket, amennyiben esetleg kérdésetek lenne, és ezúton is bátorítanék mindenkit, hogy csatlakozzon a Kalculate-csapathoz, és adjon hozzá néhány további képességet.

Linux Journal 2002. június, 98. szám

Jason Mott
(jmott@users.sourceforge.net)
független programozó és tanácsadó. Jelenleg a rochesteri ElementK-nak dolgozik (➲ http://www.elementk.com) New Yorkban, hálózati oktató honlapjuk kialakításában segíti őket. Részidőben Linux-tanácsadó, és ha szabadideje engedi, linuxos asztali alkalmazásokat készít.

Az fsck rendszerellenőrző és -helyreállító eszköz

Tudjuk, hogy rendszerünk sohasem omlik össze, a valóságban azonban a legrosszabb időnként mégis bekövetkezhet. Ha nincs szünetmentes áramforrásunk, elég lehet, ha valaki véletlenül kihúzza a hálózati csatlakozót vagy áramszünet áll be. Ha ez megesik, előfordulhat, hogy a fájlrendszer megsérül. Ha a rendszer a leállás pillanatában éppen adatokat írt egy állományba, jó eséllyel csonka fájl és adatszemét marad vissza. Bárki, aki hosszabb ideje használ Linuxot (vagy Unixot), tanúsíthatja, hogy ezen a területen a rendszernek nincs szégyellnivalója. Az ext2 fájlrendszer az ilyen nehézségek kezelésében alapvetően jobb, a baleset időről időre mégis bekövetkezik.
Amikor valami ilyesmi történik, a rendszer a következő indítási próbálkozásnál észreveszi, hogy a lemezterület nem szabályosan lett kifűzve (gondoljunk csak a Windows ScanDisk programjára).
A program, amelyik a rendellenességet észleli, ugyanaz, mint amelyik a helyreállítást végzi. A program neve fsck (ez egyébként a file system check, vagyis a fájlrendszer-ellenőrzés rövidítése). Az imént az ext2 fájlrendszer melletti érvet említettem azért, hogy most valami érdekeset mutathassak. Az `fsck` parancs nem az fsck programot futtatja. Ez csak a felülete a különböző fájlrendszerek javítóeszközeinek.
Az ext2 fájlrendszerhez tartozó program neve fsck.ext2, bár lemezünkön a program ext2fs néven található. Mivel mindkettő ugyanaz, mindegy, melyiket használjuk.
Mivel az fsck önműködően elindul, nem feltétlenül kell tudnunk róla, hogy kézi indítására is lehetőség van. Ehhez azonban az szükséges, hogy az ellenőrizni kívánt fájlrendszert kifűzzük. A másik lehetőség, hogy a rendszert egyfelhasználós módban indítjuk (`linux single` beírása a LILO parancssorába). Egy fájlrendszer ellenőrzéséhez a következő parancsot használhatjuk:

```
fsck /dev/hda5
```

Ez a legegyszerűbb módja a lemezellenőrzésnek. Amennyiben a lemez kifűzésével nem volt gond, az fsck visszatér. Ha mindenképpen végre kívánjuk hajtatni az ellenőrzést, a `-f` kapcsolót kell használnunk.

```
fsck -f /dev/hda5
```

Ha a lemezzel gondok merülnek fel, és az fsck nem biztos benne, hogy mit tegyen, a változtatások előtt a felhasználó megerősítését kéri. Ha mindent rá akarunk hagyni, a `-y` kapcsolót használhatjuk, ezzel minden esetleges kérdésére igenlő választ adunk:

```
fsck -f -y /dev/hda5
```

Amennyiben az fsck úgy találja, hogy egy fájlrészletet képtelen visszafűzni a rendszerbe, a fájldarabot a *lost+found* könyvtárba helyezi. Mivel az fsck szakértő módon végzi feladatát, általában semmit sem fogunk találni benne. Ennek ellenére, ha rendszerünk nem szabályosan lett leállítva, és arra kényszerült, hogy az fsck-t futtassa, talán érdemes megnézni a *lost+found* könyvtárat, hátha mégis valami „elveszett és megkerült”.

Részlet Marcel Gagné: Linux-rendszerfelügyelet című könyvéből